Semantic Query Processing in Object-Oriented Database Systems

Jong-Jin Sung and Jong-Tae Park

Department of Electronic Engineering
Kyungpook National University, Taegu, 702-701 Korea

Abstract : A new semantic query processing technique in an object-oriented database system is presented. Issues related to the query processing in object-oriented database systems are discussed. The query processing technique takes advantage of semantic data integrity constraints to generate more efficient access plans. Semantic information related to the target objects of a given query is utilized in a suitable way, either by eliminating the unnecessary part of the query or by transforming the given query into a more efficient form. Heuristics which guide query processor into generating efficient access plans using semantic knowledge are developed.

1 Introduction

Nowadays, there is a growing interest in the application of object-oriented concept to the areas of software development, programming language design and database management system design. The conventional datatypes such as integers, reals and characters impose a restriction on the way data are represented, so that it is less useful. On the other hand, the object-oriented concept tries to express everything in the world as an object, and it provides more powerful expressibility. The representation and processing of complex objects in an DBMS(Database Management System) by using this concept ensures a great advantage. Complex object here is a mixed form of multiple related objects.

Several researches are going on to design DBMS with object-oriented concept, i.e., the design of an object-oriented DBMS. It has a big advantage in storing and processing various kinds of information. But complex objects having another objects as components have a complex form, and relationships among objects are complicated. Effective definition and manipulation of data such as complex objects in an Object-Oriented Database(OODB) are not easy. To solve this problem, many researches are going on.

In this article, a new method utilizing semantic knowledge for the efficient processing of queries in an object-oriented DBMS is presented. It is known that object-oriented DBMS generally has poor performance in its processing speed. Semantic knowledge, guided by a set of heuristics associated with efficient query processing techniques, is taken advantage of to improve the performance.

There have already been a handful of researches on the application of semantic knowledge for the query processing in a relational DBMS[8,14]. On the other hand, in an object-oriented DBMS, there have, within author's knowledge, been no previous attempts to apply the semantic information for the efficient processing of the queries.

The design of Semantic Query Optimization(SQO) system in an OODB system requires several preliminary works: i.e., defining the architectural model of the object-oriented DBMS, defining the format of an object-oriented query and developing its processing methodology based on this model. The scheme for the representation of semantic knowledge in an OODB system should also be developed. Currently, there is no standard object-oriented database model. Accordingly, there is no standard query processing methodology. Access methods with respect to data storage have been investigated in a various ways. Taking account on these situations, we assume no particular object-oriented DBMS model. Rather, the typical features of OODB systems are assumed to be provided in our object-oriented DBMS model. We focus our attention on investigating the applicability of semantic knowledge for the query processing problem in an object-oriented DBMS, and on developing a set of proper heuristics which could guide the query processor to execute the queries more ef-

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi
©World Scientific Publishing Co.

ficiently by using the semantic knowledge.

Shenoy[14] developed simple semantic query optimization system in a relational DBMS. We extend Shenoy's framework to OODB system environment. Furthermore, new inference-guiding heuristics which are applicable to an OODB system are developed. These heuristics make query processing plans more efficient by utilizing semantic knowledge. As in Shenoy's system our SQO system is composed of two stages, i.e., expansion and reduction. Different heuristics are applied in each stage.

In the following section, previous works associated with these problems are introduced. In Section 3, query expression and query processing techniques are described. A formalization of SQO in OODB is given in Section 4. In Section 5, heuristics are proposed and examples are provided to illustrate the application of semantic query processing heuristics in OODB systems. The architecture of semantic query optimization is described in Section 6. Finally, the conclusion and future research areas are followed in Section 7.

2 Previous Works

In this section, previous research works on query processing in OODB systems are described.

Shaw and Zdonik[12] investigate query optimization problem in an OODB system, based on the concept of object equivalence. Two objects are defined to be 0-equal if they have the same object identity[6,12,13]. Other concepts of object equivalence like i-equal are provided to define deep-equality[6,12,13]. They also define query algebra[13] suitable for object-oriented DBMS. They, based on these concepts of equivalence and query algebra, provide a mechanism for the identification of equivalent queries which would produce the same result. They present a query optimization technique which selects the most efficient one among these equivalent queries.

Clutin, et.al., develop the object-oriented DBMS, O_2, and define a query language, Reloop for O_2 which is similar to SQL[4]. Algebra dealing with selection, projection, and Cartesian product operations and a new macro algebra are defined for query processing. However, the methods which could provide data update operations and data definition are not completed. The optimization problem is also not mentioned.

Tanaka and Chang[15] define natural join operation in an object-oriented DBMS. It can be executed between objects of any type, so it is supposed to be well suited to object-oriented DBMS manipulating complex objects.

Kim and others[7] develop indexing techniques for the processing of queries in the object-oriented DBMS, ORION These are described in Section 3.3.

There is another research on indexing technique, used in OPAL which is the object-oriented DBMS language developed for Gemstone DBMS[2].

It is noted that, all these previous research works investigate query processing problem, for their specific object-oriented DBMS architectures and models, without mentioning the utilization of semantic knowledge.

