Semantic Query Optimization for ODMG-93 Databases*

M. F. van Bommel
Department of Mathematics and Computer Science
St. Francis Xavier University
Antigonish, Nova Scotia, Canada B2G 2W5
mvanbomm@stfx.ca

Abstract

We present a graphical technique for semantic query optimization for ODMG-93 compliant databases. The OQL object query is represented as a graph based on the information contained in the ODL object schema. The graph reflects the constraints on possible objects satisfying the query. Vertices represent objects, which may be members of interfaces, ranges of values, constant values, objects returned by subqueries, object collections (count, sum, etc.), or constructs (sets, lists, etc). Key constraints and inverse relationships contained in the schema are used to perform join elimination, join simplification, and scan reduction via graph transformations. A marking scheme on the graph is used to perform subquery to join transformations and the detection of unnecessary duplication elimination. We argue that our approach generalizes previous research on object-based query optimization by providing a more natural representation of the query allowing simpler, more intuitive transformations, and leading to more efficient evaluation plans.

1. Introduction

Semantic query optimization, which employs knowledge captured in the data model in the form of integrity constraints to transform a query into an equivalent one which may be executed more efficiently, has great benefit when applied to query processing in object databases. Early work on queries for the relational model have achieved some success; however, little research has focused on object-oriented models. One such model, ODL, the data model of the Object Database Standard ODMG'93 [5], includes key constraints and inverse relationships which can be used in optimizing queries expressed in the query language of ODMG'93, OQL. Possible optimizations include subquery to join transformations, join elimination, join simplification, scan reduction, and the removal of unnecessary duplicate elimination. The methodology proposed also can to assist in join order selection, but this discussion is beyond the scope of this paper.

Many works propose optimizations based on constraints and rules, but focus on the relational model or knowledge representation languages. These approaches fail to fully capture the knowledge and meaning of OQL queries, especially in the area of path traversal, and fail to include the interactions of the inverse relationships of ODL. Works on optimizing object databases either support only a small subset of possible optimizations or translate the object query to a more traditional language. One such work [7] targets ODMG databases, but translates the schema and query to DATALOG, thus losing some of the original domain knowledge and object-oriented representation of the data. In fact, for final optimization, the original text of the query is required to regain some of the constructs, such as the constructors *set* and *list* and collection expressions *count*, *unique*, and *sum*.

In this work, we focus on a natural representation of the query as a graph, with nodes representing objects or constructs, and edges properties. Cycles in the graph are used to represent inverse constraints. Supernodes, or nodes containing nodes, are used to represent join and selection conditions in the query. With this representation, we eliminate the extra overhead encountered in translating the schema and query to DATALOG, and the optimized query to OQL. We also benefit from the ease of manipulation of the graph for optimization. Further, we do not require analysis of the original query in the final rewriting of the optimized query back to OQL.

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and by the St. Francis Xavier University Council for Research.

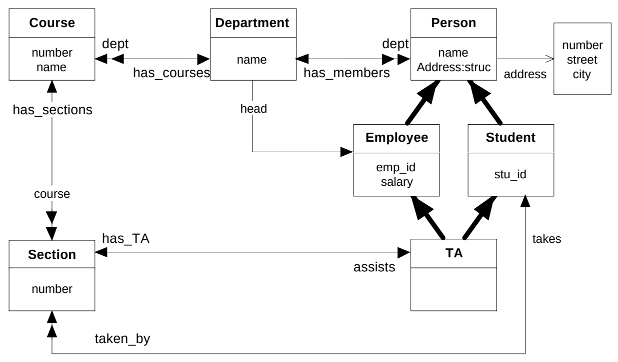

Figure 1. University schema graph.

As an example of the type of transformations possible, consider the following query on the partial university schema of Table 1 (also illustrated in Figure 1). The query is to retrieve all those teaching assistants who assist some section of a course where the head of the department offering the course is the same as the head of the department of the teaching assistant of the particular section. One possible writing of the query is as follows:

```
SELECT DISTINCT t
FROM   TA as t,
       t.assists as s
WHERE EXISTS
   SELECT *
   FROM   Course as c
   WHERE  s.course = c
   AND    c.dept.head = s.has_TA.dept.head.
```

It follows from the key constraint on the head of a department that the two departments are the same, but it is less obvious that the subquery can be transformed into a join and that the DISTINCT clause is unnecessary. Also, careful analysis and manipulation is required to determine that the last join condition can be rewritten as "`c.dept = t.dept`".

