Semantic Query Optimization for Object Databases

J. Grant[1], J. Gryz[2], J. Minker[2,3], L. Raschid[3,4]

Department of Computer and Information Sciences[1]
Towson State University
Department of Computer Science[2], Institute of Advanced Computer Studies[3],
and Maryland Business School[4]
University of Maryland, College Park

E-mail: grant@midget.towson.edu {jarek,minker}@cs.umd.edu louiqa@umiacs.umd.edu

Abstract

We present a technique for semantic query optimization (SQO) for object databases. We use the ODMG-93 standard ODL and OQL languages. The ODL object schema and the OQL object query are translated into a DATALOG representation. Semantic knowledge about the object model and the particular application is expressed as integrity constraints. This is an extension of the ODMG-93 standard. SQO is performed in the DATALOG representation and an equivalent logic query, and subsequently an equivalent OQL object query, are obtained. SQO is based on the residue technique of [3]. We show that our technique generalizes previous research on SQO for object databases. It can be applied to queries with structure constructors and method application. It utilizes integrity constraints about keys, methods, and knowledge of access support relations, to produce equivalent queries, which may have more efficient evaluation plans.

1. Introduction

In this paper we show how to apply *semantic query optimization* (SQO) techniques to query processing in object databases. SQO uses semantic knowledge, in the form of integrity constraints (ICs), to reformulate an object query into an equivalent query that can be evaluated more efficiently. SQO has been applied to relational and deductive databases [3]. In particular, it was shown that a logic-based approach using the method of partial subsumption is a general technique that encompasses various special cases of SQO considered by other researchers. Here, we demonstrate that SQO can be adapted to object databases.

*This research has been supported by the following grants: NSF IRI 9300691, ARPA/ONR 92-J1929

Our approach to SQO for object databases is the following: We transform an ODL object schema, into a relational schema (DATALOG representation), so that the object semantics is reflected as integrity constraints in the DATALOG representation. Correspondingly, each OQL object query is transformed to a query on the DATALOG representation. SQO is performed at the DATALOG level to produce an equivalent (and optimized) DATALOG query. The optimized DATALOG query is then transformed back to an optimized OQL object query. We are able to cover a large fragment of the ODMG-93 [4, 5] OQL language.

We note, in particular, that the ODMG-93 standard does not include proper facilities for the expression of integrity constraints. SQO can be done using only the integrity constraints implicit in the object model. In this paper, we show that the ability to express additional integrity constraints (particular to the application), explicitly, as an extension to the object model, would be useful for increasing the capability of a system to perform query optimization.

Much work has been done to develop techniques for SQO in deductive and relational databases [3, 8, 10, 11, 13, 15]. Recently, the issue of SQO in the context of object databases has also been considered [1, 12, 14, 16, 17]. We extend this work in the following ways: Our technique allows optimization of a much larger class of queries than considered before. In particular, queries with method application and structure constructors can be optimized. We use a uniform DATALOG representation for queries, schema and integrity constraints, and present explicit algorithms for each step of the optimization process.

Using several examples, we illustrate the following: integrity constraints can be used to identify contradictions and hence eliminate queries that need not be evaluated. Semantic information on the type hierarchy can be used to reduce the scope of a query to a subclass; this is a useful optimization technique in object databases that maintain the extents of classes. Key constraints identify identical objects and

can be used to eliminate unnecessary join operations and object accesses from the object base. Integrity constraints about methods can be used to optimize queries in which these methods are applied. The knowledge of access support relations [9] can be used to eliminate joins, or to produce alternate equivalent queries. These queries could not be produced without the application of the semantic knowledge. SQO is therefore an enhancement to traditional optimization. It produces alternate equivalent queries for which there may be more efficient query evaluation plans. The generation of plans and the actual selection of the optimized query to be evaluated would be determined by a cost-based physical optimizer.

The paper is organized as follows: Section 2 provides background on the residue method of [3] which is used in our optimization technique. Section 3 reviews the ODMG-93 data model. In Section 4, we first present an overview of our technique, and then we present the schema and query transformation. Several examples are used in Section 5 to illustrate the applications of the technique. The paper concludes in Section 6.

2. Background

We first present background on the residue method developed in [3] for SQO in deductive databases. The basic idea of the residue method is the use of partial subsumption, to attach integrity constraint fragments, called residues, to relations during the semantic compilation phase, before any queries are posed. After a query is presented to the database system, residues are used, where possible, to transform the query to one that is semantically equivalent to the original query, but whose execution may be faster, i.e., a more efficient evaluation plan. A residue is intuitively a formula that is true for any query containing a relation name to which the residue is attached. In particular, if a residue is a single literal, it may be added (or removed from a query that already contains it), without affecting the answer to the query.