3 Representation and Processing of an OODB Query

In this section, we will use Reloop query language supported by the object-oriented DBMS, O_2, for the representation of our OODB queries. The reason why we use Reloop is that it provides a query representation which is easy to understand. We are not going to rely on any specific query processing system to explain our semantic query optimization technique. We accommodate all the aspects which a typical OODB system would provide. In this section, a few examples of OODB query representation and processing techniques are presented. First, we will show example data of an OODB and then describe our query representation and processing techniques upon them. Data presented here will be used subsequently in Section 5 to describe semantic query processing.

3.1 Example Data

Classes defined here have the form $C_i = [a_1{:}t_1, a_2{:}t_2, \ldots, a_n{:}t_n]$ where i and n are arbitrary integers. C_i is a class name, a_n is an attribute name and t_n represents the type of an attribute here. Now, we are going to define example classes.

```
Univ_Stud = [Record     : Personal,
             Stud_Id    : integer,
             Major      : Department,
             Score      : integer,
             Activity   : {string}]

Personal = [Name       : string,
            Person_Id  : integer,
            Addr       : [City : string, Eve/Str : string,
                            Zip : integer]]

Department = [M_Name : string,
              Detail : string,
              Prof   : string]
```

The main operation we shall use in this article with these data is to reference the values of attributes. Methods and interfaces should therefore be defined to reference each attribute value in classes. In ORION OODB system, a query is represented mainly by messages. These messages are doing the role of interfaces. The predicates of a query here include referencing messages which reference the values of attributes, and relational operator messages $(>, <, =)$ which compare them.

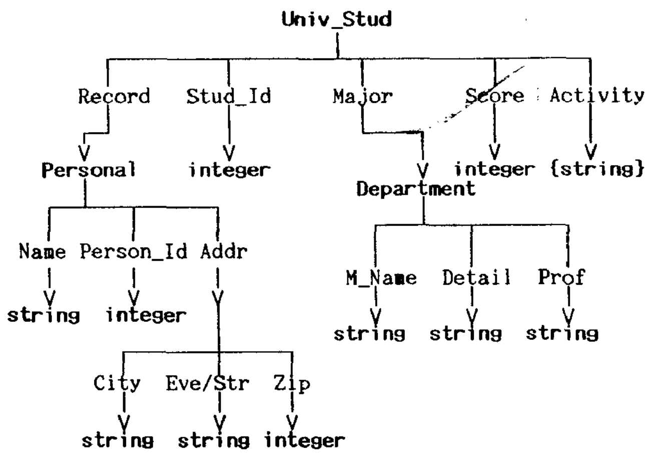


Figure 1: Component hierarchy tree of the example data. The attribute Activity is a set of string type.

The above data are depicted in Figure 1, which gives us an easy look of hierarchical, internal structures of classes and attribute types. In Figure 1, Univ_Stud, Personal, and Department are all classes. Among them the Personal and the Department are nested to the Univ_Stud each as an attribute type. These are called nested classes or branch classes, and the Univ_Stud is a root class. From a nesting relationship, a nesting part is defined as a parent class and a nested part as a child class. Nesting can be made in a cycle and multiple ways. Instances of the class Personal and those of the Department are referenced as values of the attributes Record and Major respectively. This operation of fetching an instance is called object instantiation[1]. In Figure 1 an instantiation is expressed as a down arrow.

3.2 Representation of a Query

Based on the data presented in the previous section, we are going to represent an OODB query in SQL_like form of Reloop which is similar to SQL. The role of SQL_like form of Reloop in this article is limited to the representational aspect of a query. The processing method of a query does not follow that of Reloop.

Example 3.1

Query "Find student id numbers and average scores of the students who are majoring in electronic engineering and living in Taegu."

select [student id number : Stud_Id(u),
    average score:Score(u)]
from u **in** Univ_Stud
where M_Name(Major(u)) = "electronic engineering"
  and City(Addr(Record(u))) = "Taegu"

In SQL_like form, u in the **from** clause represents an instance of the class Univ_Stud. The representation "M_Name(Major(u))" in the predicates of the **where** clause shows the path u.Major.M_Name within the class component hierarchy which can be divided by parentheses ("(", ")"). This kind of representation is also used in the Iris object-oriented DBMS[5]. Predicates in **where** clause represent comparison of indicated attributes with another specified attribute or constant values. These are the conditions requested by the query.

Each OODB system has its own query processing and optimization techniques which suit well with its system structure and characteristics. Some useful query processing techniques like indexing and join are supported by almost all OODB systems to help efficient processing.

[1]In this article, it is also called simply instantiation

3.3 Indexing Techniques in OODB Systems

Indexing in an OODB system reduces query execution time as the indexing in relational database does. According to the research of Kim and others [7] upon indexing techniques, there are some interesting points.

First, there are two ways of traversal within a complex object. One is forward traversal and the other is reverse traversal. Forward traversal is to traverse within a component hierarchy from the root to the terminal nodes in depth-first order. Reverse traversal, on the contrary, does that from the terminal nodes to the root. With these two traversal methods, if we use properly indexed attributes, we can achieve considerable cost reduction in query processing that requires searching in nested classes[2].

One more technique about indexing is class-hierarchy indexing. The conventional indexing is regarded as single-class indexing made within a scope of one class. But the class-hierarchy indexing is made within many classes related to each other by inheritance relationship, i.e., within superclasses and subclasses.