The remainder of this paper focuses on the development of a tool to perform the above optimizations and is organized as follows. Section 2 examines previous work in semantic query optimization. This is followed by definitions and notation in section 3, and the construction of the graphical representation of a query in section 4. Section 5 discusses the types of optimizations that are possible using our approach, and summary comments and future work are presented in section 6.

```
interface Person
(extent people)
{    attribute String name;
     attribute Struct Address
        {Short number,
         String street,
         String city} address;
     relationship Department dept;
        inverse Department::has_members;
};

interface Student :  Person
(extent students key stu_id)
{    attribute Short stu_id;
     relationship Set<Section> takes
        inverse Section::taken_by
};

interface Employee :  Person
(extent employees key emp_id)
{    attribute Short emp_id;
     attribute Short salary;
};

interface TA : Employee, Student
(key assists)
{    relationship Section assists
        inverse Section::has_TA;
};

interface Department
(extent depts keys name, head)
{    attribute String name;
     relationship Employee head;
     relationship Set<Person> has_members
        inverse Person::dept;
     relationship Set<Course> has_courses
        inverse Course::dept;
};

interface Course
(extent courses key (dept, number))
{    attribute String number;
     attribute String name;
     relationship Department dept;
        inverse Department::has_courses;
     relationship Set<Section> has_sections
        inverse Section::course;
};

interface Section
(extent sections keys (course, number), has_TA )
{    attribute Short number
     relationship Course course
        inverse Course::has_sections;
     relationship Set<Student> taken_by
        inverse Student::takes;
     relationship TA has_TA
        inverse TA::assists;
};
```

Table 1. The University Schema.

2. Background

Semantic query optimization using domain knowledge has been studied for some time. Many such works involve a knowledge base of rules about the application domain, while others focus on integrity constraints.

Hammer and Zdonik [8] use a knowledge base about the application domain and a database to perform query transformations. A pattern matcher and a rewriter replace subexpressions in a query by more efficient ones. The queries are expressed in a entity-relationship style.

King [11] transforms queries to reflect the explicit semantic rules in a knowledge base for a relational database system. The constraints are on values only; no dependencies are used. Optimizations include index introduction, join introduction, scan reduction, join elimination, and the detection of unsatisfiable conditions.

Jarke, Clifford, and Vassiliou [9] deal with semantic query optimization in the context of an optimizing PROLOG front-end to a relational database system. Function-free conjunctive queries are considered. Three types of integrity constraints are allowed: value bounds, functional dependencies, and referential constraints. A query graph is constructed to deal with inequalities and tableaux are used with functional dependencies to semantically optimize queries.

Shenoy and Ozsoyoglu [15] optimize based on subset and implication integrity constraints which express rules that restrict the domains of attributes. A query is represented by a query graph which is modified for optimizations. Transformations include restriction elimination, index introduction, scan reduction, and join elimination.

Chakravarthy, Grant, and Minker [6] discuss semantic query optimization in the framework of deductive databases, and deal with a very general class of integrity constraints expressed as Horn clauses. In this general framework, all of the above optimizations are possible, as well as transformations that answer the query. Grant *et al.* [7] continue this work by defining the translation of ODMG-93 standard ODL databases and OQL queries to DATALOG, and applying the same optimizations, leading to a translation back to optimized OQL queries. Such a translation takes into account the key constraints and inverse relationships of the OQL query, but loses some of the semantic intent, requiring access to the original query in the final rewriting. They do argue that the ability to express additional integrity constraints explicitly would be a useful extension to the object model to increase the capability to perform semantic query optimization.

Other works involving query optimization in object databases are on smaller classes of queries with less optimizations possible [12, 14, 18]. Bergamaschi *et al.* [1, 2, 3] describe query optimization using description logics inference techniques on integrity constraint rules. A database is expressed as a set of inclusion statements and a query is transformed into a virtual description logics type. Subsumption is then used to transform a query into its most specialized generalization classes, thus reducing its complexity. Borgida and Weddell [4] discuss the inclusion of key constraints and functional dependencies in Classic, the "most expressive" description logic for which reasoning is tractable, while still possessing polynomial-time subsumption and consistency checking algorithms.

