Logic-Based Query Optimization for Object Databases

John Grant, Jarek Gryz, Jack Minker, *Fellow*, *IEEE*, and Louiqa Raschid, *Member*, *IEEE*

Abstract—We present a technique for transferring query optimization techniques, developed for relational databases, into object databases. We demonstrate this technique for ODMG database schemas defined in ODL and object queries expressed in OQL. The object schema is represented using a logical representation (Datalog). Semantic knowledge about the object data model, e.g., class hierarchy information, relationship between objects, etc., as well as semantic knowledge about a particular schema and application domain are expressed as integrity constraints. An OQL object query is represented as a logic query and query optimization is performed in the Datalog representation. We obtain equivalent (optimized) logic queries and, subsequently, obtain equivalent (optimized) OQL queries for each equivalent logic query. In this paper, we present one optimization technique for *semantic query optimization* (SQO) based on the residue technique of [6], [7], [8]. We show that our technique generalizes previous research on SQO for object databases. We handle a large class of OQL queries, including queries with constructors and methods. We demonstrate how SQO can be used to eliminate queries which contain contradictions and simplify queries, e.g., by eliminating joins, or by reducing the access scope for evaluating a query to some specific subclass(es). We also demonstrate how the definition of a method, or integrity constraints describing the method, can be used in optimizing a query with a method.

Index Terms—Datalog, integrity constraints, logic query, methods in object DBMS, object DBMS, ODL, ODMG, OQL, semantic query optimization (SQO), semantic residues.

1 Introduction

In this paper, we demonstrate the application of *semantic query optimization* (SQO) techniques to query processing in object databases. SQO uses semantic knowledge, in the form of integrity constraints (ICs), to reformulate a query and obtain a semantically equivalent query, which can possibly be evaluated more efficiently. Informally, two queries are semantically equivalent if they obtain the same answers in a database that satisfies the same set of integrity constraints. SQO has been applied to relational and deductive databases [6], [7], [8]. 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.

In a relational database, semantic information about the particular application domain has to be explicitly obtained by a database administrator and there is usually a limited amount of such information available. In contrast, an object database typically includes a much larger variety of semantic information and, thus, is perhaps a more suitable candidate to benefit from exploiting semantic knowledge to obtain equivalent, and possibly optimized, queries. For example, semantic knowledge about the object model may include knowledge about the class hierarchy, object identity, relationships between objects, etc. In addition, we may obtain semantic knowledge that is particular to the application domain, just as we would for a relational schema. Finally, we may obtain semantic knowledge about a method implementation. As discussed in Section 5, semantic knowledge on a method may either be the method definition or integrity constraints that describe the behavior of the method. All of this semantic knowledge may be utilized by SQO to obtain semantically equivalent and possibly more efficient queries for object databases.

We present a technique for SQO and we demonstrate this technique for ODMG database schemas defined in ODL, and object queries expressed in OQL. The object schema is represented using a logical representation (Datalog). Semantic knowledge about the object data model, e.g., class hierarchy information, relationship between objects, etc., as well as semantic knowledge about the particular schema and application domain, are expressed as integrity constraints. An OQL object query is represented as a logic query (Datalog) and query optimization is performed in the Datalog representation. We obtain equivalent (optimized) logic queries and, for each equivalent logic query, we subsequently obtain equivalent (optimized) OQL queries. As we show in Section 2, we improve substantially on previous logic-based approaches to SQO in object databases.

We only consider unnested **select-from-where** queries. However, we are able to consider a fairly large class of queries since techniques for unnesting certain subqueries, when they occur in the **select** and **from** clauses, have been proposed in [15] and can be applied in our approach to obtain unnested queries. Some subqueries appearing in the

- *J. Grant is with the Department of Mathematics and the Department of Computer and Information Sciences, Towson University, 8000 York Rd., Towson, MD 21252. E-mail: jgrant@towson.edu.*
- *J. Gryz is with the Department of Computer Science, York University, 4700 Keele St., Toronto, Ontario, Canada. E-mail: jarek@cs.yorku.ca.*
- *J. Minker is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: minker@cs.umd.edu.*
- *L. Raschid is with the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. E-mail: louiqa@umiacs.umd.edu.*

Manuscript received 2 July 1997; revised 1 June 1999; accepted 13 July 1999. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 105325.

where clause also could be unnested in the same way as is done in SQL. Object queries include constructors, e.g., *struct, set, list*, etc., and also perform aggregate functions on collections of objects. We do *not* translate these constructors in the object query into the logic query and, hence, we do not perform SQO over collections of objects in the logical representation. However, in some restricted cases, we can optimize some object queries with these constructors. To explain, after we obtain an equivalent logic query, in the Datalog representation, we consider *both* the initial object query and the equivalent logic query in order to produce the equivalent object query. Thus, in some cases, we are able to preserve the constructors of the original object query and express them in the equivalent object query, even if these constructors do not appear in the initial logic query or the equivalent logic query. We will illustrate this with an example in Section 4.

Using an example ODMG database schema defined in ODL, and object queries in OQL, we demonstrate a number of possible optimization strategies using SQO and particular integrity constraints. For example, we demonstrate how SQO can identify contradictions and, thus, eliminate queries containing contradictions. We also demonstrate how SQO can be used to possibly simplify queries. For example, the access scope for evaluating a query can be reduced from a class to some particular subclass(es) depending on particular integrity constraints; this can reduce the number of objects that are retrieved from the database and is a possible optimization strategy. Further, queries can also be simplified by eliminating joins. One strategy to eliminate joins is using integrity constraints in the form of key constraints. Another strategy is to use *access support relations* [25] or special relations that store OIDs of objects that represent a materialized view corresponding to some path expression. Semantic knowledge about these special relations can be used to eliminate some joins and has the potential to simplify the query.

Application domain knowledge about methods can also be represented as integrity constraints in our approach and SQO can be used to optimize queries. In particular, we consider methods defined for a single object; this is because we do not represent collections of objects in the logic representation. SQO can be applied to methods that are predefined queries or arbitrary procedures. Simplification of the query, by SQO, may involve limiting the method application to objects in particular subclass(es) or adding new conditions to the query, which serves to reduce the number of objects on which the method is applied. A query with a method is optimized at a shallow level if the only available semantic knowledge is specific facts about the method. Optimization at a deep level requires that the encapsulation of the method must be broken. Semantic knowledge may either be the method definition itself, in the case where the method is defined as a query or, in the case where the method is an arbitrary procedure, semantic knowledge about the behavior of the method may be expressed as a function expressing some relationship between the input and output values. In this case, simplification by SQO may involve computing the function that expresses the behavior of the method, rather than applying the method to the object.

SQO makes extensive use of integrity constraints. Hence, for object databases with a variety of semantic knowledge, the space of semantically equivalent queries that must be considered by the cost-based physical optimizer could be fairly large. In this scenario, heuristics must be used to guide the optimizer. Many of the heuristics developed for SQO in relational databases [6], [37], e.g., those that restrict the introduction of selections on attributes that are not indexed, etc., also apply to object databases. However, as discussed earlier, there are several other types of simplifications for object queries that do not occur in SQO for relational databases. In more complex situations involving methods, SQO may produce a query in which a function that describes the behavior of a method, with respect to some object, is computed. In this case, heuristics must be used to determine if the cost of computing the function is cheaper than applying the method to this object. In many cases, the heuristics may be dependent on the particular object database implementation. We realize that this is an important issue and is critical to the success of SQO in object databases. While we identify where heuristics must be used to guide SQO, in this paper, we do not develop such heuristics. This research on developing and using heuristics is beyond the scope of this paper.

The paper is organized as follows: The next section provides background on research in SQO. Section 3 reviews the residue technique of [6], [7], [8] which is used in our optimization method. In Section 4, we first present the ODMG data model and present an overview of our method for SQO. We provide algorithms for translating an ODL schema into a relational representation and discuss representing semantic knowledge from the object model and for the object schema as integrity constraints. We then consider unnested **select-from-where** OQL queries and provide an algorithm for obtaining a corresponding logic query. We also present an algorithm for obtaining an object query from a logic query. We then use several examples to illustrate various optimization strategies for SQO in object databases. We conclude this section by discussing the treatment of existentially and universally quantified object queries. In Section 5, we first review methods in object databases and, then, we provide several strategies for optimizing queries with methods. We briefly consider method refinement. We identify future research in which other query optimization techniques can also be incorporated into object databases, using our Datalog representation for schemas and queries, and conclude in Section 6.

2 Background

Much work has been done to develop logic-based techniques for semantic query optimization (SQO) in deductive and relational databases [6], [23], [26], [34], [37]. Subsequently, this work has been extended to databases with recursion [27], [28], [32] and negation [19]. More recent work considers SQO in the context of object databases (ODB) and two different approaches have been proposed.

The first approach is based on manipulating special data structures designed specifically for SQO in ODB. Queries or

parts of queries are represented by these data structures. Heuristics are developed to produce query transformation rules tailored to these data structures and these transformations are applied to the query. Since the number of heuristics is limited, the number of query transformations that are applied is also limited and predictable; this approach is thus computationally simple. The disadvantage of this approach is that the heuristics and transformations only work for a specific query language and specific types of optimization or transformation. In some cases, they only work for certain types of ICs. This makes this approach implementation dependent and rigid.

The following two papers represent this approach: Pang et al. [35] present an algorithm for three types of semantic optimization for simple object queries. The algorithm uses a data structure, called a transformation table, which stores information about the predicates used in the query, classifying them as imperative, optional, or redundant. This information is utilized to eliminate some of the possible query transformations as not profitable. Lee and Yoo [29] identify several new types of semantic query transformations, specific to the object data model. A query is represented as a quadruple of graphs from which nodes are removed or added, depending on the type of transformation.

The second approach for SQO is logic-based. Queries, rules, and ICs have a uniform representation in a logical formalism and deduction is used to produce equivalent query forms. There are no restrictions placed on the transformations. All possible semantically equivalent queries will be obtained for that particular formalism.

Three different kinds of logical formalisms have been applied for SQO. Yoon and Kerschberg [40] assume that the database schema, rules, and queries are expressed in F-logic. They have shown how a query can be transformed into a semantically equivalent query which incorporates information contained in the integrity constraints. The transformation is done by means of resolution (on the query and ICs) extended to account for inheritance information. The authors also discuss the issue of conflict resolution strategies and the evaluation precedence of reformulated queries. The second formalism [5] uses description logics. Queries are expressed as OCDL (Object Constraint Description Language) types which can be expanded by SQO. The third formalism is based on Datalog. Yoon et al. [41] transforms an object schema into a deductive database schema and uses the residue technique of [6] to optimize a query. The optimization algorithm is only sketched there and applies to simple object queries.