3.4 Join Operation in OODB Systems

In this Section, we give the definition of join operation in an OODB system. The join in an OODB is conceptually similar to that of a relational database. It is used to create relationships between objects each of which belonging to possibly different classes. Our definition of join is similar to Ojoin proposed by Shaw and Zdonik[13]. Suppose that there are collections of objects, S1 and S2. Join(S1, S2, A1, A2, p) = $\{\langle A1{:}s1, A2{:}s2 \rangle \mid s1 \text{ in } S1 \wedge s2 \text{ in } S2 \wedge p(s1,s2)\}$ where p is a predicate defined over objects from S1 and S2. $\langle A1{:}s1, A2{:}s2\rangle$ is a pair in which A1 and A2 are attribute names. A predicate is usually defined using comparison operators($=, >, <$, etc). For equality($=$) operator, the join is referred to as equi-join. Here, equality is defined in terms of many concepts such as the equality of two object values and object identities. The equality concept associated with (shallow)deep-equality[6,10] and

[2]For details, refer to [7].

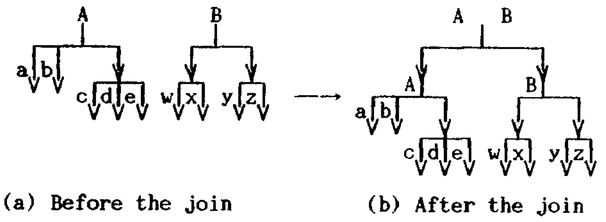


Figure 2: Form of data in join operation.

(i)id-equality[10,12] can also be used.

Join of objects in classes A and B in Figure 2 (a) will result in the form in Figure 2 (b). It is noted that the result of join preserves the internal structures of both classes A and B.

4 A Formalization of SQO in OODB Systems.

Semantic Query Optimization(SQO) is a technique that uses semantic knowledge for query optimization. It uses the semantic knowledge about objects to transform a query into more efficient one generating the same result. In a relational database a lot of researches have been done on it. We introduce a SQO approach to the OODB query optimization. OODB systems have many different ways of query processing. The SQO system can be implemented and inserted as a preprocessing system of OODB query optimization systems.

In this section, the formalization of OODB SQO is given. A database query is to extract portions of data from target objects satisfying given conditions. A target of a query can be either a single class or multiple classes depending on the users' requests. The predicate of a query for SQO here is restricted to be the conjunction of simple comparison operations.

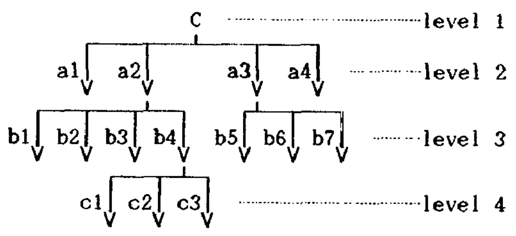
Semantic knowledge used in a SQO is the semantic integrity constraints (SIC) which are expressions of some rules that must be observed from database application viewpoint. These rules are stored in the knowledge base and are applied to the queries in a SQO system for semantic query transformation. Semantic query transformation brings about an alternative form of a query which is semantically equivalent to the original one. The heuristics are used to transform the queries into more efficient forms. SICs, in this article, are restricted to be in the form of Horn clauses and only the implication integrity constraints are used. An example of the SIC using data in Figure 1 is shown below. The semantic knowledge "If someone has name Jone Doe and majors in electronic engineering, then his id number is 8903124." is expressed as "(Name(Record(u)) = "Jone Doe" $\wedge$ M_Name(Major(u)) = "electronic engineering") $\rightarrow$ Stud_Id(u) = 8903124". The front part of the implication mark($\rightarrow$) is called antecedent part and the rear part is called consequent part.

A SIC is said to be *applicable* to an attribute of an OODB if predicate in the antecedent part of the SIC matches up to the attribute of an OODB. For example, for an instance u of a class in an OODB, if the values of attributes Name(Record(u)) and M_Name(Major(u)) are "Jone Doe" and "electronic engineering" respectively, it is said that this SIC is applicable to the attributes Name and M_Name.

We will prove that, in the OODB, SIC can be used for SQO and the results of a query and its transformed one by the application of SIC are the same.

Lemma 1 *Any attribute can, regardless of its level in a component hierarchy tree of an OODB, be used for SIC predicates.*

Proof :



a1	b1(a2)	b2(a2)	b3(a2)	c1(b4(a2))	c2(b4((a2))
c3(b4(a2))	b5(a3)	b6(a3)	b7(a3)	a4	

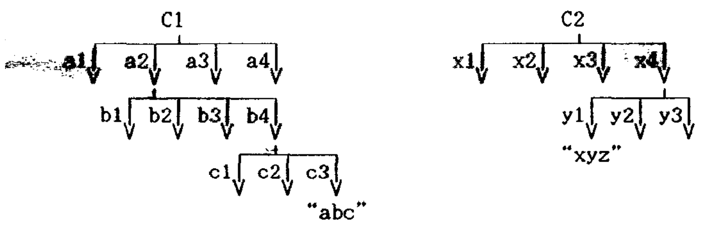
Case 1: c2(b4(a2(C))) = "xyz" $\rightarrow$ a4(C) $\geq$ 5000
Case 2: a1(C) < 40 $\rightarrow$ b5(a3(C)) $\geq$ 600
Case 3: b6(a3(C)) = "Ford" $\rightarrow$ c1(b4(a2(C))) = "car"