The approach in this paper is based on a more natural representation of an object oriented database query. The transformation of such a query to a knowledge base representation, horn clauses, or description logics loses some of the semantics of the underlying data model and query language. A graphical representation maintains the semantics by encoding everything as objects connected via property links.

3. Definitions

To begin, we note that we restrict OQL queries to the **select-from-where** queries. Further, we include only those operations that can be abstracted as a class. This covers the majority of commonly asked queries. The **select** clause is assumed to contain so called *projection attributes*, which are either expressions involving simple identifiers defined in the **from** clause, or definitions of such identifiers based on existing identifiers or subqueries, such as "`d.name as n`" and "`(select max(salary) from Employee) as m`." The **from** clause also defines such identifiers using expressions or subqueries. The **where** clause contains join conditions (*e.g.* "`s.course = c`"), projection conditions (*e.g.* "`d.division = 1`"), membership tests (*e.g.* "`d.head in Employee`"), and subqueries. With these limitations, we now generalize our notion of a class and a property, and provide some other basic definitions and notation.

A *class* is either an interface in the schema (*interface class*), a query to be optimized or a subquery of that query (*query class*), a range of values for an attribute (*range class*), a constant value (*constant class*), a collection expression such as *exists, unique, count, sum* etc. (*collection class*), a constructor expression such as *struct, set, list* etc. (*constructor class*), or a relational or Boolean operator (*operator class*). The subsumption of a class C by a class C' is denoted $C \leq C'$.

The *properties* of a class depend on the type of class. An interface class has properties that are either attributes or relationships of the interface. The properties of a query class include the identifiers of the projection attributes and variable definitions. Range and constant classes have no properties. Collection and constructor classes have properties representing the elements being collected or constructed, respectively. Finally, operator classes have two properties, one for the operand on the left side of the operator, and one for that on the right side.

The *type* of a property P for a class C, written $Type(C, P)$, is the type defined for it (in the query if it is a identifier, in the schema if it is an attribute or relationship, or by the collection or constructor expression) or a type subsumed by that type. Properties are assumed to be single valued, which is sufficient for our purposes, since lists, sets, and bags are considered classes themselves.

A *query graph* $\mathbf{G}$ is a quadruple $< N, S, E, r_G >$, consisting of a set N of nodes labeled with class names, a set S of supernodes, a set E of edges labeled with attribute names, and a distinguished root node r_G in N labeled with the name of the query class. A *supernode* s is a set of one or more nodes, called *subnodes*. The supernode s of a node n is denoted $\texttt{Super}(n)$. Elements of E will be written $< s, n, P >$, where s is a supernode, n a node, and P a property name. Elements of N will be written $n : C$, where C is the class name for the node, denoted $Cl(n) = C$.

A *path* in a query graph is a sequence of edges, starting with an edge from a supernode to a node, then from the supernode of the node to another node, and so on. A *path function*, denoted pf, is either a query variable, or a query variable followed by a sequence of property names, separated by dots. A path function *describes* a path in a query graph if it contains the sequence of edge labels in the path from the root vertex r_G.

A *join condition* is a statement in the WHERE clause of a query of the form $pf_1 = pf_2$, and denotes that the two paths reach the same node for any possible query graph satisfying the query. In the sample query above, the equation "`s.course = c`" is a join condition.

An *inverse constraint* in the schema definition has the form **relationship** C_1 P_1 **inverse** C_2::P_2. Note that it is not assumed that C_1 and C_2 are the same class since one or both may be collection or constructor classes. This is evident from the Course interface in the inverse constraint "**relationship** `Department dept` **inverse** `Department::has_courses`" and the constraint "**relationship** `Set<Section> has_sections` **inverse** `Section::course`", where the latter employs a constructor class.

4. Query Graph

Several functions are required for the creation of the query graph. First, the MERGE function in Figure 2 is a recursive function to combine nodes to reflect inverse constraints in the schema. Supernodes are combined and located using the UNION function if Figure 2, which is an extension of that by Nelson and Oppen [13]. The retrieval of $\texttt{Super}(n)$ can be done in constant time using the approach in [13]. UNION takes time linear in the sum of the lengths of the two supernodes being combined plus the time to modify the edges.