Using an example, we informally show how residues are used for SQO. The reader is referred to [3] for details. We use standard notation for DATALOG: predicates and constants start with lower-case letters; attribute names, which play the role of variables, start with upper-case letters. In ODMG-93, the convention is reversed. In the relational database, both relation names and attribute names start with upper-case letters.

Example 1 *Let the relational database contain the following relations:*
Student(St_id, Name), Takes_section(St_id, Sec#), Faculty(Sec#, Fac_id, Age). *Assume also, that there is an integrity constraint that all faculty members are more than 30 years old, expressed as follows, in the DATALOG representation:*

$$IC: \ Age > 30 \leftarrow faculty(Sec\#, Fac\_id, Age)$$

By the method of partial subsumption, as described in [3], with the IC and the relation Faculty, *we can construct the following residue:* {*Age*> 30 ←}, *which is then attached to the relation* Faculty. *Intuitively, it means that any query with the predicate* faculty *must satisfy the condition stated by the residue.*

The following DATALOG query asks for the names of all students taught by professors younger than 18:

Q(Name)← student(St_id, Name),takes_section(St_id, Sec#), faculty(Section#, Fac_id, Age), Age<18

Now, since the query contains the predicate faculty, *the residue applies here, and the following semantically equivalent query is produced:*

Q'(Name)← student(St_id, Name),takes_section(St_id, Sec#), faculty(Sec#, Fac_id, Age), Age<18, Age>30.

The query contains a contradiction. This means that it cannot return any answers or else the database would violate the IC. Hence it need not be evaluated. A syntactic optimizer for DATALOG, upon discovering the contradiction, would halt the processing of the query.

Since the technique described here was developed originally for deductive databases, it can handle views, as well as rules, easily, if they are part of an object database.

3. The ODMG-93 Data Model

The lack of a standard for object databases has been a major limitation to their more widespread use, and also to transferring tools and techniques developed for relational databases to object databases. ODMG-93 [4, 5] describes a standard for object database management systems developed by the members of the Object Database Management Group (ODMG). This standard provides, among other things, a syntax for the Object Definition Language (ODL) and the Object Query Language (OQL).

The ODMG-93 object model can be summarized as follows:

- The basic modeling primitive is the *object*. Every object has an identity referred to as an *object identifier* (OID).
- The state of objects is defined by the values they carry for a set of *properties*. These properties may be either *attributes* of the object itself or *relationships* between the object and one or more other objects.

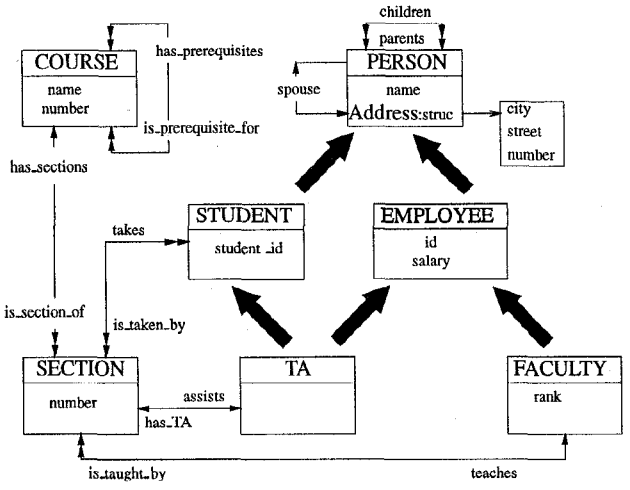

Figure 1. Example database schema. Each box represents a class, heavy arrows indicate class hierarchy, thin arrows describe relationships between classes. Methods are not included in the figure.

- The behavior of objects of a type is defined by a set of *methods* (also called *operations*), that can be executed on an object of the type.
- Objects can be categorized into *classes (or types)*. All objects of a given type exhibit common behavior and a common range of states. Object types are related in a subtype/supertype graph. All properties, and operations defined on a supertype are inherited by a subtype.

The schema in Figure 1, which is a slight modification of the example from [4], illustrates the major features of the ODMG-93 data model.

We clarify the difference between attributes and relationships. Only relationships can relate objects. Attributes do not have OIDs; their individuality is dependent on the individual object to which they apply. Hence, a structure, e.g., Address, which is an attribute of some object, cannot be shared by, (i.e., be an attribute of), another object. Similarly, the value of an attribute must be changed explicitly and cannot change dynamically, by a change of OID. We do not make this assumption in the DATALOG representation, i.e., we assume that all attributes of types other than base types (such as strings or integers) have OIDs. This modification allows us to obtain a uniform DATALOG representation where each relation (including those representing a structure) includes an OID. This change does *not* affect the semantics of ODMG-93 with respect to SQO, since we do not perform any operations on structures which are not allowed in ODMG-93. This will be clear when we describe the translation to DATALOG.