Without loss of generality, we assume that a typical component hierarchy tree is structured as shown above. Using the concept of the path, all the attributes in the tree structure can be restructured into elements of a tuple in a relational database. This means that, with the concept of the path, attributes of any branches of a tree can be referenced directly, not affecting other attributes. Therefore as in the case 1, the SIC composed of c2 in the antecedent part and a4 in the consequent part can be applicable to the attributes c2 and a4, where c2 is at a high level and a4 at a low level of the component hierarchy tree. In the case 2, the SIC having an attribute at a low level as an antecedent part and a high level as a consequent part is also applicable. The case 3 shows that no matter where the attributes are positioned in a component hierarchy tree, the attributes can be used for the SIC predicates. ■

For simplicity, two classes are said to be joined when collections of instances of those classes are joined.

Lemma 2 *Any attributes in the classes which will be joined can be used for the SIC predicates.*

Proof :



c3(b4(a2(C1))) = "abc" $\rightarrow$ y1(x4(C2)) = "xyz"

If classes C1 and C2, shown above are joined, then single temporary or permanent class like the one in Figure 2(b) will be formed and this joined class now becomes the target of both the query and the SIC. According to Lemma 1, any attribute in this joined class can be used for SIC predicates. Therefore it can be said that any attribute of two classes to be joined will be used in the SIC predicates. ∎

Lemma 3 *The result of the transformed query by the application of SICs is the same as that of the original one.*

Proof : If the antecedent part of SIC is satisfied for an instance of OODB, then the consequent part is also satisfied. Therefore, if query conditions match the SIC's antecedent part, the SIC's consequent part can also be added to the query conditions without altering the result. ∎

This lemma provides the basis on which we can achieve query optimization, in the OODB, by using SICs as in the relational database.

5 Heuristics for Semantic Query Optimization

In order to transform the query efficiently, semantic knowledge in the knowledge bases are searched and applied to the query transformation. Among the many applicable SICs, some of them transform the query into the less efficient one. Heuristics are used to transform more efficiently. In a relational database, King[8] suggests some useful heuristics. In this article, we suggest new heuristics that are specific to OODB model.

5.1 Heuristics

From the viewpoint of the data structures, the OODB class and the relational database relation scheme are associated with a shape which is a set of tuples. But OODB instances have attributes that are instances of another complex class. In Figure 1, an OODB class forms a tree-shaped hierarchical structure having nested classes in its branches. This structure is far different from the simple tuple structure of a relational database. It requires different traversal mechanisms to find the target attribute values for a query. Taking into account this structural property of an OODB, new heuristics which are different from those for relational database are needed. The heuristics that can be used for a SQO in OODB systems are presented below. Details on these ideas will be explained with the illustrating examples in the subsequent subsection.

H1 Index Introduction
While using the attributes which are in the query, try to use a clustered indexed attribute.

H2 Instantiation Reduction.
Use the attribute which resides in a low level in data component hierarchy.
This heuristic is based on the observation that an object instantiation itself costs much. If all the attributes in the query are in the branches far from the root, unnecessary instantiations(operation) can be saved by using an attribute near to the root. This heuristics can be used in the system that provides forward traversal.

H3 Instantiation Introduction
Use another branch class by instantiation which is not traversed by the original query. This heuristic is useful in the system that permits reverse traversal. In this case, the nested class to be newly used should have clustering link to the parent class and the number of instances are much smaller than that of the parent class.

H4 Instantiation Elimination
Eliminate the redundant instantiation operation.
There are many cases in which, in the execution of a query to find some specific objects, the nested class which is not necessary for the query doesn't have to be fetched.

H5 Join Introduction
Use a clustering link from another class. Even if a class is not included in the query, it can be used by join when it has much smaller number of instances and has clustering link into the class which is constrained by the query.

H6 Scan Reduction
Prior to performing the cross referencing for the join operation, we can reduce the number of inner scans of the join by using additional restrictions.

H7 Join Elimination
Eliminate a class if its attribute has no contribution to the query result.

Among these heuristics, H1, H5, H6 and H7 are similar to the King's heuristics for a relational database[8]. However, due to the structural differences between a relational database and an OODB, there are some differences to be considered. These details will be explained next with the example for each heuristic.

5.2 Examples using Heuristics

First, we show semantic knowledge(SIC's or semantic rules) which will be used in the examples.

R1 "Professor Hong is in physics department."
(Prof(Major(u)) = "Hong" → M_Name(Major(u)) = "physics")

R2 "If the value of i is x, then that of d is y."[3]
(i(g(a(C1_instance))) = x → d(C1_instance) = y)

R3 "Students in electronic engineering department have student id numbers which have 0 in the third digit and 3 in the fourth digit."
(Stud_Id(u) = ??03???[4] → M_Name (Major(u)) = "electronic engineering")

R4 "No student who was born after 1968 has the student id number that starts with 86 which represents the year when the student enters the collage."
(Stud_Id (u) = 86⋆[5] → Person_Id(Record(u)) ≤ 68⋆)

In our queries, the example data in Section 3.1 are used. To explain each heuristic, we show an example query(Q), a semantic rule(R) and an efficiently transformed query(TQ) using the semantic rule.