The third function locates a node reached by a path function from the root node. The LOCATE function in Figure 2 follows a path if it exists, or creates one if not, and returns the node.

Finally, the process of creating a query graph is described by function CREATE in Figure 3, which has been proven to reflect the interaction of the key and inverse constraints with the conditions in the query, given the subsumption hierarchy. A query graph $\mathbf{G}_Q$ for query Q is generated by executing the eleven steps of the algorithm, representing modifications to represent the various elements of the query. Note that the nodes created for the variables in the SELECT clause must be marked for retrieval.

The construction of the graph for the query on the university schema is illustrated in Figure 4. Not indicated is that the node for variable "t" is marked for retrieval. Note that supernodes are indicated by ovals except where they only contain a single node, then just the node is shown. Graph (a) shows the result after the first eight steps of CREATE, where the root node is created in step 1, the node labeled "DISTINCT" is added in step 2, edges are added for the query variables in steps 3 and 4, the node for the subquery is created in step 5, and the supernodes (indicated by dotted ovals) are created in step 8. Graph (b) shows the result after a call to UNION in step 9 for the key constraint on heads of departments. Graph (c) illustrates the result after a call to MERGE in step 10 for the inverse constraint on the assistant of a section. Graph (d) illustrates the final graph for the query after removal of the distinct clause and the conversion of the subquery to a join—two processes to be discussed.

The other steps in CREATE are to handle elements of a query not in the example. Step 6 handles subsumption restrictions from the **where** clause of a query by enforcing objects to belong in particular classes. Step 7 handles constants contained in the query. Finally, step 11 removes redundant variables in the query; that is, variables which refer to objects for which there exists another variable in the query.

```
function MERGE($n_1$,$n_2$)
  1. If $n_1 = n_2$ then return.
  2. If $n_2$ is the root $r_G$ of some graph, then $r_G = n_1$.
  3. Relabel $n_1$ with the greatest common subclass of
     classes in the labels of $n_1$ and $n_2$.
  4. For each supernode $s$ and property $P$ where there exists
     an edge $< s, n_2, P >$ in $E$, modify the edge to be
     $< s, n_1, P >$.
  5. For each node $n$ and property $P$ where there exists an
     edge $< \mathtt{Super}(n_2), n, P >$ in $E$, remove the edge from
     $E$. If there exists a node $n'$ where there exists an edge
     $< \mathtt{Super}(n_1), n', P >$, then MERGE($n$,$n'$); otherwise
     add edge $< \mathtt{Super}(n_1), n, P >$.
  6. UNION($n_1, n_2$).
  7. Remove node $n_2$ from $N$ and return.
```

```
function UNION($u$, $v$)
  1. If $\mathtt{Super}(u) = \mathtt{Super}(v)$ then return.
  2. For each node $n$ and property $P$ where there exists an
     edge $< \mathtt{Super}(v), n, P >$ in $E$,
     (a) if there exists an edge $< \mathtt{Super}(u), n', P >$ in
         $E$ then remove $< \mathtt{Super}(v), n, P >$ from $E$ and
         MERGE($n$, $n'$),
     (b) otherwise modify the original edge in $E$ to be
         $< \mathtt{Super}(u), n, P >$.
  3. For each node $n$ in $\mathtt{Super}(v)$, assign $\mathtt{Super}(n)$ to be
     $\mathtt{Super}(u)$.
  4. Add all nodes in $\mathtt{Super}(v)$ to $\mathtt{Super}(u)$.
```

```
function LOCATE($G$, $pf$)
  1. Let $pf = V.P_1.P_2.\cdots.P_k$ and $n_0$ be the node where
     edge $< \mathtt{Super}(r_G), n_0, V >$ is in $E$.
  2. For each $1 \leq i \leq k$, if $< \mathtt{Super}(n_{i-1}), n_i, P_i >$ is not
     in $E$, add node $n_i$:($Type(Cl(n_{i-1}), P_i)$ to $N$ and edge
     $< \mathtt{Super}(n_{i-1}), n_i, P_i >$ to $E$.
  3. Return $n_k$.
```