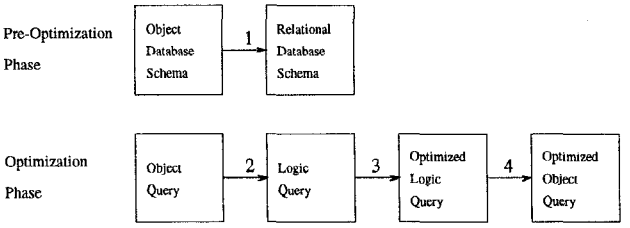

Figure 2. Query optimizations steps. The arrows are not intended to indicate the complete input to each step.

4. Semantic Query Transformation

4.1. Overview of the Technique

We perform SQO for the first-order logic representation of a query expressed in OQL. Our optimization technique has several steps as shown in Figure 2. Prior to optimization, the object database schema, described in ODL, is translated into an equivalent DATALOG representation (Step 1). It is similar to translations in [2, 17]. This translation is intended to capture the semantic information encoded in the object data model, such as object identity, inheritance, types of relationships between classes (one-to-one, one-to-many, many-to-many), and keys. We assume that all integrity constraints are expressed in the DATALOG formalism, including those particular to an application. During optimization, each OQL query is then translated into its logical representation (Step 2). Next, we apply the residue method to obtain all queries that are semantically equivalent to the original logic query (Step 3). Finally (Step 4), we obtain an optimized OQL query, and possibly more than one query.

We emphasize, that the input to the transformation of Step 4 contains both the optimized logic query, and the original OQL query. This is necessary if we want to retain all of the extralogical features of OQL, such as structure constructors. The transformation of Step 2, although sufficient for SQO, does not preserve these features, hence they cannot be retrieved into the optimized OQL query from the logic query alone. To guarantee the full equivalence of the original and the modified OQL queries with respect to structure constructors, we reverse map *changes* done during the optimization of the logic query to *changes* of the OQL query. In other words, Step 4 is not a translation of the logic query, but a modification of the original OQL query according to the mapping of the changes.

The output of SQO in our approach may contain *several* semantically equivalent queries. Each query should then be evaluated by a conventional cost-based optimizer, and the query whose evaluation plan has the lowest cost is chosen for evaluation in the database. Since the number of semantically equivalent queries generated by SQO can be large, heuristics are required. Such heuristics would guide the query transformation process, so that only promising transformations are generated. Many of the heuristics developed for SQO in relational databases [3, 15] are applicable. Guidelines for the efficient search of promising semantic transformations in object databases have been suggested in [14, 12]. However, a more complete set of heuristic rules for SQO in object databases is implementation dependent, and we do not consider this issue here.

We note here that the overhead incurred by the rewriting phases (Steps 1, 2, and 4), in our technique, is negligible compared to the cost of SQO itself (Step 3). The complexity of Step 1, schema transformation, is linear with respect to the number of classes, relationships, structures, and methods defined in object database. Although this cost may be non-trivial for large databases it would be amortized over a large class of queries. The complexity of Steps 2 and 4, query transformation, is linear with respect to the size of the query. Step 3, on the other hand, is exponential in the number of integrity constraints applicable to a query and will dominate the entire optimization process.

4.2. Schema Translation

We represent an ODL schema in the DATALOG representation by relations and integrity constraints. Each class, structure, relationship and method of the object data model is represented as a relation; all facts about the class hierarchy, object identity, relationships, and keys are expressed by means of integrity constraints.

RELATIONS

Each class, structure, relationship and method defined in an ODL schema becomes a relation in the DATALOG schema. The translation for the schema of Figure 1 is provided in [7]. The following rules specify the translation:

1. Assume that a class C has the following attributes in this order:

   simple attributes $A_1, ..., A_n$;

   structure attributes $S_1, ..., S_m$

   For each such class C, create the following 1+n+m - ary relation:

   $$c(OID, A_1, ..., A_n, OID_{S_1}, ..., OID_{S_m})$$

   where OID_{S_i} is an OID of a structure of type S_i

   Note: Subclasses inherit the attributes of their superclasses in an unambiguous way.

2. For each structure S with attributes as above, create the following 1+n+m - ary relation, as was done for classes:

   $$s(OID, A_1, ..., A_n, OID_{S_1}, ..., OID_{S_m})$$

3. For each relationship R between classes C_1 and C_2, create the following binary relation:

   $$r(OID_{C_1}, OID_{C_2})$$

   where OID_{C_1} and OID_{C_2} are OIDs of tuples respectively from the relations representing C_1 and C_2.