Our approach to SQO is also logic-based and, thus, does not have the inherent limitations of the first approach. We consider a much larger class of queries compared to any of the previous approaches and, in particular, we are able to handle queries with constructors. We make extensive use of semantic knowledge and we consider a larger variety of simplifications in comparison to previous logic-based approaches [41]. Finally, our research is the first to systematically address the issue of using semantic knowledge to optimize methods.

3 Semantic Query Optimization for Relational Databases

This section contains background on semantic query optimization for relational databases, as described in [6]. The material is simply sketched here; the reader is referred to the original paper for details. Additional issues, such as nonrange restricted queries, the handling of negation, are also given in that paper. The formalism of [6] carries over to the context of OODBs. The basic idea is the use of a theorem proving technique, called 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, the residues are used, where possible, to transform the query to another query that is semantically equivalent to the original query but whose execution is potentially faster. For the sake of simplicity, the method is illustrated using a nonsorted logic.

We first present background on first-order logic and the notation used throughout the paper. A *literal* is either an atomic formula or the negation of an atomic formula. All facts, integrity constraints, and queries are represented by *clauses*. A clause is a disjunction of literals: $S_1 \vee \ldots \vee S_m \vee \neg R_1 \vee \ldots \neg R_n$. We use a Prolog-like notation and write this clause as $S_1 \vee \ldots \vee S_m \leftarrow R_1, \ldots, R_n$ and call $S_1 \vee \ldots \vee S_m$ the *head* and $R_1, \ldots, R_n$ the *body* of the clause. In this paper, we discuss only *Horn* clauses, i.e., clauses for which $m \leq 1$. The *null* clause has both a null body and a null head; it represents a contradiction.

Atomic formulae stand for evaluable (or built-in) relations such as: $>$, $<$, $\leq$, $\geq$, $=$, $\neq$, or relations defined as part of the database. Integrity constraints, IC, are rules expressing restrictions that the database must satisfy. We use the following conventions: Capitalized attribute names serve as variables in clauses; we write only those attributes that are of interest in a query or an integrity constraint. Constants and relation names start with lowercase letters. Queries are written as: $Q(Projected\_Attributes) \leftarrow Body$, where $Projected\_Attributes$ is the list of projected attributes.

We start the presentation of SQO for relational databases with the concept of subsumption:

Defintion 1. *A clause C_1* subsumes *a clause C_2 if there is a substitution σ such that $C_1\sigma$ is a subclause of C_2.*

For example, if

$$C_1 = p(X, Y, a) \leftarrow q(X, Z), r(X, Y, Z, a)$$

and

$$C_2 = p(b, Y, a) \leftarrow q(b, Y), r(b, Y, Y, a), s(a),$$

then C_1 subsumes C_2 by the substitution $\{X = b, Z = Y\}$.

To understand partial subsumption, we illustrate the basic algorithm to test for subsumption between C_1 and C_2. First, the proposed subsumed clause, C_2 in this case, is instantiated to a ground clause by a substitution θ using new constants $k_1, \ldots, k_n$ not present in C_1 or C_2. Thus,

$$C_2\theta = p(b, k_1, a) \leftarrow q(b, k_1), r(b, k_1, k_1, a), s(a),$$

where $\theta = \{Y = k_1\}$. Then, $C_2\theta$ is negated:

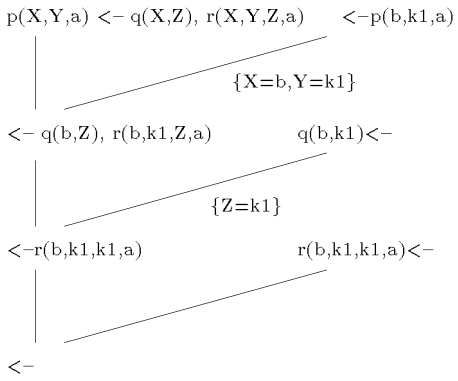


Fig. 1. Refutation tree for subsumption.

$$\neg C_2\theta = \{\leftarrow p(b, k_1, a); q(b, k_1) \leftarrow; r(b, k_1, k_1, a) \leftarrow; s(a) \leftarrow\}.$$

The final step is the construction, if possible, of a linear refutation tree with C_1, the proposed subsuming clause, as the root, and using at each step an element of $\neg C_2\theta$ in the resolution. The result is that C_1 subsumes C_2 if and only if there is a refutation tree that ends with the null clause. In our example, such a tree exists, as seen in Fig. 1.

The subsumption algorithm is applied in partial subsumption with an integrity constraint taking the part of the subsuming clause (C_1) and a relation (negated) taking the part of the subsumed clause (C_2). In general, the null clause is not obtained because integrity constraints usually do not subsume relations. However, an integrity constraint may partially subsume a relation, leaving a fragment at the bottom of a refutation. Such a fragment is called a *residue*. A residue, if nontrivial, represents an interaction between a relation and an integrity constraint.

Before starting subsumption, constants must be taken care of in the body of an integrity constraint, as the following example illustrates. Suppose that $r(X, Y, Z)$ is a relation and $X = a \leftarrow r(X, b, Z)$ is an integrity constraint. The subsumption algorithm does not work here because $X = a \leftarrow r(X, b, Z)$ cannot be resolved with $r(k_1, k_2, k_3) \leftarrow$, the negation of the instantiation of $r(X, Y, Z)$. A technique, called *expansion*, is applied to the integrity constraint to eliminate undesirable constants such as the "b" above. In this case, expanding $X = a \leftarrow r(X, b, Z)$ yields $X = a \leftarrow r(X, X_1, Z), X_1 = b$, which can be resolved with $r(k_1, k_2, k_3) \leftarrow$ to yield $k_1 = a \leftarrow k_2 = b$. This is not yet the residue. The substitution θ must first be reversed by applying θ^{-1}. The residue is $X = a \leftarrow Y = b$, intuitively representing the integrity constraint's effect on r. Actually, in some cases, an extra step, called *reduction*, is necessary for obtaining a useful residue. This step basically takes care of removing redundant atoms and reversing the result of expansion. We leave out the details of expansion and reduction here.

Next, we state the definitions of partial subsumption and residue.

Definition 2. *We say that IC* partially subsumes *the relation R if IC does not subsume $\leftarrow R$, but a subclause of IC^+ (where IC^+ is a result of expanding IC) subsumes $\leftarrow R$.*

Definition 3. *Given an integrity constraint IC and relation R, apply the subsumption algorithm to IC^+ and $\leftarrow R$ until no more resolutions are possible. Let C be the clause at the bottom of a refutation tree. Then, $(C^-)\theta^{-1}$ (where C^- is a result of reducing C) is a* residue *of IC and R.*

There may be several different residues between an integrity constraint and a relation. If the residue is IC^-, that is, there was no resolution, we call it a *maximal* residue. We call a residue that is always true, like $a = a \leftarrow$, a *redundant* residue. Such residues are not useful. Finally, we call an integrity constraint *merge-compatible* with a relation if there is at least one residue that is neither maximal nor redundant. These are the residues used in semantic query optimization.

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

We show informally on the following example how residues are used for SQO.

Example 1. Let the database contain the following relations: *Student(St_id, Name)*, *Takes_section(St_id, Section#)*, *Faculty(Section#, Fac_id, Age)*. Assume also that there is an integrity constraint, $\mathcal{IC}_1$, which says that all faculty members are more than 18 years old. Formally:

$$\text{Age} \geq 18 \leftarrow \text{ faculty(Section\#,Fac\_id, Age).} \qquad \mathcal{IC}_1$$

By the method of partial subsumption, as described above with $\mathcal{IC}_1$, and the relation *Faculty*, we can construct the residue: $\{Age \geq 18 \leftarrow\}$ (no expansion or reduction was necessary here), which is attached to the relation *Faculty*. Intuitively, it means that any query with the predicate *faculty* must satisfy the condition stated by the residue.

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

$$\begin{aligned} \text{Q(Name)} \leftarrow\ & \text{student(St\_id, Name),} \\ & \text{takes\_section(St\_id, Section\#),} \\ & \text{faculty(Section\#, Fac\_id, Age), Age} < 18. \end{aligned}$$

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

$$\begin{aligned} \text{Q'(Name)} \leftarrow\ & \text{student(St\_id, Name),} \\ & \text{takes\_section(St\_id, Section\#),} \\ & \text{faculty(Section\#, Fac\_id, Age),} \\ & \text{Age} < 18, \text{Age} \geq 18. \end{aligned}$$

The query contains a contradiction. This means that it cannot return any answers, otherwise the database would violate $\mathcal{IC}_1$. Hence, the query need not be evaluated. An optimizer, upon discovering the contradiction, should halt the processing of the query.

4 Semantic Query Transformation

4.1 ODMG Data Model

The lack of standards for object databases has been a major limitation to their more widespread use and also to the ability of transferring tools and techniques developed for relational databases to object databases. Reference [13] describes a standard for object database management

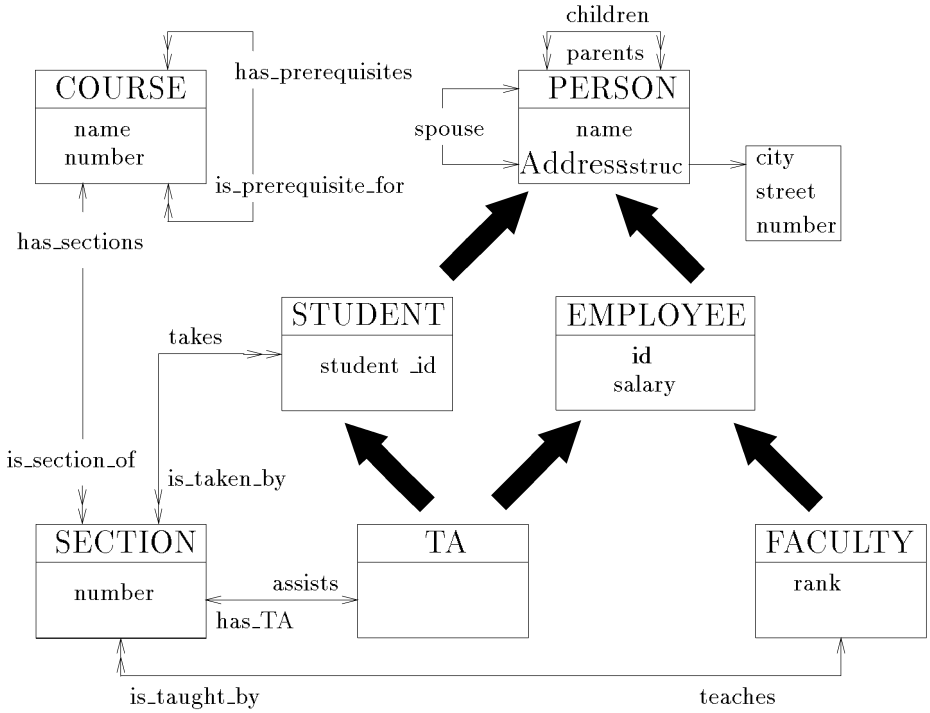


Fig. 2. Example database schema. Each box represents a class, heavy arrows indicate class hierarchy, thin arrows describe relationships between classes. The data in a box are attributes for that class; additional attributes are inherited from superclasses. Methods are not included in the figure.

systems developed by the members of the Object Database Management Group (ODMG). This standard provides, among other things, a syntax for an Object Definition Language (ODL) and an Object Query Language (OQL).