Figure 2. Auxiliary functions.

```
function CREATE($Q$)
  1. Initialize $\mathbf{G}_Q$ as $< \{r_Q : Q\}, \{\mathtt{Super}(r_Q)\}, \emptyset, r_Q >$.
  2. For each collection or constructor $C$ in the query, add
     a new node $n : C$ to $N_Q$.
  3. For each "$C$ as $V$" in the FROM clause,
     (a) if no node $n : C$ was created in step 2 for $C$, add
         a new node $n : C$ to $N_Q$, and
     (b) add a new edge $< \mathtt{Super}(r_Q), n, V >$ to $E_Q$.
  4. For each "$pf$ as $V$" in the SELECT or FROM clause of
     the query,
     (a) add a new node $n$ : "Object" to $N_Q$,
     (b) add an edge $< \mathtt{Super}(r_Q), n, V_Q >$ to $E_Q$,
     (c) let $n' =$ LOCATE($G_Q$,$pf$), and
     (d) UNION($n, n'$).
  5. For each subquery $S$, CREATE($S$) (note that the sub-
     query is a class and should be rooted at the node cre-
     ated in Step 3 if the subquery is part of a "$C$ as $V$"
     expression). The node for any variable in the subquery
     not defined there should be the one from the parent
     query.
  6. For each subsumption constraint "$pf$ in $C$" in the
     WHERE clause, let $n =$ LOCATE($G_Q, pf$).
     (a) If $C$ is a subquery $S$, MERGE($n$, $r_S$),
     (b) otherwise if $n'$ was created for $C$ in step 2,
         MERGE($n$, $n'$),
     (c) otherwise relabel $n$ with the greatest common
         subclass of $Cl(n)$ and $C$.
  7. For each $pf = C$ for some constant $C$, relabel node
     $n =$ LOCATE($r_Q, pf$) with $C$.
  8. For each join condition $pf_1 = pf_2$:
     UNION( LOCATE($G_Q$, $pf_1$), LOCATE($G_Q$, $pf_2$) ).
  9. For each interface $C$, constraint "key X" in $C$, and
     nodes $n_1 : C_1$ and $n_2 : C_2$ where $C_1 \leq C$ and $C_2 \leq C$,
     if, for each $P$ in $X$ there exists nodes $n_3$ and $n_4$ in $N$
     where $< \mathtt{Super}(n_1), n_3, P >$ and $< \mathtt{Super}(n_2), n_4, P >$
     are in $E$, then if $n_3 = n_4$, MERGE($n_1$,$n_2$), otherwise if
     $\mathtt{Super}(n_3) = \mathtt{Super}(n_4)$, UNION($n_1$,$n_2$).
 10. For each "relationship $C_1$ $P_1$ inverse $C_2$::$P_2$" and
     each node $n_1 : C$ where $C \leq C_1$ and there exists a
     node $n_2$ in $N$ where $< \mathtt{Super}(n_1), n_2, P_1 >$ is in $E$,
     (a) if there exists a node $n_3$ in $N$ where edge
         $< \mathtt{Super}(n_2), n_3, P_2 >$ is in $E$, then MERGE($n_1$,$n_3$),
     (b) otherwise create edge $< \mathtt{Super}(n_2), n_1, P_2 >$.
 11. For each pair of distinct variables $V_1$ and $V_2$ where
     there exists a node $n$ in $N$ with $< r_Q, n, V_1 >$ and
     $< r_Q, n, V_2 >$ in $E$, remove $< r_Q, n, V_2 >$ from $E$.
```

Figure 3. Algorithm for query graph creation.