4. For each method M defined on objects of class C, with user provided arguments $A_1, ..., A_n$, and which returns a value $Value$, create the following $n + 2$-ary relation m:

   $$m(OID_C, A_1, ..., A_n, V)$$

   where $V = Value$ if $Value$ is a base value; else, $V = OID_{Value}$, where OID_{Value} is an object identifier of the structure $Value$.

INTEGRITY CONSTRAINTS

Integrity constraints capture semantic information of the object schema. For object databases that allow explicit representation of particular integrity constraints, such constraints should also be translated into DATALOG. Integrity constraints expressing functional dependencies should be included here as well.

1. OID Identification

   - For each relation r obtained from a relationship R between classes C_1 and C_2, the following hold:

     $$c_1(OID_1, A_1, ..., A_n) \leftarrow r(OID_1, OID_2)^1$$
     $$c_2(OID_2, B_1, ..., B_m) \leftarrow r(OID_1, OID_2)$$

     where $A_1, ..., A_n$ and $B_1, ..., B_m$ are attributes of c_1 and c_2, respectively.

   - For each class C containing the OID_S of a structure S as an attribute, the following holds:

[1] We assume that variables that appear in the head of an integrity constraint but do not appear in the body of that constraint, e.g., $A_1, ..., A_n$, are existentially quantified.

$$s(OID_S, B_1, ..., B_n) \leftarrow$$
$$c(OID, OID_S, A_1, ..., A_m)$$

- For each method M containing the OID of a class C the following holds:

$$c(OID, A_1, ..., A_n) \leftarrow$$
$$m(OID, B_1, ..., B_m, V)$$

where m is the relational representation of M

2. Subclass hierarchy

For each pair of classes C_1 and C_2, where C_2 is a subclass of C_1, the following holds:

$$c_1(OID, A_1, ..., A_n) \leftarrow$$
$$c_2(OID, A_1, ..., A_n, A_{n+1}, ..., A_m)$$

3. Inverse relationships between classes

For each pair of relations r_1 and r_2 obtained from an inverse relationship between C_1 and C_2, the following hold:

$$r_1(OID_1, OID_2) \leftarrow r_2(OID_2, OID_1)$$
$$r_2(OID_2, OID_1) \leftarrow r_1(OID_1, OID_2)$$

4. One-to-one constraints

For each relation r obtained from a one-to-one relationship R, the following hold:

$$OID_2 = OID_3 \leftarrow$$
$$r(OID_1, OID_2), r(OID_1, OID_3)$$

$$OID_2 = OID_3 \leftarrow$$
$$r(OID_2, OID_1), r(OID_3, OID_1)$$

4.3. Query Translation

Step 2 in Figure 2 is the query translation. The input to this procedure is an OQL query and the relational representation, (from Step 1), of the ODL schema. The output is the DATALOG representation of the OQL query. Not all of the features of OQL can be represented in DATALOG. A DATALOG query cannot create new objects (i.e., it lacks constructors). Thus, an OQL query and its DATALOG representation can be considered equivalent only in the sense that the DATALOG query retrieves precisely those tuples, (according to the schema transformation), representing the objects retrieved by the OQL query. A formal proof of the correctness of our transformation is beyond the scope of this paper.

In this paper, we use the OQL syntax of [4]; the revised OQL definition of [5] does not change the expressive power of the language. We restrict OQL queries in the following ways:

- We consider only **select-from-where** queries. This covers the majority of commonly asked queries.

- We do not translate constructors such as *struct, set, list* etc., or collection expressions such as *unique, count, sum* etc., into DATALOG. Such operators are not relevant for SQO and can be directly carried over into the optimized OQL query. One of the key ideas of our approach is the fact that the optimized DATALOG query, (without constructors, expressions, etc.), is not translated into a new OQL query, since in this case all such constructors and expressions that could not be expressed in the DATALOG query would not be retained. Instead, we map the modifications done in the DATALOG query into corresponding modifications to the original OQL query. Thus, we apply SQO to a larger class of OQL queries, without changing the format of the answer set.

  For clarity of presentation we also ignore set expressions, e.g., *intersect, union, except*, which can be represented in DATALOG.

- The algorithm we present works for unnested queries only. Subqueries appearing in the **where** clause should be unnested as is done in SQL. A canonical form for the representation of OQL queries, and techniques for unnesting certain subqueries from the **select** and **from** clauses have been proposed in [6]. We expect that this unnesting will precede SQO.