Example 5.1

This example shows that a clustered indexed attribute is used by appling the first heuristic, *index introduction*.

$Q1$ "Find student id numbers of the students whose advisor is Hong."

```
select Stud_Id(u)
from u in Univ_Stud
where Prof(Major(u)) = "Hong"
```

R1 "Professor Hong is in physics department."
(Prof(Major(u)) = "Hong" → M_Name(Major(u)) = "physics")

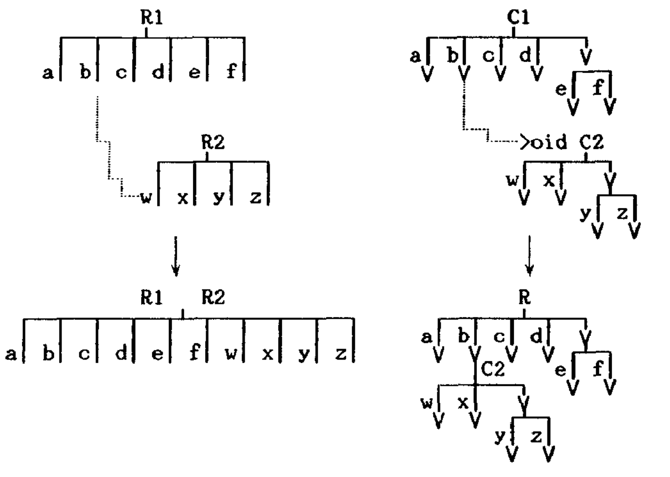
$TQ1$ "Find student id numbers of the student whose major is physics and advisor is Hong."

```
select Stud_Id(u)
from u in Univ_Stud
where M_Name(Major(u)) = "physics" and
        Prof(Major(u)) = "Hong"
```

In this example, the original query $Q1$ searches every instance object of Univ_Stud with the restriction that advisor is Hong. If there is a semantic rule that says professor Hong is in physics department and a clustered index is on the M_Name attributes, we can first check M_Name attribute to select rapidly the students majoring in physics as a newly added restriction. Thus we can reduce the range of valid targets, and within this reduced range, finding the students whose advisor is Hong can be done much faster than the original query processing would do. That is to say, using clustered index, efficient query processing can be achieved.

There is one thing to be noted about appling this heuristic to the OODB. Fortunately, in the above example, the newly introduced indexed attribute M_Name is in the same branch class as the attribute Prof used in the query, i.e., the class Major. But, if the newly added indexed attribute is in a different class far away, then it would not be a good way. For instance, if in the above example, the semantic knowledge to be used is **R1'** : "Students of professor Hong are all living in Taegu city." and it is assumed that a clustered index is on the City attribute. We will then use a new restriction City(Addr(Record(u))) = "Taegu". In this case, in order to use this new restriction, query processing needs to traverse from the root of the Univ_Stud class to the City attribute to check which instances have string "Taegu" in it. In the middle of this process, there occurs instantiation (operation) from the Record attribute to the nested Personal class. This can be very costly, because all the instances in the class Personal should be searched.

Kim and others[7] state that fetching an instance of a child class from its parent class is similar to a join operation in a relational database. Depicted in Figure 3 are relational database join and an OODB instantiation. The class C1 in Figure 3 (b) nests class C2 which is the type of the attribute **b**. As in Figure 3 (b), finding and relating the instances of C2 from **b** is almost the same as equi-join operation between the attribute **b** of C1 and the oid[6] of C2 instances.



(a)Join in a relational DB (b)Instantiation in an OODB

Figure 3: Comparison between relational database join and OODB instantiation.

[3]See Figure 4

[4]The ? character stands for *any single character*.

[5]The ⋆ character stands for *any sequence of n characters*.

[6]Object identifier

As illustrated above, the instantiation in nesting relationship can require large searching time. I/O processing cost for searching data in the secondary storages should not be ignored. Therefore, when using Example 5.1, index introduction, the semantic rule **R1**′, i.e., (Prof(Major(u)) = "Hong" $\rightarrow$ City(Addr(Record(u))) = "Taegu" requires referencing of a new class Personal which is not constrained in the original query. So, the mechanism which could evaluate the amount of I/O cost spent by an additional instantiaton and the time saved by introduction of index should be devised in the SQO system.

Considering this example, it is reasonable for the designer to make index on the oid of all the nested instances or, from the very first in the system itself, to connect every nested class with index like join index of Valduriez[16], in order to make fast instantiation possible. But maintaining those indices is costly.

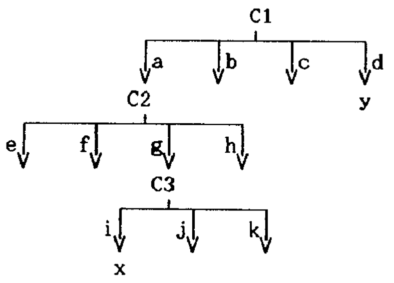


Figure 4: Example data for Q2.

Example 5.2

This example, using the data from Figure 4, shows the heuristic associated with ***instantiation reduction***.