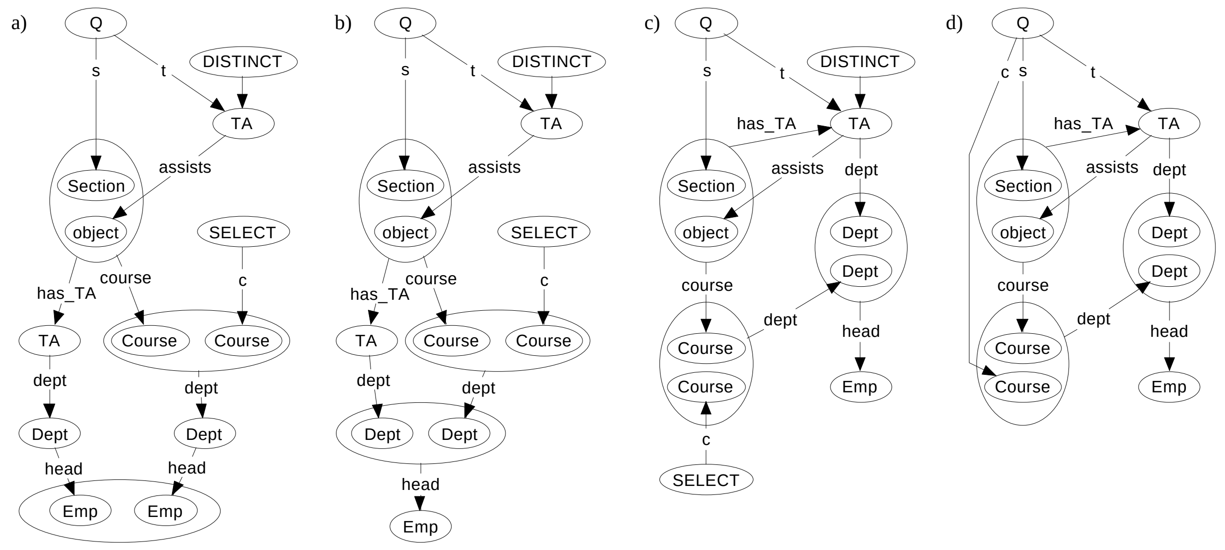

Figure 4. Query graph.

It is clear that CREATE terminates. First no path is created with length longer than the longest path function mentioned in the query, since all paths are created using procedure LOCATE. Second, each recursive call to MERGE removes another vertex, and since the graph is finite, the number of calls to MERGE is finite. Finally, since each modification made to the graph is done to satisfy a construct or constraint, and the number of these is finite, the number of modifications must be finite.

The following theorem establishes the correctness of the construction. This does not illustrate that the most optimal graph is produced, but that the graph is semantically valid. No claim is made as to the optimality of the resulting graph.

Theorem 1 *The query graph G_Q resulting from the call* CREATE*(Q) for a query Q accurately reflects the state of any query object O retrieved by the query; that is, the following conditions hold:*

1. *If pf describes a path from r_Q to some node $n : C$, then the object reached by property path pf from O is in class C.*

2. *If pf_1 and pf_2 describe two paths from r_Q to the same node n, or to two nodes in the same supernode, then the same object must be reached by property paths pf_1 and pf_2 from O.*

Proof: The formal proof is similar to that found in [16] for a more complex data model, and is too long for this paper. It is based on an axiomatization of the types of constraints considered, and illustrates via the axioms how each step of the construction maintains the correctness of the graph. It also establishes the completeness of the axioms and the completeness of the construction; that is, that any constraint that follows from the schema and query is modeled by the resulting graph.

5. Optimizations

Several optimizations are performed during the construction of the query graph. They include join elimination, join simplification, and scan reduction. These are evident in the optimized query below. Other optimizations are also possible based on the graph, as illustrated below.

One such optimization, first introduced by Kim [10], involves the rewriting of correlated, positive existential subqueries as joins, thus avoiding the processing of the query with a nested-loop strategy in favor of a choice of alternate join methods. For example, in the query, since it can be determined that the subquery block can return only one course for each teaching assistant ("assists" is functional), then the subquery can be moved up into the main query.

The process to make this determination involves a marking of supernodes. All supernodes are initially unmarked. The supernodes reached by the variables and constant conditions of the top level query are then marked. In the example in Figure 4, these would be the supernodes reached by the "s" and "t" variables. Each time a node is marked, key constraints are checked to determine if the key of a class is marked, and if so, the supernode of the class is also marked. As well, supernodes reached by single-valued attributes and relationships from a supernode are also marked. This would lead to the marking of the course supernode in the example. Once the supernodes of the variables of a subquery are marked, the subquery can be removed and converted into a join. This results in the removal of the "SELECT" node in graph (d) of Figure 4.

Another optimization possible is the determination of unnecessary duplicate elimination. Consider the example query on the university schema. A typical access plan would require a search or sort to remove duplicate solutions in order to satisfy the DISTINCT clause. It can be determined that for no teaching assistant can there exist more than one section and course satisfying the query. Using the above marking procedure, marking the supernode for "t" would leading to the marking of all other vertices, thus there could be no duplicates. This results in the removal of the "DISTINCT" supernode in graph (d) of Figure 4.