The ODMG 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.
- 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 behavior of objects of a type is defined by a set of *methods* (also called *operations*) that can be executed on objects of the type.

The schema in Fig. 2 (which is a slightly modified example from [13]) illustrates the major features of the ODMG data model in the form of a diagram.

The above specification describes the most typical features of object databases. Perhaps the only point that needs to be clarified is the difference between attributes and relationships. Only the latter, according to the standard, can relate objects. Attributes do not have OIDs; their individuality is determined by the individual object to which they apply. Hence, a structure which is an attribute of some object cannot be shared, i.e., it cannot be an attribute of another object or contain substructures that change dynamically. In our logic-based representation, for uniformity, we introduce an OID for structures. This does not create a problem in the optimized object queries since OIDs for structures are never introduced into the object queries. We note that techniques for representing structure attributes without OIDs as specified by ODMG, in the relational framework, have been developed in [21].

4.2 Overview of our Method

We perform SQO for the first-order logic representation of a query expressed in OQL. Our optimization technique has several steps, shown in Fig. 3. During a preoptimization phase, the object database schema, described in ODL, is translated into an equivalent logical representation in Step 1. It is similar to translations proposed in [3], [41]. This translation is intended to capture all 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 key constraints. We assume that all integrity constraints are expressed in the logical formalism as well. In the optimization phase, each OQL query is then translated into its logical representation in Step 2. Next, we apply the residue method to obtain all queries semantically equivalent to the original logical query in Step 3. Finally, in Step 4, we obtain an optimized OQL query and, possibly, more than one.

We should emphasize here that the input to the transformation of Step 4 contains not only an optimized logic query, but also 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, loses all these

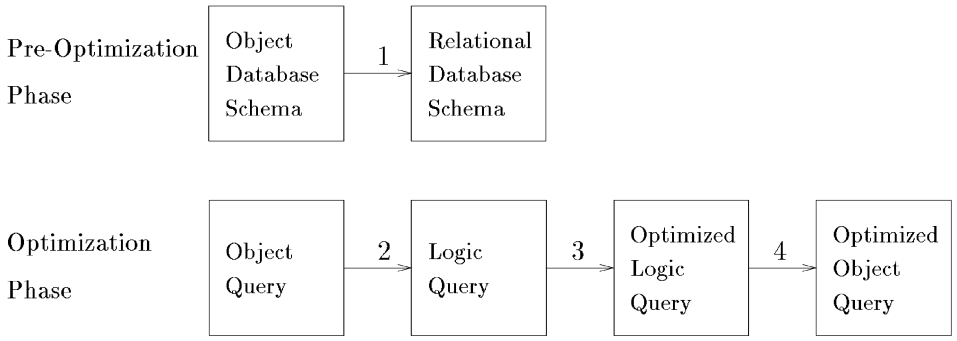


Fig. 3. Query optimization steps. The arrows are not intended to indicate the complete input to each step.

features, hence they cannot be retrieved into the optimized OQL query from the logic query alone. To guarantee full equivalence of OQL queries (the original and the modified), we 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.

The output of SQO in our approach contains *several* semantically equivalent queries. Each of these queries should then be evaluated by a conventional cost-based physical optimizer and the query with the lowest cost processed in the database. Since the number of semantically equivalent queries generated by SQO can be large and, in the worst case, it is exponential in the number of residues applicable to the input query, the algorithm should be accompanied by heuristics. Such heuristics would guide the query transformation process so that only promising transformations are generated. A strategy to eliminate some of the useless residues (such as redundant residues) has already been described in Section 3. Many of the heuristics developed for SQO in relational databases [6], [37], e.g., those that restrict the introduction of selections on attributes that are not indexed, etc., also apply to object databases. However, as mentioned in the introduction, there are several other types of simplifications for object queries that do not occur in SQO for relational databases. Some guidelines for the efficient search of promising semantic transformations in object databases have been suggested in [29], [35]. In many cases, the heuristics may be dependent on the particular object database implementation. We do not consider the development of heuristics in this paper.

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. 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 nontrivial for large databases, it would be amortized over a large number 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.

4.3 Schema Translation

We represent an ODL schema in the relational model 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. The definition of a method or facts about the behavior of a method, are also expressed as integrity constraints. The choice of the pure relational model versus nested relational or F-logic (which are closer in spirit to the object data model) was made for its simplicity and expressive power. It is easy to show that our translation of an ODL schema into a relational schema is complete in the sense that a reverse translation uniquely retrieves the original ODL schema.

We reiterate that the goal of our transformation is to express all semantic information encoded in an ODL schema using a Datalog representation. Our use of this transformation in no way implies that the object database is actually implemented using deductive database or relational database technology.

4.3.1 Relations

Each class, structure, relationship, and method defined in an ODL schema becomes a relation in the relational schema. The following rules specify the translation. In the Appendix, we give a partial translation for the schema of Fig. 2.

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

   simple attributes $A_1, \ldots, A_n$;
   structure attributes $S_1, \ldots, S_m$.

   Note: Subclasses inherit the attributes of their superclasses (in an unambiguous way). For each class C, create a $1 + n + m$-ary relation

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

   where OID_{S_i} is an OID of a structure of type S_i.
2. For each structure S with attributes as above, create a $1 + n + m$-ary relation

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

3. For each relationship R, between classes C_1 and C_2, create a 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[1] M, defined on objects of class C, with user provided arguments $A_1, \ldots, A_n$, that returns a value $Value$, create an $n+2$-ary relation m:

$$m(OID_C, A_1, \ldots, A_n, V),$$

where OID_C is an OID of an object of type C and $V = Value$, if $Value$ is a base value, or $V = OID_{Value}$, where OID_{Value} is an object identifier, otherwise.

4.3.2 Integrity Constraints

Integrity constraints created by the rules below are intended to capture semantic information that is built into the object schema. For object databases that allow the explicit representation of integrity constraints,[2] such constraints should also be translated into Datalog and added here.

We assume throughout the paper that all variables that appear in the head of a rule and do not appear in its body are existentially quantified.

1. OID Identification:
   - For each relation r obtained from a relationship between classes C_1 and C_2, we have

     $$c_1(OID_1, A_1, \ldots, A_n) \leftarrow r(OID_1, OID_2),$$
     $$c_2(OID_2, B_1, \ldots, B_m) \leftarrow r(OID_1, OID_2),$$

     where $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are attributes of c_1 and c_2, respectively.[3] We do not obtain such constraints for subclasses of C_1 and C_2.
   - For each class C containing the OID of a structure S as an attribute, we have $s(OID, B_1, ..., B_n) \leftarrow c(OID_1, A_1, ..., A_m, OID)$.
   - For each method M containing the OID of a class C, we have

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

     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, we have

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

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, we have

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

4. Many-to-one constraints:
   For each relation r obtained from a many-to-one relationship R, we have

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

5. Key and dependency constraints:
   - For each relation c obtained from a class or structure, we have

     $$A_1 = B_1 \& \ldots \& A_n = B_n \leftarrow c(OID, A_1, \ldots, A_n), c(OID, B_1, \ldots, B_n).$$

   - For each relation c obtained from a class or structure, if A_i is a key,

     $$OID_1 = OID_2 \leftarrow c(OID_1, A_1, \ldots A_i, \ldots, A_n), c(OID_2, B_1, \ldots, A_i, \ldots B_n).$$

   - Method constraints:
     For each method that is a predefined query, the definition of the method will be specified as a view definition. For each method that is an arbitrary procedure, facts describing the behavior of the method will be expressed as integrity constraints. Details of the view definition and method constraints are presented in Section 5.

4.4 Query Translation

Step 2 in Fig. 3 is the query translation phase. 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 latter retrieves from the relational database precisely the tuples representing (according to the schema transformation) the objects retrieved by the former. A formal proof of the correctness of our transformation is beyond the scope of this paper.[4] The material below considers conjunctive queries. It can be extended easily to handle disjunctive queries. A disjunctive query maps into a set of Datalog queries, each of which must be answered.

In this section, we consider OQL queries restricted in the following ways:

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

1. We refer to one type of method only, i.e., methods applicable to single objects returning a single value. A more comprehensive discussion of methods is presented in Section 5.

2. Reference [13] does not allow the explicit representation of integrity constraints.

3. The A_i and B_j are essentially Skolem functions.

4. The proof of the equivalence of Datalog queries, before and after optimization, is provided in [8]. Extending that proof to the equivalence between the original and modified OQL queries would require describing a single semantics over both Datalog and OQL.

- 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 is not translated into a new OQL query (in which case, all constructors from the original query would be missing). Rather, we map the modifications done in the Datalog query into modifications of the original OQL query and, thus, do not have to reconstruct the OQL query from scratch. In this way, we can apply the technique to a large class of OQL queries without changing the format of the answer set.

  For clarity of presentation, we also ignore set expressions *intersect, union, except*,[5] which can be represented in Datalog.
- The algorithm we present works for unnested queries only. Subqueries appearing in the **where** clause should be unnested in the same way it is done in SQL. Techniques for unnesting certain subqueries from the **select** and **from** clauses have been proposed in [15], [14] and will be used to simplify a nested query, prior to translation.
- Universally and existentially quantified queries can also be transformed into Datalog; this is discussed in Section 4.6.

Before the algorithm is applied to an OQL query, we simplify the query by removing from it all path expressions and substituting them with "one-dot" expressions, i.e., expressions of the form $x.y$, where neither x nor y are path expressions. This is done in the following way.

1. For all $P.x$, where P is a path expression:
   - Substitute $y.x$ for $P.x$, where y is an identifier that does not occur anywhere else in the query.
   - Add to the **from** clause the expression $P\ y$.
2. Repeat until there are no more path expressions in the query.

The following OQL query has path expressions replaced by one-dot expressions:

select $x.A$
from $P\ y$
where $z.B\ \theta\ c$,

where x, y, z are identifiers, P is a "one-dot" expression, A, B are attributes or methods, and c is a constant. There may be one or more expressions of the form $x.A$ in the **select** clause, one or more phrases of the form $P\ y$ in the **from** clause, and zero or more phrases of the form $z.B\ \theta\ c$ in the **where** clause. For readability, we do not cover the case where there is an expression $z.B\ \theta\ v.C$ in the **where** clause since it is a trivial extension of the case where $v.C$ is a constant.

