1-1992

Query Optimization in OODB

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

Hongjun LU
National University of Singapore

Beng Chin OOI
Institute of Systems Science, Singapore

Published in Database Systems for Advanced Applications '91: Proceedings of the Second International Symposium on Database Systems for Advanced Applications, April 2-4, 1991, Tokyo, Japan. pp. 1-10.

Query Processing in OODB

HweeHwa Pang[1], Hongjun Lu[2], BengChin Ooi[1]

[1]Institute of Systems Science
[2]Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511

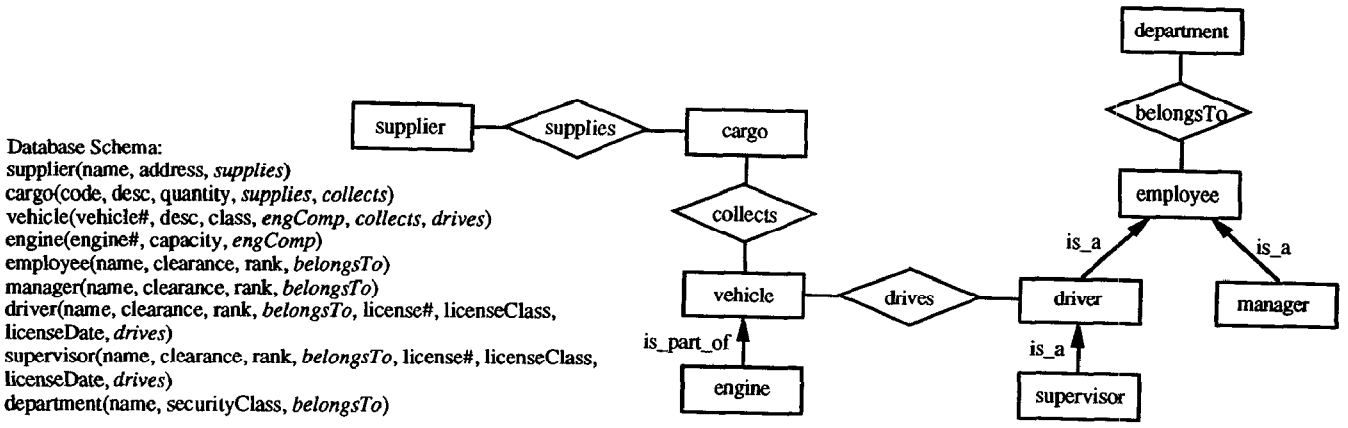
Abstract In object-oriented databases, relationships are generally maintained explicitly. The partial result of a retrieved object can be used to efficiently retrieve related objects. Instead of optimizing joins as in relational database systems, pointer chasing is optimized in object-oriented database systems. Further, semantics inherent in the object-oriented database, like superclass-subclass relationships and composite-component relationships between object classes, must be realised. In this paper, we describe our initial result in query optimization in an object-oriented database system. Semantic query transformation is used to preprocess the query. The semantically optimized query is then translated into a query evaluation plan which comprises method invocations that can be evaluated directly by the system. In the process of query evaluation plan generation, initial results tend to show that a one source query plan is almost optimal. A prototype based on this design has been completed and some results from a simulation study on this prototype are also reported in this paper.

1. Introduction

One of the essential properties of the object-oriented database systems is encapsulation, whereby information in any class is hidden from users and the only way to access the information is through methods provided in that class. Methods are procedures written in a programming language. The computational completeness of the programming language makes methods significantly more expressive than the relational algebra. However, it is still desirable to provide an ad-hoc query interface to object-oriented database management systems (OODBMS) that allows users to pose high level queries in a declarative language, much like SQL in the relational context. Such a language has limited expressive power but, on the other hand, allows users to interact with the OODBMS using the schema and is ideal for non-expert users. A query processor must be provided to translate such high level queries to invocations of methods in the database. Due to the nature of the object-oriented concepts, some features that are absent in relational database systems must now be considered. Before a query can be processed, all its inherent semantics such as the scope of a class must be realised. Since more semantics are captured in OODB, it is desirable that the available semantic knowledge is used to optimize the query. In contrast to relational database systems, joins of two unrelated objects are not that common in OODBMS. Objects in a query are usually related, and their relationships are explicitly maintained. As such, objects can be retrieved using the information stored in the already retrieved objects. In this paper we present our effort in query optimization in an OODBMS.

In [Kor88], Korth attempted to recast a subset of object queries as relational queries so as to enable those queries that are expressible in a relational language to be optimized using standard relational techniques. Osborn [Osb88] investigated query optimization on an object algebra, taking into account the difference between identity and quality of the results of two queries. The EXODUS optimizer generator [GrD87] creates an optimizer from a rule-based description of the query algebra. The algebra can be extended by augmenting an algebra description file. Freytag [Fre87] used transformation rules to generate query evaluation plans (QEPs) from initial query specifications. The transformation rules can easily be modified to accommodate changes or extension to the set of possible QEPs that the optimizer is capable of generating.

This paper reports the design of our query processor. When the query processor receives a high level query, semantics inherent in the relationships that involve object classes in the query are used to modify it to ensure the correctness of the query results. Such relationships include the superclass-subclass and composite-component relationships. The next step is semantic optimization, a technique which uses available semantic knowledge to transform the query into a new query that produces the same answer as the original query, but requires less resources to evaluate. As semantic optimization increases the processing overhead considerably, efficiency of the overall optimizer must be taken into consideration. In the final step, an efficient query evaluation plan is generated, in which the sequence of the retrieval of objects and their access methods are determined. Once the QEP is determined, the elementary operations are then mapped into method invocations which can be evaluated directly by the OODBMS. There may be more than one method to implement an elementary operation, and the method that is most efficient in the query context will be selected. The final sequence of method invocations makes up the executable QEP. Since OODBMS supports extensible data types and hence new methods, it is important that the set of methods available to the optimizer be extensible. The translation process is guided by a search strategy that is of polynomial complexity. Some preliminary studies were carried out on a prototype we developed, and the results show that our design is indeed feasible and reasonably efficient.