Finally, rewriting the query based on the optimized query graph is a straightforward procedure. A breadth-first traversal through the graph from the root, noting the remaining subqueries, collection, constant, and operator classes, would result in a query. Nodes marked for retrieval would be used in the SELECT clause. Supernodes containing multiple nodes indicate join conditions, and multiple paths reaching a single node indicate the relationships already present in the database.

With this approach, a possible optimized query for the university schema would be

```
SELECT t
FROM   TA as t,
       t.assists as s,
       s.course as c
WHERE  c.dept = t.dept.
```

The resulting query simply involves path traversals and one join condition, a significant reduction in complexity over possible strategies based on the original query.

6. Summary

We have presented a semantic query optimizer for ODMG-93 databases. Our optimization techniques can be applied to a large class of OQL queries, specifically **select-from-where** queries containing operations that can be abstracted as classes. Possible optimizations discussed include subquery to join transformations, join elimination, join simplification, scan reduction, and the removal of unnecessary duplicate elimination.

Using the natural representation of an object-based query as a graph, we plan to extend this work in several ways. First, we intend to consider a larger class of OQL queries, thus permitting more of the flexibility possible in the ODMG-93 proposal. Second, we will include reasoning about Boolean and relational operators by having special operations on their respective classes that are triggered by the presence of certain properties. Third, we intend to propose the increase in the expressiveness of ODL by allowing more general functional dependencies and path functional dependencies, as well as equational constraints in the model [17]. We also intend incorporate more general forms of integrity constraints. Together these will permit further optimizations. Fourth, we will investigate the possibility of using the graph in the process of join order selection using a graph marking scheme and including index classes. Finally we plan to incorporate our techniques for semantic query optimization into an ODMG-93 compliant database system for experimental evaluation.

References

[1] J. P. Ballerini, D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A semantics-driven query optimizer for OODBs. In *International Workshop on Description Logics*, Rome, June 1995.

[2] S. Bergamaschi and C. Sartori. ODB-QOptimizer: A tool for semantic query optimization in OODB. In *International Conference on Data Engineering*, Bermingham, UK, April 1997.

[3] S. Bergamaschi, C. Sartori, D. Beneventano, and M. Vincini. ODB-Tools: A description logics based tool for schema validation and semantic query optimization in object oriented databases. In *Fifth Conference of the Italian Association for Artificial Intelligence*, Rome, 1997.

[4] A. Borgida and G. Weddell. Adding uniqueness constraints to description logics. In *Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases*, pages 85–102, Montreux, Switzerland, December 1997.

[5] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan Kaufmann, San Francisco, CA, 1996.

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

[7] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings of the 13th International Conference on Database Engineering*, pages 444–454, Birmingham, U.K., April 1997.

[8] M. M. Hammer and S. B. Zdonik. Knowledge based query processing. In *Proceeding of the Sixth VLDB Conference*, pages 137–147, Montreal, October 1980.

[9] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing PROLOG front-end to a relational query system. In *ACM SIGMOD International Conference on Management of Data*, pages 296–306, Boston, Massachusetts, June 1984.

[10] W. Kim. On optimizing an SQL-like nexted query. *ACM Transactions on Database Systems*, 7(2):443–469, September 1992.

[11] J. J. King. QUIST–a system for semantic query optimization in relational databases. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 510–517. IEEE Computer Science Press, September 1981.

[12] Y.-W. Lee and S. Yoo. Semantic query optimization for object queries. In *Proceedings of the International Conference on Deductive and Object Oriented Databases*, pages 467–484, Singapore, 1995.

[13] C. G. Nelson and D. C. Oppen. Fast decision algorithms based on union and find. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 114–119, Providence, Rhode Island, November 1977.

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

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

[16] M. F. van Bommel. *Path Constraints for Graph-Based Data Models.* PhD thesis, Department of Computer Science, University of Waterlo, 1996.

[17] M. F. van Bommel and G. E. Weddell. Reasoning about equations and functional dependencies on complex objects. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):455–469, June 1994.

[18] S. Yoon, I. Song, and E. Park. Semantic query processing in object-oriented databases using deductive approach. In *Proceedings of the Conference on Information and Knowledge Management*, pages 150–157, Baltimore, 1995.