***Q*2** "Find instances of C1 whose attribute i has value x."

> **select** C1_instance
> **from** C1_instance **in** C1
> **where** i(g(a(C1_instance))) = x

R2 "If the value of i is x, then that of d is y"
(i(g(a(C1_instance))) = x $\rightarrow$ d(C1_instance) =y)

***TQ*2** "Find instances of C1 whose attributes d and i have values y and x respectively."

> **select** C1_instance
> **from** C1_instance **in** C1
> **where** d(C1_instance) = y **and**
> i(g(a(C1_instance))) = x

If all the attributes used in the original query are in the high level, i.e., near to the terminal nodes of component hierarchy tree, then by using an another attribute which is in the low level in the tree, we can reduce the number of instantiations.

The newly added restriction in $TQ2$, "d(C1_instance) = y" uses the attribute whose distance from the root is the smallest, i.e., the attribute is positioned in the lowest level in the component hierarchy. It can be applied prior to instantiation of C2 and C3. If this restriction "d(C1_instance) = y" is not satisfied, there's no need to apply the restriction "i(g(a(C1_instance))) = x". Instantiations of C2 and C3 may not be necessary here.

We present below the formulas to estimate the possible cost reduction. Referring to the data in Figure 4, we define that the number of instances of class C_i is N_i and the number of data pages in the storages occupied by C_i is N_i/P where P is the number of instances in one page. It can be shown that I/O cost for the original query $Q2$ is formulated as

$$N_1/P + N_1N_2/P + N_1N_2N_3/P$$

It is noted that the estimation on the cost of the instantiation is based on nested-loop method. After we transform the query using semantic knowledge R2, the cost of $TQ2$ becomes

$$N_1/P + \alpha_1 N_1N_2/P + \alpha_1 N_1N_2N_3/P, \quad \text{where } 0 \leq \alpha_1 \leq 1$$

Here, α_1 is the selectivity determined by the restriction "d(C1_instance) = y" which is applied to the class C1. α_1 has the value between 0 and 1, so $\alpha_1 N_1 \leq N_1$. Therefore, this cost formula shows that the cost of $TQ5$ is reduced by the factor α_1 in comparison with the Q. Since α_1 usually has the value far less than 1, the cost reduction can be great.

Example 5.3

This example shows the ***instantiation introduction*** heuristic. This heuristic can be applied to the system that allows reverse traversal. A child class which is not constrained in the original query can be used when it has clustering link to the parent class which appears in the original query. The child class should have much smaller number of instances than those of the parent. This example is explained using the data of Figure 1.

***Q*3** "Show the average score of the student whose student id number is 8603111."

> **select** Score(u)
> **from** u **in** Univ_Stud
> **where** Stud_Id(u) = 8603111

R3 "Students majoring in electronic engineering have student id numbers which have 0 in the third digit and 3 in the fourth digit."
(Stud_Id(u) = ??03??? $\rightarrow$ M_Name(Major(u)) = "electronic engineering")

***TQ*3** "Show the average score of the student whose major is electronic engineering and whose id number is 8603111."

select Score(u)
from u **in** Univ_Stud
where M_Name(Major(u)) = "electronic engineering"
  and Stud_Id(u) = 8603111

Many Univ_Stud instances can share the same Department instance as the value of the Major attribute. In a real world, this means that under one advisor (professor) there can be many students.
Estimated numbers of the two class instances are

- Univ_Stud - as many as the number of students in our university, 20,000 instances , 4,000 pages of data (5 instances/ a page)
- Department - as many as the number of professors, 500 instances, 100 pages (5 instances/ a page)

The reason why the number of instances of class Department does not match with that of M_Name but that of Prof is that there are more than one professor in each department.

From the data in Figure 1, to search with a restriction "Stud_Id(u) = 8603111", we should read in 4,000 pages of all the Univ_Stud data. But with the knowledge that a student who has student id number of 8603111 is in the electronic engineering major, we can use the class Department by instantiation. The cost estimation is as follows. The size of the class Department is 100 pages and its oid has clustering link to the Major attribute of the class Univ_Stud. At first, 100 pages of Department class data from storages are read in and every electronic engineering students are selected. With the oids of the selected Department instances, Univ_Stud instances of students whose major is electronic engineering can be read in directly by using clustering link. Let's assume that students in electronic engineering major are 2,000 and number of the pages are 400. Then, the reading of the Department class data takes 100 pages and reading through clustering link takes 400 pages, so the total number of pages to be read in are 500. This is 8 times smaller than the original 4,000 pages. Thus we achieved great efficiency by instantiation introduction.

Example 5.4

This example illustrates the heuristic, ***Instantiation elimination***. It is assumed that, the student id number starts with two least digits of the entrance year, and the personal id number starts with those of the birth year.

*Q*4 "Show the student id numbers and the average scores of students whose entrance year is 1986 and birth year is before 1969."

select [student id:Stud_Id(u), average score:Score(u)]
from u **in** Univ_Stud
where Stud_Id(u) = 86⋆ **and**
  Person_Id(Record(u)) ≤ 69⋆

R4 "No student who was born after 1968 has the student id number that starts with 86 which represents the college entrance year."
(Stud_Id (u) = 86⋆ → Person_Id(Record(u)) ≤ 68⋆)

*TQ*4 "Show the student id numbers and the average scores of students whose entrance year is 1986."

select [student id:Stud_Id(u), average score:Score(u)]
from u **in** Univ_Stud
where Stud_Id(u) = 86⋆

Looking at the semantic knowledge R4, we can notice that the restriction "birth year is before 1969" is redundant. Thus the instantiation of the class Personal just for using that redundant restriction is also redundant. So, by eliminating unnecessary instantiation operation we can save a lot of I/O cost.