The algorithm for translating this restricted OQL query into DATALOG is similar to the algorithm for translating an SQL query into DATALOG. The complete algorithm is provided in [7]. Some points to note in the treatment of OQL queries are path expressions and method names. Path expressions are removed from an OQL query and substituted with "one-dot" expressions, i.e., expressions of the form $X.Y$, where neither X nor Y are path expressions. Since methods are represented by relations in our approach, all references to method names are translated as references to their relational representations in DATALOG. Note that the DATALOG representation of an OQL query has the form: *Q(Projected_Attributes)← Body*. The following example illustrates how the translation is carried out:

Example 2 *Consider a query in OQL against the schema of Figure 1, which has only one-dot expressions.* taxes_withheld *is a method defined on the class* Employee*; it takes one user-provided argument* Rate *and returns a value of type* real.

select z.name, w.city
from x in Student
y in x.Takes

z in y.Taught_by
w in z.Address
where x.name = "john", z.taxes_withheld(10%)<1000

The first step in the algorithm is the translation of all statements in the **from** *clause. This yields the following atoms added to the body of the DATALOG query, where OQL identifiers x, y, z, w play the role of OIDs X, Y, Z, W in the DATALOG representation of the query:*

student(X,...),takes(X,Y),taught_by(Y,Z),address(W,...)

Next, we translate the expressions of the **select** *clause. This requires first identifying classes or structures over which the identifiers* z *and* w *range.* w *has been already identified as ranging over the structure type* Address. *To identify the domain of* z *we use the following integrity constraint created during schema transformation:*

faculty(Z,...)← taught_by(Y,Z)

This integrity constraint states that the second argument of the relationship Taught_by *in ODL is an object from the class* Faculty. *Hence, the following atom is added to the query:*

faculty(Z,...)

We also need to add attributes referred to in the select *clause to the list of* Projected_Attributes *of the DATALOG query. To avoid ambiguity, attributes with the same name in the OQL query are indexed in the DATALOG representation;* name *in* z.name *and* name *in* x.name *are attributes of different objects, so they are represented as* Name_1 *and* Name_2, *respectively. Thus,* Name_1 *and* City *become the* Projected_Attributes.

In the where *clause* x.name = "john" *is translated as:*

$$Name_2 = \text{"john"}$$

Finally, the statement z.taxes_withheld(10%)<1000 *is translated to the following, as specified in the schema translation:*

$$taxes\_withheld(Z, 10\%, V),\ V < 1000$$

The final DATALOG query is as follows:

$$Q(Name_1, City) \leftarrow student(X, Name_2),\ takes(X,Y),\ taught\_by(Y,Z),\ faculty(Z, Name_1, W),\ address(W, City),\ Name_2 = \text{"john"},\ taxes\_withheld(Z, 10\%, V),\ V < 1000$$

Suppose that after the DATALOG query has been optimized, as sketched in Section 2), there are no contradictions in the optimized query. Then, we map the changes introduced in the DATALOG representation of the query, by SQO, back to the OQL query. The only changes that can be made in a DATALOG query are the addition or removal of one or more literals. A deleted (added) literal can represent either an evaluable relation, i.e., an atom of the form $X = Y$, $A\ \theta\ k$ or $A\ \theta\ B$, where X, Y are identifiers, A, B, are attributes, and k is a constant, or it can be a database relation, i.e., a literal of the form $p(...)$ or $\neg p(...)$, where p refers to a relation P. The algorithm below provides a mapping from these DATALOG query modifications into the corresponding query modifications in the original OQL query.

Let c, d be relations representing classes or structures C, D, and let r be a relation representing the relationship R.

ALGORITHM **DATALOG_to_OQL**

1. Adding (removing) an evaluable atom.

   Let $X = Y$ be an atom added to (removed from) the DATALOG query.

   - Add (remove) $x = y$ to (from) the **where** clause.

   Let $A\ \theta\ k$ be an atom added to (removed from) the DATALOG query. Let $c(X, ..., A, ...)$ be an atom in the DATALOG query.

   - Add (remove) $X.A\ \theta\ k$ to (from) the **where** clause.

   Let $A\ \theta\ B$ be an atom added to (removed from) the DATALOG query. Let $c(X, ..., A, ...)$, $d(Y, ..., B, ...)$ be atoms in the DATALOG query.

   - Add (remove) $X.A\ \theta\ Y.B$ to (from) the **where** clause.

2. Adding (removing) a predicate literal.

   Let $c(X, ...)$ be an atom added to (removed from) the DATALOG query.

   - Add (remove) *X in C* to (from) the **from** clause.

   Let $r(X, Y)$ be an atom added to (removed from) the DATALOG query.

   - Add (remove) *Y in X.R* to (from) the **from** clause.

   Let $\neg c(X, ...)$ be an atom added to (removed from) the DATALOG query.

   - Add (remove) *X not in C* to (from) the **from** clause.