Attributes with the same name but in different classes should be given different names in the translation. The procedure for identifying classes (structures) of identifiers follows the algorithm.

Identifiers play the role of OIDs in the Datalog representation of a query. The Datalog translation of the OQL query has the form: *Q(Projected_Attributes)*$\leftarrow$ *Body*. We assume that atoms are added to the *Body* of the Datalog query only if they have not been added there in previous steps of the algorithm (i.e., we guarantee nonredundancy of the Datalog query).

ALGORITHM **OQL_to_DATALOG**
1. For each $x.A$ in the **select** clause
   - 1.1. If A is an attribute,
     - 1.1.1. Add $c(X, \ldots, A, \ldots)$ to the *Body*, where c is the class (structure) for which x is the identifier.
     - 1.1.2. Add A to the *Projected_Attributes*.
   - 1.2. If A is a method $M(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are user provided arguments,
     - 1.2.1. Add $c(X, \ldots)$ to the *Body*, where c is the class (structure) for which x is the identifier.
     - 1.2.2. Add $m(X, A_1, \ldots, A_n, V)$ to the *Body*, where $m(X, A_1, \ldots, A_n, V)$ is the relational representation of the method M.
     - 1.2.3. Add V to the *Projected_Attributes*.
2. For each $z.B\ \theta\ c$ in the **where** clause,
   - 2.1. If B is an attribute,
     - 2.1.1. Repeat step 1.1.1. (with $x.A = z.B$).
     - 2.1.2. Add $B\ \theta\ c$ to the *Body*.
   - 2.2. If B is a method
     - 2.1.1. Repeat steps 1.2.1.-1.2.2. (with $x.A = z.B$).
     - 2.1.2. Add $V\ \theta\ c$ to the *Body*.
3. For each $P\ y$ in the **from** clause,
   - 3.1. If $P = C$, where C is the name of a class,
     - 3.1.1. Add $c(Y, \ldots)$ to the *Body*, where c is the relational representation of C.
   - 3.2. If $P = x.R$, where R is the name of a relationship,
     - 3.2.1. Add $r(X, Y)$ to the *Body*, where r is the relational representation of R.
   - 3.3. If $P = x.S$, where S is the name of a structure,
     - 3.3.1. Add $s(Y, \ldots)$ to the *Body*, where s is the relational representation of S.
     - 3.3.2. Put Y as a structure identifier of structure S in the class c to which X refers, i.e., $c(X, \ldots, Y, \ldots)$.

To identify classes (structures) to which identifiers in an OQL query refer, we need to find the appropriate statement in the **from** clause. Let x be an identifier whose class (structure) we want to identify. Then, there must be a statement $P\ x$ in the **from** clause. If P is the name of a class, then x refers to the class P. If $P = y.S$, where S is the structure name, then x refers to the structure S. Otherwise,

5. Except cannot generate unsafe formulas in Datalog if the output query from OQL is union compatible.

$P = y.R$, where R is an OODB relationship. Then, there is an integrity constraint linking the relationships and classes:

$$c(X, A_1, \ldots A_n) \leftarrow r(Y, X),$$

where r is the relational representation of R. In this case, x refers to the class C for which c is the relational representation.

We trace the **OQL_to_DATALOG** algorithm on the example based on the schema of Fig. 2.

Example 2. Let the following be a query in which path expressions have been replaced with one-dot expressions. The query refers to the schema of Fig. 2. *Taxes_withheld* is a method defined on the class *Employee*; it takes one user-provided argument *Rate* and returns a value of type number.

select z.name, w.city
from Student x
x.Takes y
y.Is_taught_by z
z.Address w
where x.name = "john"and $\mathrm{z.Taxes\_withheld}(10\%) = 1,000$.

We follow the above algorithm step by step as follows:

1. There are two expressions in the **select** clause: *z.name and w.city* and both contain attribute names. They are identified as belonging, respectively, to the class *Faculty* and the structure *Address*. Hence, the following two atoms are added to the body of the query:

   $$faculty(Z,\ Name_1), address(W,\ City)$$

   and $Name_1$ and $City$ are added to the projected attributes in the *Projected_Attributes*.
2. Of the two expressions in the **where** clause, *x.name* = "*john*" contains an attribute name and $z.taxes\_withheld(10\%) = 1,000$ contains a method name.

   For the first one, the class *Student* is identified and the following atoms are added to the body of the query:

   $$student(X,\ Name_2),\ Name_2 =''\ john.''$$

   The second expression contains an identifier z which refers to the class *Faculty* which was already added to the query in Step 1. Thus, only the following two atoms are added:

   $$taxes\_withheld(Z, 10\%, V), V = 1,000.$$

3. Of the four phrases in the **from** clause, *Student x* has already been translated and added in Step 2. The expression *x.takes y* is translated as:

   $$takes(X, Y).$$

   The expression *y.is_taught_by z* is translated as:

   $$is\_taught\_by(Y, Z).$$

   The expression *z.address w* is translated as $address(W, City)$. Since it has already been added in Step 1, it need not be added here. However, we need to substitute W as an attribute holding the OID of the structure *Address* in the class to which z refers, which is *Faculty*. Thus, the translation of *Faculty* becomes $faculty(Z, Name_1, W)$

Recall that we do not indicate all attributes of the OQL classes in Datalog atoms. The final query has the following form:

Q' ($\mathrm{Name_1}$, City) ← student(X, $\mathrm{Name_2}$), takes(X,Y),
is_taught_by(Y,Z),
faculty(Z,$\mathrm{Name_1}$,W),
address(W,City), $\mathrm{Name_2}$="john",
taxes_withheld(Z,10%,V),
$V = 1000$.

After the Datalog query has been optimized (as sketched in Section 3), we are able to transform the original OQL query into a semantically optimized form. As we stated before, the idea here is to map the changes introduced in the Datalog representation of the query to changes in 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 removed (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 a database relation (i.e., a literal of the form $p(\ldots)$ or $\neg p(\ldots)$, where p is a relation name). The algorithm below provides a straightforward mapping of these Datalog query modifications to OQL query modifications.

Let c, d be relations representing classes or structures C, D and r be a relation representing a relationship R; P is either a class or a one-dot expression.

ALGORITHM **DATALOG_to_OQL**

1. Adding (removing) an evaluable atom.
   Let $X = Y$ ($X \neq Y$) be an atom added to (removed from) the Datalog query.
   - Add (remove) $x = y$ ($x \neq y$) to (from) the **where** clause.

   Let $A\ \theta\ k$ be an atom added to (removed from) the Datalog query. Then, there must be an atom $c(X, \ldots, A, \ldots)$ 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. Then, there must be atoms $c(X, \ldots, A, \ldots)$ and $d(Y, \ldots, B, \ldots)$ in the Datalog query.
   - Add (remove) $x.A\ \theta\ y.B$ to (from) the **where** clause.
2. Adding (removing) a predicate literal.
   Let $c(X, \ldots)$ be an atom added to (removed from) the Datalog query.
   - Add (remove) *C x* to (from) the **from** clause.[6]

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

6. If the **from** clause already contains an expression $P\ x$ and $c(X, \ldots)$ has been added to the Datralog query, then we can substitute $P\ intersect\ C\ x$ for $P\ x$ in the **from** clause.

• Add (remove) *x.R y* to (from) the **from** clause.

Let $\neg c(X, \ldots)$ be an atom added to the Datalog query. Then, there must be an expression *P x* in the **from** clause.

• Substitute *P except C x* for *P x* in the **from** clause.

Let $\neg r(X, Y)$ be an atom added to (removed from) the Datalog query. Then, there must be an expression *P x* in the **from** clause.

• Substitute *P except x.R y* for *P x* in the **from** clause.

4.5 Applications

Next, we present several examples of semantic optimization. In each case, we present an OQL query (already modified to remove path expressions), its Datalog representation Q, one or more integrity constraints that can be applied to the query, an optimized Datalog query Q', and an optimized OQL query. We do not include the full list of attributes for each predicate in a query or an integrity constraint. We only write those attributes which are projected, selected, or joined in a query. All queries refer to the schema of Fig. 2.

Example 3. (Contradiction detection). Consider a query that asks for the names of students taught by faculty members younger than 18 years old. The OQL and Datalog representations are as follows:

select x.name
from Student x
x.takes y
y.is_taught_by z
where z.age < 18

Q(Name)← student(X,Name),takes(X,Y),
is_taught_by(Y,Z),faculty(Y,Age),
Age < 18.

Recall that we had an integrity constraint ($\mathcal{IC}_1$) stating that all faculty members are more than 18 years old. Thus, this query is bound to fail. The Datalog representation of the query is optimized (details of the optimization are given in Example 1), and a contradiction is added to the query. It is as follows and is not evaluated:

Q'(Name)← student(X,Name),takes(X,Y),
is_taught_by(Y,Z),faculty(Y,Age),
Age < 18, Age > 18.

Example 4. (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

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

Suppose we know that all faculty members are more than 30 (since they are more than 18 by $\mathcal{IC}_1$ and the set of all faculty members is a subclass of the set of all persons $\mathcal{IC}_2$). Then, the objects of the class *Faculty* should not be considered when evaluating the query.[7]

person(X,Name,Age)← faculty(X,Name,Age). $\mathcal{IC}_2$

Given $\mathcal{IC}_1$ and $\mathcal{IC}_2$, we can derive the following integrity constraint:

$Age \geq 30 \leftarrow$ faculty(X,Name,Age),
person(X,Name,Age), $\mathcal{IC}_3$

which can also be expressed as:

$\neg$ faculty(X,Name,Age)← person(X,Name,Age),
Age < 30. $\mathcal{IC}_4$

Optimization of Q with $\mathcal{IC}_4$ 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. A heuristic would be used to identify when such extents are maintained so that this optimization may be performed.

Example 5. (Join reduction). 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

7. 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.

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), faculty(W,Name_2), $\text{Name}_1 = \text{Name}_2$.

$\mathcal{IC}_5$ states that *Name* is a key for the relation *Faculty*.

$\text{X}_1 = \text{X}_2 \leftarrow$ faculty($\text{X}_1, \text{Name}_1$), faculty($\text{X}_2, \text{Name}_2$), $\text{Name}_1 = \text{Name}_2$. $\mathcal{IC}_5$

$\mathcal{IC}_6$ states that two objects with the same OIDs are identical.