6 The Architecture for Semantic Query Optimization

6.1 The Architecrture

In this section, we are going to describe the architecture for semantic query optimization. As shown in Shenoy[14], we use two stages of transformations in our SQO system, expansion stage and reduction stage. During the semantic query transformation, expansions of the query are first performed, and then reductions follow next. However, details of operations in each stage as well as heuristics are different from those in Shenoy[14].

As stated before, a specific method adopted by a conventional query optimizer is not an important issue here. We transform the query into the efficient one in the preprocessing phase which is then applied to the conventional query optimization. In Figure 5, the architecture of the SQO system is shown.

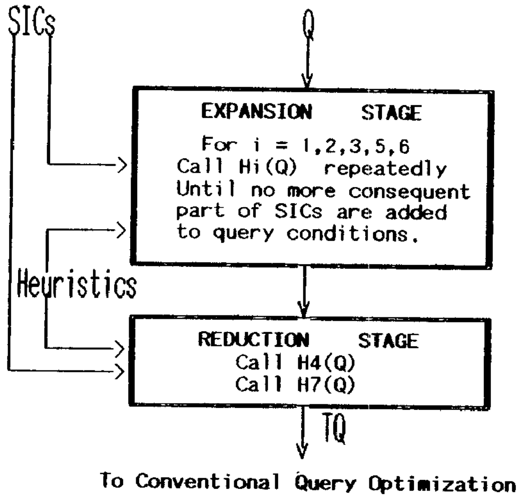


Figure 5: Architecture of the SQO system.

Query expansion implies addition of other restrictions to the original query. Added restrictions are the ones that could lead to the more efficient processing plan when applied. Query reduction means elimination of the redundant restrictions and/or targets that are unnecessary for the desired result. Among the heuristics we have proposed, H1, H2, H3, H5 and H6 contribute to the query expansion stage, i.e., the stage 1 of the preprocessing. The rest of the heuristics are used in the stage 2 where query reduction is performed.

The system works as follows. Each heuristic is implemented into a procedure $H_i(Q)$. In the expansion stage, the procedures $H_1(Q)$, $H_2(Q)$, $H_3(Q)$, $H_5(Q)$ and $H_6(Q)$ are called and executed in a proper order. Callings are repeated until all the applicable SICs are tried out. When no more SICs can be applicable, the reduction stage follows. In the reduction stage $H_4(Q)$ and $H_7(Q)$ are applied. In each heuristic procedure $H_i(Q)$, a query is transformed by the specific way that the heuristic requires. The algorithm of the procedure $H_i(Q)$ is presented below. In the step A, the possibility of appling the heuristic is examined, and if applicable, transformation of the query is performed in the step B. Application of the next $H_i(Q)$ procedure is followed for more transformation. After the application of all the heuristics, the original query Q must have been transformed into TQ. This query TQ produces the same result as the original query Q, while the processing cost of the TQ might be reduced.

```
Procedure  Expand-Query
/*  This  procedure  describes  the  operations  for
expansion stage */
  Begin
    Repeat
      For  i = 1, 2, 3, 5, 6
      Call Hi(Q), in a proper order
    Until no more new consequent part of SIC is
            added to query predicates
  End {Expand-Query}

Procedure Reduce-Query
/*  This  procedure  describes  the  operations  for
reduction stage */
  Begin
    Call H4(Q)
    Call H7(Q)
  End {Reduce-Query}

Procedure Hi(Q)
/*  This  procedure  describes  the  transformation
operations guided by heuristic Hi */
/*  Q:query,  Q.pred:predicate  of  Q,  R:a  set  of
integrity rules, SIC:a rule, SIC.ante:antecedent part
of the SIC, SIC.cons:consequent part of the SIC */
  Begin
A { If Hi is applicable to Q
  ┌ Then
  │   For all SIC ∈ R Do
B {   If SIC.ante matches conditions of Hi and Q.pred
  │   Then transform Q.pred (and Q.target) as Hi
  └         requires
  End {Hi(Q)}
```

6.2 Effectiveness of the SQO

It is proven that the query generated by semantic query optimization is semantically equivalent to the original one. It is also shown that semantic query optimization generates efficient access plans in terms of processing cost. The result of original query Q is expressed as $\textit{Result}(Q)$, and that of the transformed query TQ as $\textit{Result}(TQ)$.

Theorem 1 *The result of the query which is transformed by the application of semantic knowledge under the guidance of the heuristics, is the same as that of the original one. In other words, Q is semantically equivalent to TQ, i.e.,*

$$\textit{Result}(Q) = \textit{Result}(TQ)$$

Proof : Heuristics are used only for selecting semantic knowledge and appling that to a query in such a way that could reduce the cost. In other words, profitable SICs are only applied to a query while SICs themselves are not affected by the heuristics. According to Lemma 3, i.e., the transformation of a query by the application of SICs does not change the result, $\textit{Result}(Q) = \textit{Result}(TQ)$ follows. ∎

Now, we show that the cost of TQ is less than or at least equal to that of Q. It is assumed that the time spent by CPU is much smaller than that of I/O access to the storage, so it could be ignored. We consider only I/O access time for the cost estimation. In Section 5, we showed that the heuristics used in SQO transform a query into the efficient one with lower cost. $\textit{Cost}(Q)$ represents the cost of Q, and $\textit{cost}(TQ)$ does that of TQ.