Let $\neg r(X, Y)$ be an atom added to (removed from) the DATALOG query.

- Add (remove) *Y not in X.R* to (from) the **from** clause.

5. Applications

We present several examples of semantic optimization. In each case, we present an OQL query, its DATALOG representation Q, one or more integrity constraints that can be applied to the query, an optimized DATALOG query[2] Q', and an optimized OQL query. We do not include a full list of attributes for each predicate in a query or an integrity constraint, and we only include those attributes which appear in a query. All queries refer to the schema of Figure 1.

Using these examples, we illustrate the following: Integrity constraints can be used to identify contradictions and hence eliminate queries that need not be evaluated. Semantic information on the type hierarchy can be used to reduce the scope of a query to a subclass; this is a useful optimization technique in object databases that maintain the extents of classes. Key constraints identify identical objects and can be used to eliminate unnecessary join operations and object accesses from the object base. Integrity constraints about methods can be used to optimize queries in which these methods are applied. The knowledge of access support relations [9], can be used to eliminate joins, or to produce alternate equivalent queries. SQO obtains queries that could not be produced without the application of semantic knowledge. These queries may have more efficient evaluation plans and this is determined by a cost-based optimizer.

1. Contradiction detection

In this example we show how an integrity constraint describing an aspect of the implementation of a method can be used to optimize queries applying that method. In this particular case, SQO introduces a contradiction to the query in a similar way as shown in Example 1 of Section 2.

Assume that we have the following integrity constraints. The first one states that the salary of faculty members exceeds 40K.

IC1: Salary>40K← faculty(OID,Salary)

The second integrity constraint describes an aspect of the implementation of the method *taxes_withheld*, for e.g., the method is monotonic with respect to the *Salary* of objects on which it is computed.

IC2:
Value1≥ Value2←taxes_withheld(OID1,Rate,Value1),
faculty(OID1,Salary1),
taxes_withheld(OID2,Rate,Value2),
faculty(OID2,Salary2),
Salary1 ≥ Salary2.

Suppose, we also have the following fact that for *Salary=30K*, the method application with *Rate=10%* evaluates to *Value=3000*:

Value = 3000 ← employee(OID3,30K),
taxes_withheld(OID3,10%,Value)

We are now able to obtain the following integrity constraint[3] which states that with *Rate=10%*, all faculty members pay more than 3000 in taxes:

IC3:
Value> 3000 ←taxes_withheld(OID,10%,Value),
faculty(OID)

Now consider the query of Example 2 of Section 4.3. Optimizing the DATALOG representation of this query with IC3 yields the following query:

$Q'(Name_1, City)$ ← student(X, $Name_2$), takes(X,Y),
taught_by(Y,Z), faculty(Z,$Name_1$,W),
address(W,City), $Name_2$="john",
taxes_withheld(Z,10%,V), V<1000,
V>3000

This query contains a contradiction so it need not be evaluated.

2. Access scope reduction

Many object databases maintain the extent for a class, i.e., the OIDs of all objects in that class. By using integrity constraints on class hierarchies, it may be possible to reduce the scope of a query to a subclass. Manipulation of the extents of the classes and subclasses may lead to plans that possibly reduce the number of objects that are accessed from the object database. Consider a query that asks for names of all persons younger than 30. The OQL query and its DATALOG representation are as follows:

select x.name
from x in Person
where x.age<30

[2]The reader is referred to [3] for details on the optimization of DATALOG queries.

[3]We refer the reader to [7] for details of this simplification.

Q(Name)← person(X,Name,Age),Age< 30.

Suppose we know that all faculty members are more than 30 (IC4) and the set of all faculty members is a subclass of the set of all persons (IC5). Then, the objects of the class *Faculty* should not be considered when evaluating the query.[4]

IC4:
Age≥ 30 ← faculty(X,Name,Age)

IC5:
person(X,Name,Age)← faculty(X,Name,Age)

Given IC4 and IC5, we can derive the following integrity constraint:

IC6:
$Age \geq 30$ ← faculty(X,Name,Age),person(X,Name,Age)

which can also be expressed as:

IC6′:
¬ faculty(X,Name,Age)← person(X,Name,Age),Age<30

Optimization of Q with $IC6'$ yields the following:

Q′(Name)← person(X,Name,Age), Age<30,
¬faculty(X,Name,Age).

Since a new literal has been added to the DATALOG representation of the query, the algorithm **DATALOG_to_OQL** yields the following optimized OQL query.

select x.name
from x in Person
x not in Faculty
where x.age<30

One possible way to evaluate this query is to use the class extents for *Faculty* and *Person*, to first identify those objects that are in class *Person* but not in class *Faculty*. and then retrieve only those object instances. Such an optimization would be possible in object databases that maintain the extents of classes.