$\text{Name}_1 = \text{Name}_2 \leftarrow$ faculty($\text{X}_1, \text{Name}_1$), faculty($\text{X}_2, \text{Name}_2$), $\text{X}_1 = \text{X}_2$. $\mathcal{IC}_6$

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:

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.

Example 6. (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 [25]. 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:[8]

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:

Q_1(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:

has_ta(V,W)← takes(X,Y),is_section_of(Y,Z), has_sections(Z,V). $\mathcal{IC}_7$

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 $\mathcal{IC}_7$, 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."

8. This type of relation is called a canonical extension of access support relation for $\mathcal{P}$ in [25].

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.

In all of the examples presented above, only one more efficiently evaluable query has been created for each case. As stated in Section 4.2, this is achieved by first constraining the search of possible transformations by means of heuristics and then using a syntactic optimizer to evaluate the execution cost for each of the semantically equivalent queries.

4.6 Other Types of Queries

1. Existential Quantification:
   This type of query has the following form: *exists x in* e_1*:* e_2, where e_1 is a collection and e_2 is a predicate. It returns *true* if there is at least one element of collection e_1 that satisfies e_2, and *false* otherwise. The translation is done in two steps. First, the existential query is transformed into a select-from-where query of the following form:[9]

   select x
   from e_1 x
   where e_2

   Then, the above query is translated into Datalog by algorithm **OQL_to_DATALOG**, with one modification, i.e., the list *Projected_Attributes* is empty.
   Consider the following OQL query:

   exists x in Sections: x.Taught_by.name = "Turing"

   The Datalog translation of this query is as follows:

   q()← section(X), taught_by(X,Y),
   faculty(Y,Name), Name = "Turing"

2. Universal Quantification:
   A query of this type has the following form: *for all x in* e_1*:* e_2, where e_1 is a collection and e_2 is a predicate. Since universal queries cannot be directly expressed in Datalog, the translation from OQL to Datalog will also occur in two steps. We first note that the following two statements in first-order logic are equivalent:

   - $\forall X\ (p(X) \rightarrow r(X))$
   - $\neg\exists X\ (p(X) \wedge \neg r(X))$

   Thus, the universal query: *for all x in* e_1*:* e_2 is equivalent to an existential query *not*(*exists x in* e_1*:* *not* (e_2)). Let

   $$\mathrm{v}() \leftarrow \mathrm{Body}.$$

   be the Datalog translation of the query *exists x in* e_1*:* *not* (e_2). Then, if v is treated as a view, the query *not*(*exists x in* e_1*:* *not* (e_2)) can be expressed in Datalog as follows:

   $$\mathrm{q}() \leftarrow \neg v.$$

9. If the query that is obtained is a nested query, then it must first be unnested, as described in [15].

5 Semantic Optimization of Queries with Methods

Methods (operations) are a distinct feature of an object query language. They are operations that may be called with user-provided arguments to be executed as part of a query or an update in a database. Methods are defined for a particular class of objects and are applicable to all objects in that class and its subclass(es). In this section, we first discuss a classification of methods; the methods that are considered go beyond the ODMG standard. We then identify the class of methods for which we present a technique for SQO. We present details of SQO for methods that are predefined queries in a query language and for methods which are implemented as arbitrary procedures. For each type of method, we present examples of *shallow* and *deep* level SQO. We illustrate the importance of heuristics for SQO with methods to be successful. We also briefly consider method refinement (of the method implementation).

5.1 Types of Methods for Applying SQO

Methods can be classified on three orthogonal characteristics, as follows:

1. Value returned: A method can
   a. return a value;
   b. not return a value, but have side-effects.
2. Domain of method application: A method may be defined for
   a. a single object;
   b. a collection of objects.
3. Implementation: A method may be implemented as
   a. a predefined query in a query language;
   b. an arbitrary procedure in a programming language.

Methods that return no value but may have side effects (1b) usually implement update procedures in a database. We do not consider SQO for these methods since we do not consider optimization of updates. We only consider methods that return a (simple) value. We note that methods can also have input and output parameters that enable them to return collections of objects [13]; this is not considered here for simplicity.

Methods defined for a collection of objects (2b) may perform aggregate functions, e.g., MAX, MIN, AVG, etc., on

these collections. Since we do not explicitly represent collections of objects in the logic representation, we do not perform SQO on these methods.

For methods that are defined for a single object (2a) and return a value (1a), we distinguish between methods that are implemented as predefined queries (3a) and methods that are arbitrary procedures (3b). SQO can be applied at two different levels for these methods and these levels are informally labeled *deep* and *shallow* level optimization. A query with a method is optimized at a shallow level if the only available semantic knowledge is specific facts about the method. Optimization at a deep level requires that the encapsulation of the method must be broken. To perform optimization at a deep level, we use semantic knowledge about the method definition in the case that the method is a predefined query in a query language. In the case where the method is an arbitrary procedure, the behavior of the method, with respect to the object to which it is applied, must be expressed as a relationship, which is then used for optimization at a deep level. Heuristics are needed to determine if deep level optimization is beneficial in both cases.

To relate to previous research, [2], [33] proposed that the encapsulation of the method be broken and that the implementation be revealed to the optimizer. Such information was to be stated in a form that would be understood and efficiently processed by the optimizer. References [10], [20] suggest rewriting methods by means of equivalences that specify the semantics of methods. In our approach for SQO for methods, we can provide shallow level optimization even when the definition of the method or the method behavior is unknown, i.e., the encapsulation of the method is not broken. However, for deep level SQO, the implementation of a method is revealed by providing the method's definition, or by a relationship describing the behavior of the method. Our technique has an important advantage over existing approaches; we use a uniform framework for describing semantic knowledge of methods and of the database schema. Thus, we are able to generate semantically equivalent queries which could not have been generated without this combined semantic knowledge.

5.2 Predefined Queries

Predefined queries (called POSTQUEL functions in [38]) are methods written in a query language. An example of such a method is *Student_of*, which is defined on the class *Faculty* and takes *Name* (of a faculty member) as its argument. It returns all students who are taught by the faculty member. The method definition is as follows:

define function Students_of(Arg)
select y
from Faculty x
x.Teaches z
z.Is_taken_by y
where x.Name = Arg

We first consider the case where the method definition is not available and only shallow level SQO can be performed. We then consider the situation where the method definition is available and deep level SQO is possible. However, with deep level SQO, there is a need for heuristics to determine if the deep level rewriting is optimal.

Suppose we assume that the method's definition is not available. In this case, if there is semantic knowledge in the form of specific facts about the method, then we can perform shallow level optimization. Suppose we know that a faculty member named *Johnson* has no students (because he does not teach any classes). Then, we can state the following specific fact about this method expressed as an integrity constraint.

$$\leftarrow \text{students\_of(X,Name,Value), faculty(X), Name="Johnson."} \qquad \mathcal{IC}_8$$

Suppose we consider the following query Q and its Datalog representation:

select w
from Faculty x
x.Students_of("Johnson") w

$$Q(W) \leftarrow \text{students\_of(X,Name,W), faculty(X,Name), Name="Johnson."}$$

The integrity constraint $\mathcal{IC}_8$ will identify a contradiction and the query will fail. However, it is highly unlikely that we will obtain such specific facts relevant to the method. It is much more likely that we will *only* have the more general constraint that Johnson does not teach any classes, as follows:

$$\leftarrow \text{faculty(X,Name), teaches(X,Y), Name="Johnson."} \qquad \mathcal{IC}_9$$

Now, for SQO to be successful, we require that the definition of the method *Students_of()* be available to the semantic optimizer. The relational representation of this method definition, expressed as a view definition with parameters, is as follows:

$$\text{students\_of(X,Name,Y)} \leftarrow \text{faculty(X, Name), teaches(X, Z), is\_taken\_by(Z, Y).}$$

The Datalog representation of the query, after substituting the view definition for the method, is as follows:

$$Q(Y) \leftarrow \text{faculty(X, Name), teaches(X, Z), is\_taken\_by(Z, Y), Name="Johnson."}$$

Then, we can use the integrity constraint $\mathcal{IC}_9$ to perform deep level SQO. The SQO technique described in Section 4 is applied directly. SQO will identify a contradiction so that the query must fail. This type of query rewriting has been advocated in [1].

We note that, although both shallow and deep level SQO can be performed for these methods, we still need heuristics to determine if we are producing a more efficient plan. We illustrate with an example.

Example 7. Consider the following query:

$$Q(W) \leftarrow \text{students\_of(Z,Name,W), faculty(Z), Name="Baker."}$$

With the method substituted by its definition, the query becomes:

$$Q_1(W) \leftarrow \text{faculty(Z,Name),teaches(Z,Y), is\_taken\_by(Y,W),Name="Baker."}$$

Assume that *Baker* teaches section CMSC424 only, given by $\mathcal{IC}_{10}$. Also suppose that *Baker* is the only person who teaches that section, as given by $\mathcal{IC}_{11}$.

Number=CMSC 424 ← faculty(Z,Name),teaches(Z,Y),
section(Y,Number),
Name="Baker." $\mathcal{IC}_{10}$

Name="Baker"← faculty(Z,Name), teaches(Z,Y),
section(Y,Number),
Number=CMSC 424. $\mathcal{IC}_{11}$

With these two integrity constraints, and the fact that each section is taught by someone, we can derive the following equivalent query:

Q_1'(W)← section(Y,Number),is_taken_by(Y,W),
Number=CMSC 424.

Q_1' appears simpler than Q_1. However, suppose the method itself was efficiently implemented, e.g., using an access support relation between faculty and their students. Then, Q_1 may be optimized using this access support relation, but not Q_1'. In this case, Q_1' may be more expensive to evaluate than query Q_1. To reliably estimate the cost of each of these queries, a physical optimizer would have to keep information about the implementation (cost) of the methods. Alternatively, if this approach is too complex, we may require the use of a heuristic. A suitable heuristic may be that a method should not be substituted by its definition, and SQO should not be performed using integrity constraints, unless SQO leads to an optimized query which is always cheaper to evaluate, e.g., the optimized query cannot return any answers and, thus, need not be evaluated at all.

5.3 Arbitrary Procedures

Here, we consider a type of method (called C-functions in [38]) which is arbitrary procedures written in some programming language and which cannot be expressed by means of a query language. For example, *Taxes_withheld*, used in Section 4.4, is such a method. Recall that it is applied to objects of class *Employee* and its subclasses; its argument is a tax rate and its output represents the taxes withheld.

As with predefined queries, we distinguish between shallow and deep level optimization for this type of method. Shallow level optimization uses specific facts about the method. For deep level optimization, semantic knowledge about the behavior of the method may be expressed as a function expressing some relationship between the input and output values. Deep level optimization may then compute this function rather than apply the method. As before, heuristics are needed to determine if deep level SQO produces efficient plans. We restrict our attention to methods performing mathematical computations since it is simpler to describe the behavior of a mathematical computation using a function.