Theorem 2

$$\textit{Cost}(Q) \geq \textit{Cost}(TQ)$$

Proof : When a query Q is transformed into TQ by the application of the procedure $H_i(Q)$ for some i, it becomes an efficient one having lower cost. At this time, the CPU overhead increased by the semantic query transformation processes is so small that it can be ignored. The consecutive applications of heuristics during the expansion stage would reduce the cost as long as SICs could be applicable. Iterative running of expansion stage would not increase the cost. After the expansion stage, heuristics for reduction stage are applied, which would not increase the cost. Therefore, it is proved that $\textit{Cost}(Q) \geq \textit{Cost}(TQ)$. ∎

For the worst case, where none of the heuristics are applicable, Q and TQ have the same form, and no efficiency is achieved. In this case, even if the system calls a $H_i(Q)$ procedure, in the first step(step A in the algorithm), the query is checked for the validity of this transformation. If none of the SICs are applicable, then the query is not transformed, leaving the rest of the transformation step untouched. So, this amount of the CPU overhead can be ignored.

7 Conclusion

In this article we have shown an efficient query processing method in Object-Oriented Database(OODB) systems. Issues related to OODB systems are described. We propose a new query processing technique in OODB systems which utilizes semantic knowledge.

There have been lots of researches on query optimization in a relational DBMS, utilizing semantic data integrity constraints. However, no attempts have been made to use semantic knowledge for query optimization in an OODB system. In this article, we have first shown that semantic knowledge can be applied to the efficient query processing in an OODB system. We have proposed a semantic query processing method in which queries referencing complex objects can be processed with great efficiency by using the semantic information related to target objects.

Several heuristics are proposed which could guide query processor to proceed, using semantic knowledge, in a more profitable way in terms of processing cost. Semantic query optimization(SQO) system incorporating these heuristics is designed. The effectiveness of SQO system as well as the semantic equivalence of transformed query are proven.

In the future research work, semantic query optimization method taking into account inheritance property of an OODB system will be studied. The concept of object equality proposed by Khoshafian[6] and Masunaga[10] will be utilized in order to explore more diverse and efficient ways of processing queries in an OODB system. The implementation of SQO system is currently under study, which will be accomplished with some additional features.

References

[1] D. Beech, " A Foundation for Evolution from Relational to Object Databases," Proc EDBT, Venice, Italy, 1988.

[2] R. Bretl et. al., "The GemStone Data Management System," **Object-Oriented Concepts, Databases and Applications**, ACM PRESS, Part 3, Ch. 12, 1989.

[3] M. Carey, D. Dewitt and S. Vandenberg, "A Data Model and Query Language for Exodus," Proc. ACM SIGMOD Conf., Chicago, 1988.

[4] S. Cluetin, C. Delobel, C. Lecluse and P. Richard, "Reloop, an Algebra Based Query Language for an Object-Oriented Database system," Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989.

[5] D. H. Fishman et. al., "Overview of the Iris DBMS," **Object-Oriented Concepts, Databases and Applications**, ACM PRESS, Part 3, Ch. 10, 1989.

[6] S. N. Khoshafian and G. P. Copeland, "Object Identity," Proceedings of the Conference on Object-Oriened Programming Systems, Languages and Applications, pp. 406-416, ACM, Sept. 1986.

[7] W. Kim, K. Kim and A. Dale, "Indexing Techniques for Object-Oriented Databases," **Object-Oriented Concepts, Databases and Applications**, ACM PRESS, Part 4, Ch. 15, 1989.

[8] J. J. King, "Quist : A System for Semantic Query Optimization in Relational Databases," VLDB, pp. 510-517, 1981.

[9] C. Lecluse, P. Richard and F. Velez, "O2, An Object-Oriented Data Model," Proceedings of ACM-SIGMOD Conf., Chicago, June 1988.

[10] Y. Masunaga, "Oject Identity, Equality and Relational Concept," Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, pp. 170-187, 1989.

[11] J. T. Park, T. J. Teorey and S. Lafortune, "A knowledge-based approach to multiple query processing," Journal of Data & Knowledge Engineering, Noth-Holand, New York, Vol. 3, pp. 261-284, March 1989.

[12] G. M. Shaw and S. B. Zdonik, "Object-Oriented Queries: Equivalence and Optimization," Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989.

[13] G. M. Shaw and S. B. Zdonik, "A Query Algebra for Object-Oriented Databases," Proceedings of the 6th International Conference on Data Engineering, IEEE, 1990.

[14] S. T. Shenoy and Z. M. Ozsoyoglu, "A System for Semantic Optimization," Proceedings of the ACM SIGMOD Conf., San Francisco, pp. 181-195, May 27-29, 1987.

[15] K. Tanaka and T. S. Chang, "On Natural Join in Object-Oriented Databases," Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, pp. 279-293, 1989.

[16] P. Valduriez, "Join Indices," ACM Transactions on Database Systems, Vol. 12, No. 2, pp 218-246, June 1987.