3. Join reduction using key constraints

The query that is considered here uses a constructor **list** to create lists containing the student id of a student, and the employee id of a TA, such that they both take a section taught by the professor of the same name. This example will illustrate that our SQO technique is able to preserve the constructors in the optimized OQL query. Further, in the original query, objects identified by identifiers z and w, are involved in a join operation over the attribute *name*. This would require the retrieval of these objects from the database. Using a key constraint on attribute *name*, we can replace this join with a corresponding comparison of the OIDs. We are thus able to introduce an optimization that eliminates unnecessary object retrievals.

select **list**[s.student_id, t.id]
from s in Student
t in TA
y in s.Takes
z in y.Is_taught_by
v in t.Takes
w in v.Is_taught_by
where z.name=w.name

The DATALOG representation of the query, which does not preserve the constructor **list**, is as follows:

Q(Student_id,Id)←student(S,Student_id), takes(S,Y),
is_taught_by(Y,Z),faculty(Z,$Name_1$),
ta(T,Id),takes(T,V),is_taught_by(V,W),
ffaculty(W,$Name_2$),
$Name_1$=$Name_2$.

IC7 states that *Name* is a key for the relation *Faculty*.

IC7:
X_1=X_2 ←faculty(X_1,$Name_1$),faculty(X_2,$Name_2$),
$Name_1$=$Name_2$.

IC8 states that two objects with the same OIDs are identical.

IC8:
$Name_1$=$Name_2$ ←faculty(X_1,$Name_1$),faculty(X_2,$Name_2$),
X_1=X_2.

These two integrity constraints allow us to rewrite the query so that no objects from the class *Faculty* need be retrieved to compare their names as follows:

Q′(Student_id,Id)← student(S,Student_id), takes(S,Y),
is_taught_by(Y,Z), faculty(Z,$Name_1$),
ta(T,Id),takes(T,V), is_taught_by(V,W),
faculty(W,$Name_2$), Z=W.

Two changes have been made to the DATALOG representation of the query as follows: the atom $Name_1$=$Name_2$ has been removed and the atom z=w has been added to the query. These changes are mapped in a straightforward way to the OQL query to yield the following:

[4]For simplicity, we use the comparison $Age < 30$ in both the integrity constraint and the query. In general, SQO must handle a more general case, e.g., with $Age < 20$ in the query. We do not describe the details of the simplification here.

```
select list[s.student_id, t.id]
from   s in Student
       t in TA
       y in s.Takes
       z in y.Is_taught_by
       v in t.Takes
       w in v.Is_taught_by
where z=w
```

This optimized query may have an evaluation plan that compares OIDs z and w, where z is in the set of OIDs *y.Is_taught_by*, and w is in the set of OIDs *v.Is_taught_by*. The original query would have had a plan that retrieved objects z and w from *Faculty*. The (new) plan thus provides an optimization opportunity to reduce object accesses from the database. Note also that despite the fact that the DATALOG representation of the OQL query did not contain the constructor **list**, it is retained in the OQL query.

4. Join introduction (using access support relations)

Queries that require evaluating very long path expressions may be expensive to process. To optimize their evaluation, *access support relations* were introduced in [9]. Access support relations are separate structures that explicitly store OIDs that relate objects with each other. They may be maintained for path expressions that are accessed frequently in queries. SQO can use access support relations to reduce the number of joins in a query. SQO may also use them to obtain alternate queries which may have more optimal evaluation plans.

Consider the following path expression $\mathcal{P}$:

takes(X,Y),is_section_of(Y,Z),has_sections(Z,V),has_ta(V,W)

Assume that this path expression occurs often in queries relating the first and the last object of the path, as in the following query:

```
Q(W): ←student(X,Name),takes(X,Y),
        is_section_of(Y,Z), has_sections(Z,V),
        has_ta(V,W), Name="james"
```

An access support relation for $\mathcal{P}$ is a materialized view *asr* defined as follows:[5]

```
asr(X,W)← takes(X,Y),is_section_of(Y,Z),
          has_sections(Z,V),has_ta(V,W)
```

Given this access support relation, all queries containing $\mathcal{P}$ can be evaluated more efficiently. For example, query Q(W) can be rewritten as follows, where the view *asr* has been introduced to eliminate several joins:

Q'(W) ← student(X,Name),asr(X,W), Name="james"

Now consider the following query:

```
Q1(V) ← student(X,Name),takes(X,Y),
        is_section_of(Y,Z), has_sections(Z,V),
        Name="johnson"
```

The access support relation *asr* is not directly useful here since it only relates objects from the class identified by X, i.e., Student, with objects from the class identified by W, i.e., TA. However, suppose there is an integrity constraint that for every student registered for a course, there is a teaching assistant assigned to each section of that course, as follows:

IC9:
```
has_ta(V,W)← takes(X,Y),is_section_of(Y,Z),
             has_sections(Z,V)
```

Suppose there is also another integrity constraint that *has_ta(V,W)* is a one-to-one relationship, i.e., each section has exactly one TA and each TA has exactly one section. Then, using this integrity constraint and IC9, we obtain the following:

```
Q'1(V) ← student(X,Name),takes(X,Y),
         is_section_of(Y, Z), has_sections(Z,V),
         has_ta(V,W), Name="johnson"
```

which in turn can be rewritten as follows:

```
Q'1(V) ← student(X,Name), asr(X,W), has_ta(V,W),
         Name="johnson"
```

In the first example, using the access support relation *asr* reduced the number of joins, compared to the original query. For queries involving long path expressions, i.e., queries that require evaluating many joins, such savings could be substantial. In the second query, the use of the access support relation produced an alternate query evaluation plan which would not have been produced by a syntactic optimizer, which does not use semantic knowledge. A physical cost optimizer can make the decision as to whether this evaluation will be more efficient, compared to the query that did not use the access support relation. Thus, there is a possibility of reducing the number of joins, and the generation of alternate evaluation plans, using SQO.

[5] This type of relation is called a canonical extension of access support relation for $\mathcal{P}$ in [9].

6. Conclusions and Future Work

We presented a logic-based approach to SQO in object databases. Our optimization technique can be applied to a large class of OQL queries including queries with method application and structure constructors. We plan to extend this work in two ways. First, we intend to consider larger classes of OQL queries, e.g., existentially quantified queries. Second, we will identify the type of information, about the implementation of a method, that can be expressed in terms of integrity constraints, and that can be used for SQO. This will require us, in particular, to address the issue of method overloading. Finally, we plan to incorporate our techniques for SQO into the rule-based query optimizer of [6], which is based on an extended pattern-match algorithm for an object algebra. This would allow us to perform an experimental evaluation of the benefits of SQO.

References

[1] K. Aberer and G. Fischer. Semantic query optimization for methods in object-oriented database systems. In *Proc. IEEE International Conference Data Engineering*, pages 70–79, 1995.

[2] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. In *Proceedings of the 4th International Conference on Data Engineering*, 1988.

[3] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.

[4] R. G. G. C. (ed.). *Object Database Standard: ODMG-93*. Morgan Kaufman Publishers, 1993. Release 1.1.

[5] R. G. G. C. (ed.). *Object Database Standard: ODMG-93*. Morgan Kaufman Publishers, 1996. Release 1.2.

[6] D. Florescu. *Design and Implementation of the Flora Object Oriented Query Optimizer*. PhD thesis, University of Paris 6, Department of Computer Science, 1996.

[7] J. Grant, J. Gryz, J. Minker, and L. Raschid. Logic-based approach to semantic query optimization in object-oriented databases. Technical Report CS-TR-3623, UMIACS-TR-96-25, Dept. of Computer Science, University of Maryland, College Park, MD 20742, Apr. 1996.

[8] M. Hammer and S. Zdonik. Knowledge-based query processing. *Proc. 6th International Conference on Very Large Data Bases*, pages 137–147, Oct. 1980.

[9] A. Kemper and G. Moerkotte. Access support in object bases. In *SIGMOD Proceedings*, pages 364–374. ACM, 1990.

[10] J. King. Quist: A system for semantic query optimization in relational databases. *Proc. 7th International Conference on Very Large Data Bases*, pages 510–517, Sept. 1981.

[11] L. V. S. Lakshmanan and R. Missaoui. On semantic query optimization in deductive databases. In *Proc. IEEE International Conference on Data Engineering*, pages 368–375, 1992.

[12] Y.-W. Lee and S. Yoo. Semantic query optimization for object queries. In *Proceedings of DOOD*, pages 467–484, Singapore, 1995.

[13] A. Levy and Y. Sagiv. Semantic query optimization in datalog programs. In *Proc. PODS*, 1995.

[14] H. Pang, H. Lu, and B. Ooi. An efficient semantic query optimization algorithm. In *Proc. IEEE International Conference on Data Engineering*, pages 326–335. IEEE Computer Society Press, 1991.

[15] S. Shenoy and Z. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, Sept. 1989.

[16] J. Yoon and L.Kerschberg. Semantic query optimization in deductive object-oriented databases. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases*, pages 169–182, 1993.

[17] S. Yoon, I. Song, and E. Park. Semantic query processing in object-oriented databases using deductive approach. In *Proceedings of CIKM*, pages 150–157, Baltimore, 1995.