For optimization at a shallow level, when the semantics of a method is unknown, we use integrity constraints explicitly stating conditions (facts) about the methods themselves. These may be conditions describing the range of values of a given method independent of the input, e.g., the fact that *Taxes_withheld* returns positive values only, or they may restrict the range of values of the method for a specific subclass.

Example 8. Consider a query that asks for the names of employees whose withheld taxes are less than 1,000, for a tax rate = 10 percent, as follows:

Q(Name) ← taxes_withheld(OID,10%,Value),
employee(OID,Name),Value < 1000.

Suppose we have the fact that all faculty pay more than 3,000 in taxes, which provides the following integrity constraint:

Value > 3000 ← faculty(OID),
taxes_withheld(OID,Rate,Value). $\mathcal{IC}_{12}$

Now, Q can be optimized using our standard techniques for SQO using this integrity constraint and the fact that *Faculty* is a subclass of *Employee* to obtain the following query:

Q'(Name)← taxes_withheld(OID,10%,Value),
employee(OID,Name),
Value < 1000, ¬ faculty(OID, Name).

The potential optimization here is that the scope of applying this method to objects has been reduced since the method is not applied to objects in class *Faculty*. Again, for this optimization to be applied, we may need a heuristic that indicates that the extent of the classes are being maintained. However, as before, it is unlikely that we can obtain integrity constraints referring to the values of methods themselves.

It is more likely that, instead of a specific fact about the value of the method, we may have more general information about the salary of faculty members, which would result in the following constraint:

Salary $> 30k$ ← faculty(OID,Salary) $\mathcal{IC}_{13}$

Unfortunately, $\mathcal{IC}_{13}$ alone is insufficient to perform any optimization for a query that applies this method. We need semantic knowledge about the computation of the method, i.e., the relationship between the input and the result of the method. For this, the encapsulation of a method needs to be broken. The task of extracting information about such relationships is not trivial; it requires both a good knowledge of the method's implementation as well as an understanding of the workings of the semantic optimizer. Clearly, this process cannot be automated.[10] The behavior of a method can be expressed by defining a function as follows:

Definition 4. *Let* **c** *be a class in DB with attributes* $A_1, \ldots, A_n$. *Let* $m(OID, B_1, \cdots, B_m, V)$ *be a method defined on* **c**, *where* $B_1, \cdots, B_m$ *are the input parameters and* V *is the output value computed by the method. Then, this value* V *is defined by a* method function $f_m(A_{i_1}, \cdots, A_{i_k}, B_1, \cdots, B_m)$, *where* $A_{i_j} \in \{A_1, \ldots, A_n\}$ *if the following hold:*

10. It is assumed in [20] that the semantics of a method can only be stated by the method's implementer.

$$f_m(A_{i_1}, \cdots, A_{i_k}, B_1 \cdots, B_m) = V$$
$$\text{iff } m(OID, B_1, \ldots, B_m, V),$$
$$\mathbf{c}(OID, A_1, \ldots, A_n)$$

for all instances of DB.

Given such a function f_m expressing the behavior of the method, we may now express integrity constraints that refer to this function and thus describe the behavior of the method. A constraint about the function can be either a simple constraint or a complex constraint. We first define a simple constraint.

Definition 5. *Let* $f_m(A_{i_1}, \cdots, A_{i_k}, \ B_1, \cdots, B_m) = V$ *be a function computing method* m. *A* simple method constraint *has the following form:*

$$f_m(A_{i_1}, \cdots, A_{i_k}, B_1, \cdots, B_m)\theta V \leftarrow A_{i_1}\theta g_1(\mathcal{A}_1), \cdots, A_{i_k}\theta g_k(\mathcal{A}_k),$$

where

$$\theta \in \{=, \neq, <, >\}, A_{i_j} \in \{A_1, \cdots, A_n\}, \ 1 \leq j \leq k,$$

$\mathcal{A}_i \subseteq \{A_1, \cdots, A_n, B_1, \cdots, , B_m, V\}$, *and* g_i *is an arbitrary function.*

A simple constraint may be used to add new selection conditions to the query, with the potential of reducing the number of objects for which the method has to be computed. Alternately, a simple constraint may be used to identify a contradiction where the method need not be computed. Finally, several constraints may imply a full equivalence between a method (in a given query) and some combination of functions $g_1, \ldots, g_k$, thus allowing us to replace the computation of a method with computing the functions $g_1, \ldots, g_k$. We illustrate with examples.

Example 9. Consider the query of Example 8 that asks for the names of employees whose withheld taxes are less than 1,000, for a tax rate = 10 percent, as follows:

Q(Name) ←
taxes_withheld(OID,10%,Value),
employee(OID,Name),$\text{Value} < 1000$.

Suppose we know that if the salary of employees is larger than or equal to 50k, then their taxes are larger than or equal to 1,000. This can be expressed as the following method constraint:

$$\text{Salary} < 50\text{k}. \leftarrow \ \text{f}_{taxes}(\text{Salary}, \text{Rate}) < 1000. \qquad \mathcal{IC}_{14}$$

SQO can now use the method constraint to simplify the query. Since the Value returned by the method is constrained to be less than 1,000, we can compute withheld taxes for only those employees with salary less than 50k, as follows:

Q′(Name) ← taxes_withheld(OID,10%,Value),
employee(OID,Name,Salary),
$\text{Value} < 1,000$, $\text{Salary} < 50\text{k}$.

With the simple method constraint above, we restricted the scope of objects for which the method was to be applied. Next, we consider simple method constraints that imply a full equivalence between a method (in a given query) and some combination of functions $g_1, \cdots, g_k$, thus allowing the replacement of the computation of the method with computing the functions $g_1, \cdots, g_k$.

Example 10. We consider a modified example from [20]. A method *age(x,y)*, defined for objects of class *Person*, takes as input values x and y. This method returns a (binary) value *true* if the *age* of the person to which this method is applied is equal to x and the current year is y. Thus, the behavior of the method is that it returns a value of *true* if $year\_of\_birth = (y - x)$, for that person. Consider a query which asks for persons of age = 55 with current year equal to 1997 and its Datalog representation, as follows:

select x
from Person x
where x.age(55, 1997)

Q(X) ← person(X), age(X,55,1997,True)

A simple plan for evaluating this query would apply the method to all objects of *Person*. Suppose, however, that we consider two method constraints that completely characterize the behavior of this method, i.e., the relationship between the input and the output value that is computed, as follows:

$$\text{f}_{age}(\text{Year\_of\_Birth}, \text{Age}, \text{This\_year}) \neq \text{True} \leftarrow \text{Year\_of\_Birth} \neq \text{This\_year} - \text{Age}. \quad \mathcal{IC}_{15}$$

$$\text{f}_{age}(\text{Year\_of\_Birth}, \text{Age}, \text{This\_year}) = \text{True} \leftarrow \text{Year\_of\_Birth} = \text{This\_year} - \text{Age}. \quad \mathcal{IC}_{16}$$

Given these two constraints, we have the following equivalence:

$$\text{f}_{age}(\text{Year\_of\_Birth}, \text{Age}, \text{This\_year}) = \text{True} \equiv \text{Year\_of\_Birth} = \text{This\_year} - \text{Age}.$$

Given this equivalence, instead of computing the method for objects of type *Person*, we can compute the function that completely describes the behavior of the method, i.e., we can simply select persons whose *Year_of_Birth* satisfies the condition of the constraint. The simplified query is as follows:

$$\text{Q(X)} \leftarrow \text{person(X, Year\_of\_Birth)}, \ \text{Year\_of\_Birth} = \text{This\_Year} - \text{Age}.$$

Whether or not method constraints can be useful in improving the efficiency of query evaluation depends primarily on the cost of computing the functions $g_1, \cdots g_k$. If the arguments to these functions are fully instantiated in the query, i.e., they are user-provided parameters $B_1, \cdots, B_m$ or they are attributes $A_1, \cdots, A_n$ that are restricted by equality to be constants, then the values of these functions can be computed before the query is evaluated. Thus, even if computing these functions is expensive, it may still be worthwhile to precompute them and rewrite the query, using these functions, before the query is processed. However, if the arguments in $g_1, \cdots, g_k$, i.e., $A_1, \cdots, A_n$, are not instantiated in the query and their values need to be retrieved from objects during query processing, then the optimizer would need to determine the cost of computing $g_1, \cdots, g_k$, in order to determine if this

optimization is worthwhile. Ideally, these functions would be just identity functions, as in the cases above, and we could use heuristics to apply SQO.

Finally, we consider complex method constraints that may describe mathematical properties about functions that describe the behavior of the methods. These properties of the functions could include if they are one-to-one, monotonic, nonmonotonic, etc. This semantic knowledge may refer to the relation between the input of methods and output in a more abstract way, compared to simple method constraints.

Definition 6. *Let m be a method computed by the function $f_m(A_1, \ldots, A_n) = V$. A method constraint is* complex *when it has the following form:*

$$f_m(A_1, \ldots, A_n)\theta f_m(B_1, \ldots, B_n) \leftarrow A_{i_1}\theta_1 B_{i_1}, \ldots, A_{i_k}\theta_n B_{i_k},$$

where

$$\theta, \theta_1, \ldots, \theta_n \in \{=, \neq, <, >\},$$
$$\mathcal{A}_i \subseteq \{A_1, \cdots, A_n\}, \mathcal{B}_i \subseteq \{B_1, \cdots, B_n\}.$$

Example 11. We consider (again) the *Taxes_withheld* method. We consider a constraint specifying that the method *Taxes_withheld* is monotonic with respect to *Salary*. The constraint is thus as follows:

$$\mathrm{f}_{taxes}(\mathrm{Salary1},\ \mathrm{Rate}) > \mathrm{f}_{taxes}(\mathrm{Salary2},\ \mathrm{Rate}) \leftarrow \mathrm{Salary1} > \mathrm{Salary2}. \qquad \mathcal{IC}_{17}$$

Assume that, for $\mathrm{Rate} = 10\%$ and the $\mathrm{Salary} = 30\mathrm{k}$, the value of the method is 3,000, i.e., the following constraint holds:

$$\mathrm{f}_{taxes}(30k, 10\%) = 3,000. \qquad \mathcal{IC}_{18}$$

Thus, the following constraint can be derived:

$$\mathrm{f}_{taxes}(\mathrm{Salary}, 10\%) > 3,000 \leftarrow \mathrm{Salary} > 30\mathrm{k}. \qquad \mathcal{IC}_{19}$$

Consider again the fact given earlier as $\mathcal{IC}_{13}$:

$$\mathrm{Salary} > 30\mathrm{k} \leftarrow \mathrm{faculty}(\mathrm{OID}, \mathrm{Salary}).$$

From $\mathcal{IC}_13$ and $\mathcal{IC}_19$, we can derive the following constraint which states that faculty pay over $\$3,000$ in taxes for the rate 10%.

$$\mathrm{f}_{taxes}(\mathrm{Salary}, 10\%) > 3000 \leftarrow \mathrm{faculty}(\mathrm{OID},\ \mathrm{Salary}). \quad \mathcal{IC}_{20}$$

Now, consider the following query which instantiates the *Rate* to *10 percent* and requests employees whose taxes withheld are less than 3,000:

select x
from Employee x
where x.Taxes_withheld$(10\%) \leq 3,000$

$\mathcal{IC}_{20}$ can be used by SQO to determine that the method should not be applied to objects of *Faculty*.[11]

5.4 Method Overloading

Overloading of method names provides a particular method implementation for each class along an inheritance path [42]. For example, the method *Taxes_withheld* can have different implementations depending on whether the method's receiver, i.e., the object to which the method is applied, is of the type *Employee* or the type *TA*. In the single inheritance case, the method definition that is closest to the lowest-level subtype of the receiver is used. This case is called *simple dispatching* [42] and we only consider this case.

Method name overloading has consequences for SQO. If there are multiple implementations of a method, then there can be integrity constraints, each describing a different implementation of the same method name. One possibility for distinguishing among them is to use a subscript to refer to a class name. Suppose the method *Taxes_withheld* has two different implementations, one for the class *Employee* and another for the class *TA*, but only the former is monotonic. Then, we can express this information as follows:

$$\mathrm{f}_{taxes\_emp}(\mathrm{Salary1}, \mathrm{Rate}) > \mathrm{f}_{taxes\_emp}(\mathrm{Salary2}, \mathrm{Rate}) \leftarrow \mathrm{Salary1} > \mathrm{Salary2}.$$

The **OQL_to_DATALOG** algorithm must be modified to use the method name with the appropriate subscript. This is not a problem if the lowest subclass to which the receiver belongs is known at compile time. If this is not the case, then there are two cases. First, the subscripts will not be used and, so, SQO using the method constraints or facts describing the methods will not be applicable. The alternative is to rewrite the query so that we consider all possibilities for the lowest common subtype. The optimizer can then optimize this query appropriately. The following procedure describes this process: Assume for simplicity that the extent of a class is equal to the union of the extents of its immediate subclasses. Let Q be a query with a method M applied to an object X in class C. Let $C_1, \ldots, C_n$ be the lowest subclasses of C in which different versions $M_1, \ldots, M_m$ of M are defined. Subclasses of C to which the same method implementation applies can be clustered together into the following sets: $\mathcal{C}_{M_1}, \ldots, \mathcal{C}_{M_m}$ such that M_i applies to all objects of C_{M_i}. Let Q_i be the query Q to which the statement $X\ in\ \mathcal{C}_{M_i}$ has been added. Then, Q can be expressed as the union of the Q_is, $1 \leq i \leq m$. Now, each Q_i can be optimized using information about the implementation of M_i.

Clearly, the algorithm should be applied only when the information about the implementation of the method guarantees substantial savings in the optimized query. This, again, can only be determined using appropriate heuristics.

6 Conclusions and Future Work

In this paper, we have presented a logic-based approach to SQO in object databases that generalizes previous research. We handle a large class of OQL queries, including queries with constructors and methods. The object schema is represented using a logical representation (Datalog). Semantic knowledge about the object data model, e.g., class hierarchy information, relationship between objects, etc., as well as semantic knowledge about the particular schema and application domain, are expressed as integrity constraints. An OQL object query is represented as a logic query and query optimization is performed in the Datalog

11. This derivation also uses the fact that *Faculty* is a subclass of *Employee*.

representation. We obtain equivalent (optimized) logic queries and, subsequently, obtain equivalent (optimized) OQL queries, for each equivalent logic query.

We demonstrate a number of possible optimization strategies using SQO and particular integrity constraints. SQO can identify contradictions and, thus, eliminate queries containing contradictions. SQO can be used to possibly simplify queries. This includes reducing the access scope for evaluating a query from a class to some particular subclass(es); this can reduce the number of objects that are retrieved from the database. SQO can simplify queries by eliminating joins. One strategy to eliminate joins is using integrity constraints in the form of key constraints. Another strategy is to use *access support relations*, or special relations that store OIDs of objects that represent a materialized view, corresponding to some path expression.

SQO can be applied to methods that are predefined queries and arbitrary procedures. Simplification of the query, by SQO, may involve limiting the method application to objects in particular subclass(es) or adding new conditions to the query, which serves to reduce the actual objects on which the method is applied. A query with a method is optimized at a shallow level if the only available semantic knowledge is specific facts about the method. Optimization at a deep level requires that the encapsulation of the method must be broken. To perform optimization at a deep level, we use semantic knowledge about the method definition in the case that the method is a predefined query in a query language. In the case where the method is an arbitrary procedure, semantic knowledge about the behavior of the method may be expressed as a function expressing some relationship between the input and output values. Deep level optimization may then compute this function, rather than apply the method.

SQO makes extensive use of integrity constraints. Heuristics are needed to determine if SQO is beneficial to reduce the space of semantically equivalent queries that must be considered by the cost-based physical optimizer. In many cases, the heuristics may be dependent on the particular object database implementation. This is an area for future research.

The Flora semantic query optimizer for OQL, developed at INRIA [16], [18], [17], provides tools for unnesting queries. We expect to utilize this utility. It also has a rewriting capability that has been used to implement several of the SQO transformations described in this paper. It does not provide a general treatment for SQO and does not provide the techniques for optimizing queries with methods as is discussed in this paper. However, it is capable of handling a larger class of queries since it is not limited by a Datalog representation for optimization and it uses pattern-match based rewriting, based on a strongly typed object algebra. In future research, we hope to incorporate our techniques within this implementation so that we may explore a greater space of semantically equivalent queries. We also expect to determine the applicability of SQO to object-relational databases.

We also propose utilizing our general framework (representation and query rewriting) for SQO to transfer other techniques developed for relational databases to object databases. Such techniques include *semantic query caching* (SQC) [12], [11], [24] and *query rewriting using views* [36], [31], [9], [39]. We also plan to revise the work in the context of ODMG 2.0.

The benefit of *semantic query caching* (SQC) is obtained by using the cached results of previous queries to save on some computation.

Example 12. Assume the results of the following query are cached:

select **list**(x.name, x.age)
from Student x
where $\mathrm{x.age} \leq 19$

and the following query is posed to a database:

select **struct**(x.name, x.age)
from Student x

Now, a part of the answer set for the current query can be retrieved from the cache and the remainder need only request student names and ages for students older than 19.

We can use similar techniques, as described in this paper, to represent the cached query, and to determine if the result of the cached query can be used to answer this query. Further, we need to determine if the format of the answer of the cached query is compatible with the answer of the new query.[12]

The ability to use materialized views to answer queries is important in many applications: global information systems [30], mobile computing [4], and view adaptation [22]. Algorithms for relational queries are presented in [36], [31], [9], [39] and algorithms for object queries are presented in [18]. We note that a limited rewriting using integrity constraints is proposed in [18]. Since we express integrity constraints and views in the same representation, our framework will allow us to provide more general techniques for rewriting queries using views in object databases.

APPENDIX

The following is the result of the transformation of the OODB schema of Fig. 2 by means of the algorithm described in Section 4.3.

RELATIONS

1. Classes
   person(OID,Name,OID_S)
   employee(OID,Name,Id,Salary,OID_S)
   student(OID,Name,Id,Salary,OID_S,Student_id)
   ta(OID,Name,Id,Salary,OID_S,Student_id)
   faculty(OID,Name,Id,Salary,OID_S,Rank)
   course(OID,Name,Number)
   section(OID,Number)
2. Structures:
   address(OID,City,Street,Number)
3. Relationships:
   children($\mathrm{OID}_1, \mathrm{OID}_2$)

12. For example, if the answer set of cached query is stored as a set and a new query requests the answer set to be a bag, then, in general, the cached query cannot be used.

parents(OID_1, OID_2)
spouse(OID_1, OID_2)
has_prerequisites(OID_1, OID_2)
is_prerequisite_for(OID_1, OID_2)
has_section(OID_1, OID_2)
is_section_of(OID_1, OID_2)
takes(OID_1, OID_2)
is_taken_by(OID_1, OID_2)
assists(OID_1, OID_2)
has_TA(OID_1, OID_2)
teaches(OID_1, OID_2)
is_taught_by(OID_1, OID_2)

INTEGRITY CONSTRAINTS

Since there are many integrity constraints for the schema of Fig. 2, we present here only a representative sample of them including all integrity constraints used in the examples.

1. OID Identification
   section(OID_1,Number)$\leftarrow$ is_section_of(OID_1, OID_2)
   course(OID_2,Name,Number)
   $\leftarrow$ is_section_of(OID_1, OID_2)
   address(OID_1,City,Street,Number)
   $\leftarrow$ person(OID,Name,OID_1)
2. Object identities
   $\leftarrow$ section(OID,Number),
   course(OID,Name,Number)
3. Subclass hierarchy
   person(OID,Name,OID_S)
   $\leftarrow$ employee(OID,Name,Id,Salary,OID_S)
4. Inverse relationships between classes
   teaches(OID_1, OID_2)
   $\leftarrow$ is_taught_by(OID_2, OID_1)
   is_taught_by(OID_1, OID_2)
   $\leftarrow$ teaches(OID_2, OID_1)
5. Many-to-one constraints
   $OID_2 = OID_3 \leftarrow$ is_taught_by(OID_1, OID_2),
   is_taught_by(OID_1, OID_3)
6. Key and dependency constraints
   $\mathrm{Name}_1 = \mathrm{Name}_2 \leftarrow$ course(OID,$\mathrm{Name}_1, \mathrm{Number}_1$),
   course(OID,$\mathrm{Name}_2, \mathrm{Number}_2$)
   $\mathrm{Number}_1 = \mathrm{Number}_2$
   $\leftarrow$ course(OID,$\mathrm{Name}_1, \mathrm{Number}_1$),
   course(OID,$\mathrm{Name}_2, \mathrm{Number}_2$)
   Assume that $Number$ is the key for the relation $course$. Then, the following constraint should be added.
   $OID_1 = OID_2 \leftarrow$ course(OID_1, Name_1,Number),
   course(OID_2, Name_2,Number)

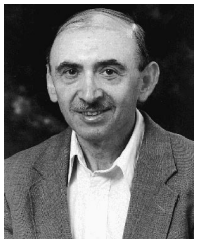
ACKNOWLEDGMENTS

We would like to thank the referees for the many constructive comments that they have made.

REFERENCES

[1] K. Aberer and G. Fischer, "Semantic Query Optimization for Methods in Object-Oriented Database Systems," *Proc. IEEE Int'l Conf. Data Eng.*, pp. 70–79, 1995.
[2] F. Bancilhon, S. Cluet, and C. Delobel, "A Query Language for o_2," *Building an Object-Oriented Database System*, pp. 243–255, Morgan Kaufman, 1992.
[3] J. Banerjee, W. Kim, and K.-C. Kim, "Queries in Object-Oriented Databases," *Proc. Fourth Int'l Conf. Data Eng.*, 1988.
[4] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environment," *Proc. SIGMOD*, pp. 1–12, 1994.
[5] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini, "Odb-qoptimizer: A Tool for Semantic Query Optimization in Oodb," *Proc. Int'l Conf. Data Eng., ICDE '97*, pp. 578–578, Apr. 1997.
[6] U. Chakravarthy, J. Grant, and J. Minker, "Logic-Based Approach to Semantic Query Optimization," *ACM Trans. Database Systems*, vol. 15, no. 2, pp. 162–207, June 1990.
[7] U.S. Chakravarthy, J. Grant, and J. Minker, "Semantic Query Optimization: Additional Constraints and Control Strategies," *Proc. Expert Database Systems*, L. Kerschberg, ed., pp. 259–269, Apr. 1986.
[8] U.S. Chakravarthy, J. Grant, and J. Minker, "Foundations of Semantic Query Optimization for Deductive Databases," *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., pp. 243–273, Morgan Kaufmann, 1988.
[9] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing Queries with Materialized Views," *Proc. 11th Int'l Conf. Data Eng.*, pp. 190–200, 1995.
[10] S. Chaudhuri and K. Shim, "Query Optimization in the Presence of Foreign Functions," *Proc. 19th Conf. Very Large Data Bases*, pp. 526–542, 1993.
[11] C.M. Chen and N. Roussopoulos, "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching," *Proc. Fourth Int'l Conf. Extending Database Technology*, 1994.
[12] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *Proc. 22nd Conf. Very Large Data Bases*, 1996.
[13] *Object Database Standard: ODMG-93.* release 1.2, R.G.G. Cattell, ed., Morgan Kaufman, 1996.
[14] L. Fegaras, "Query Unnesting in Object-Oriented Databases," *Proc. ACM Special Interest Group Management of Data*, pp. 49–60, 1998.
[15] D. Florescu, "Design and Implementation of the Flora Object Oriented Query Optimizer," PhD thesis, Dept. of Computer Science, Univ. of Paris 6, 1996.
[16] D. Florescu, L. Raschid, and P. Valduriez, "Using Heterogeneous Equivalences for Query Rewriting in Multidatabase Systems," *Proc. Int'l Conf. Cooperating Information Systems*, 1995.
[17] D. Florescu, L. Raschid, and P. Valduriez, "Answering Queries Using Oql View Expressions," *Proc. Workshop Materialized Views: Techniques and Applications, in conjunction with the ACM Special Interest Group Management of Data Int'l Conf.*, 1996.
[18] D. Florescu, L. Raschid, and P. Valduriez, "A Methodology for Query Reformulation in Cis Using Semantic Knowledge," *Int'l J. Cooperating Information Systems*, 1996.
[19] T. Gaasterland and J. Lobo, "Processing Negation and Disjunction in Logic Programs through Integrity Constraints," *J. Intelligent Information Systems*, vol. 2, no. 3, 1993.
[20] V. Gaede and O. Gunther, "Efficient Processing of Queries Containing User-Defined Predicates," *Proc. Int'l Conf. Deductive and Object-Oriented Databases*, pp. 281–298, 1995.
[21] J. Grant and T. Sellis, "Extended Database Logic: Complex Objects and Deduction," *Information Sciences*, vol. 52, pp. 85–110, 1990.
[22] A. Gupta, I.S. Mumick, and K.A. Ross, "Adapting Materialized Views after Redefinitions," *Proc. ACM Special Interest Group Management of Data*, 1995.
[23] M.T. Hammer and S.B. Zdonik, "Knowledge-Based Query Processing," *Proc. Sixth Int'l Conf. Very Large Data Bases*, pp. 137–147, Oct. 1980.
[24] A.M. Keller and J. Basu, "A Predicate-Based Caching Scheme for Client-Server Database Architectures," *The VLDB J.*, vol. 5, no. 2, pp. 35–47, Apr. 1996.
[25] A. Kemper and G. Moerkotte, "Access Support in Object Bases," *Proc. ACM Special Interest Group Management of Data*, pp. 364–374, 1990.
[26] J.J. King, "Quist: A System for Semantic Query Optimization in Relational Databases," *Proc. Seventh Int'l Conf. Very Large Data Bases*, pp. 510–517, Sept. 1981.
[27] L.V.S. Lakshmanan and R. Missaoui, "On Semantic Query Optimization in Deductive Databases," *Proc. IEEE Int'l Conf. Data Eng.*, pp. 368–375, 1992.

[28] S. Lee and J. Han, "Semantic Query Optimization in Recursive Databases," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 444–451, 1988.
[29] Y.-W. Lee and S.I. Yoo, "Semantic Query Optimization for Object Queries," *Proc. Int'l Conf. Deductive and Object-Oriented Databases,* pp. 467–484, 1995.
[30] A. Levy, D. Srivastava, and T. Kirk, "Data Model and Query Evaluation in Global Information Systems," *J. Intelligent Information Systems,* vol. 5, no. 2, Sept. 1995.
[31] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views," *Proc. ACM Symp. Principles of Database Systems,* pp. 95–104, 1995.
[32] A.Y. Levy and Y. Sagiv, "Semantic Query Optimization in Datalog Programs," *Proc. ACM Symp. Principles of Database Systems,* 1995.
[33] D. Maier, S. Daniel, T. Keller, B. Vance, G. Graefe, and W.J. McKenna, "Challenges for Query Processing in Object-Oriented Databases," *Query Processing for Advanced Database Systems,* J.C. Freytag, D. Maier, and G. Vossen, eds., pp. 337–381, Morgan Kaufmann, 1994.
[34] J.R. McSkimin and J. Minker, "The Use of a Semantic Network in Deductive Question-Answering Systems," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence,* pp. 50–58, 1977.
[35] H.H. Pang, H.J. Lu, and B.C. Ooi, "An Efficient Semantic Query Optimization Algorithm," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 326–335, 1991.
[36] X. Qian, "Query Folding," *Proc. 12th Int'l Conf. Data Eng.,* pp. 48–55, 1996.
[37] S.T. Shenoy and Z.M. Ozsoyoglu, "Design and Implementation of a Semantic Query Optimizer," *IEEE Trans. Knowledge and Data Eng.,* vol. 1, no. 3, pp. 344–361, Sept. 1989.
[38] M. Stonebraker and G. Kemnitz, "The POSTGRES Next-Generation Database Management System," *Comm. ACM,* vol. 34, no. 10, pp. 78–92, 1991.
[39] H.Z. Yang and P.-Å Larson, "Query Transformation for PSJ-Queries," *Proc. 13th Int'l Conf. Very Large Data Bases,* pp. 245–254, 1987.
[40] J.P. Yoon and L. Kerschberg, "Semantic Query Optimization in Deductive Object-Oriented Databases," *Proc. Third Int'l Conf. Deductive and Object-Oriented Databases,* pp. 169–182, 1993.
[41] S.C. Yoon, I.Y. Song, and E.K. Park, "Semantic Query Processing in Object-Oriented Databases Using Deductive Approach," *Proc. Int'l Conf. Information and Knowledge Management,* pp. 150–157, 1995.
[42] *Readings in Object-Oriented Database Systems,* S.B. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.



John Grant received the PhD degree in mathematics from New York University (Courant Institute of Mathematical Sciences) in 1970. From 1970 to 1978, he was on the faculty of the University of Florida, first in the Department of Mathematics and then in the Department of Computer and Information Sciences as a postdoctoral fellow, assistant professor, and associate professor. Since 1978, he has been at Towson University, where he is a professor of mathematics and computer and information sciences. From 1990 to 1992, he was a professor at the University of Maryland Institute for Advanced Computer Studies (UMIACS). He has also been a visiting professor of computer science at the University of Maryland. He has more than 50 publications (primarily journal articles) in databases and logic, including a database textbook.

Jarek Gryz received the PhD degree in computer science from the University of Maryland, College Park, in 1997. He is currently an assistant professor in the Department of Computer Science at York University in Toronto and a visiting scientist at the Center for Advanced Studies, IBM Toronto Laboratory. His interests include database query optimization, heterogeneous database systems, and logic programming.

Jack Minker received the BA degree, cum laude with honors, in mathematics from Brooklyn College in 1949, the MS degree in mathematics from the University of Wisconsin in 1950, and the PhD degree in mathematics from the University of Pennsylvania in 1959. He is a professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland, College Park. He has published more than 150 refereed technical papers in journals, books, and conferences. He is an editor or coeditor of four books devoted to deductive databases and logic programming. He is coauthor of a book, *Foundations of Disjunctive Logic Programming*. He serves on the editorial board of a number of journals. He was the first chairman of the Department of Computer Science (1974-1979) at the University of Maryland. He served as chairman of the Advisory Committee on Computing to the National Science Foundation (1979-1982). In 1985, he received the ACM's Outstanding Contribution Award for his work in human rights. He was elected a fellow of the American Association for the Advancement of Science based on his work in artificial intelligence, database theory, and his efforts in behalf of human rights. He is a founding fellow of the American Association for Artificial Intelligence, a founding fellow of the ACM, and a fellow of the IEEE.

Louiqa Raschid received the Btech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1980, and the PhD degree in electrical engineering from the University of Florida, Gainesville, in 1987. Since 1987, she has been at the University of Maryland at College Park. She holds a joint appointment with the Smith School of Business, the Institute for Advanced Computer Studies and the Department of Computer Science (affiliate). She was promoted to associate professor in September 1993. Her research interests include the following: scalable architectures for wide-area heterogeneous information servers, query optimization and evaluation techniques for heterogeneous distributed environments, semantic query optimization for object and object-relational databases, fixpoint and declarative semantics for rule-based programs, and updates in database systems. She is codirector of the Laboratory for Computational Linguistics and Information Processing. Since 1994, she has been a visiting scientist at the French National Laboratories for Information Sciences (INRIA). She has also been a visiting scientist with Hewlett Packard Research Labs and the Stanford Research Institute. Dr. Raschid serves on the editorial board of the INFORMS *Journal of Computing* and on the advisory board of the *Journal of Networks and Information Systems* (HERMES, France). She is a member of the IEEE, the ACM, and the Society of Women Engineers.