Note. Attributes in *italic* are pointers used to implement relationships between object classes.

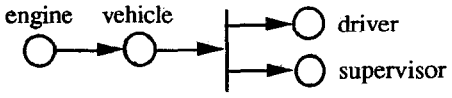
Figure 2.1: An Example Database

The paper is organized as follows. In Section 2, we give an overview of the query processor. Section 3 describes the semantic transformations that are carried out on a query before it is translated into a QEP. Section 4 presents the selection of the cheapest access path and the method for retrieving each object class. The generation of a QEP, based on the access path and methods chosen, is described in Section 5. Implementation and empirical results are presented in Section 6. We conclude in Section 7 with discussion on future work.

2. Preliminary

The concept of object-oriented engenders different requirements in query processing strategies. Firstly, instances of the subclasses of an object class are also instances of it. When instances of the class are requested, implicitly instances of the subclasses are also required (unless explicitly stated otherwise). Assuming that instances of the subclasses are not duplicated in the object class itself [Ba*87], the query processor has to explicitly add to queries subclasses of those object classes requested.

The second issue arises from the fact that there might be composite-component relationships between object classes involved in a query. To be semantically correct, such relationships must be taken into account during query evaluation.



Query: Get the engine# of each vehicle driven by class 3 drivers.

Figure 2.2: A Query Access Path

Consider the database schema shown in Figure 2.1, which is used in all the examples in this paper. In this example, we assume that a subclass (e.g. manager and driver) inherits all the attributes, including the relationships, of its superclass (e.g. employee). Unlike relational databases in which each relationship is usually represented by a separate relation, relationships in object-oriented databases are typically represented by pointers. Hence each object instance in a class participating in a relationship contains an associated set of pointers. This set of pointers stores the object *ids* of instances in other participating classes associated to it by this relationship (e.g. *belongsTo*). Some fields might also be attached to the set of pointers to represent attributes of the relationship.

A query in our system involves object classes that are linked by relationships. Retrieval begins with a chosen object class. All other object classes can only be retrieved through pointers in those classes that have already been accessed. Given a query, there are many ways to execute the query. Each way gives rise to an access path. Figure 2.2 shows an access path for a query involving some object classes (represented by circles) and relationships (represented by solid lines between the object classes) in the schema in Figure 2.1. The arrows denote the sequence in which object classes are accessed, and parallel branches starting from a vertical bar denotes parallel retrieval of these branches. In each access path diagram, there is at least one object class with no incoming arc. Such object classes are known as *sources* and the remaining object classes are *intermediates*. Object class *engine* is a source and the rest are intermediates. Object classes with no outgoing arcs are further known as *sinks*. Object classes *driver* and *supervisor* are sinks. Furthermore, if two or more branches merge at an object class, that class is known as a *sync*.

All the inputs to the optimizer are given in the following conceptual format.

(SELECT {projectList} {joinPredicateList}
{selectPredicateList}
{relationshipList} {classList})

The different parts of the query describe the attributes required, the join predicates and selection predicates on object classes, the relationships between the object classes involved, and the object classes to be accessed, respectively. A join predicate is one that involves attributes from more than one object class, while a selection predicate is one that involves attributes from only one class. To avoid ambiguity in the query path, the relationships joining the object classes involved in a query have to be explicitly identified. Though there are some redundancies, this representation is nevertheless chosen to improve the clarity of our illustrations. In what follows, restriction predicates are used to refer to join predicates and selection predicates collectively.

A declarative query is mapped to a series of elementary operations and commands. The following is a list of elementary operations:

- (SCAN class {attributeList} {joinPredicateList} {selectPredicateList} resultClass)
  Denotes an object class scan, with the attributes required (in the attributeList) from it and the join and selection predicates that have to be satisfied. *resultClass* is a virtual class (view) that contains the instances that form the results. Before concatenating the output to the partial result in *resultClass*, the join predicates in *joinPredicateList* are evaluated and instances that do not satisfy all the join predicates are eliminated. The access path and retrieval method are not specified.

- (FORK {$list_1$} .. {$list_n$})
  This is a keyword which activates *n* parallel threads to retrieve the *n* lists of object classes.

- (LINK {source} {sink} {joinPredicateList} {selectPredicateList} {relationshipList} {classList} resultClass)
  Denotes retrieval of an arbitrary number of object classes (given in *classList*) without specifying their order, except for the *source* and possibly the *sink*. The join predicates (in *joinPredicateList*) and selection predicates (in *selectPredicateList*) that the results must satisfy, and the relationships between object classes (in *relationshipList*) are also given. The output is to be placed in *resultClass*.

There must be at least one implementation (method) for each elementary operation. The most efficient method in the query context will be chosen, and the chosen method will replace the elementary operation. The final sequence of methods makes up the QEP and can be executed by the DBMS directly. As the choice of the most efficient methods is system-dependent, we will only briefly explain how this is done, using the elementary retrieval operation, *scan*, as illustration. The main emphasis of this paper, however, is on the transformation from declarative queries to elementary operations.

Each class definition includes the various methods that can be used to retrieve the object instances in it. In addition, a function must be provided for each method that gives its estimated execution cost. For example, the definition of class A might be

```
class A {
    attributes:
        a1: ..;
            :
        am: ..;
    class methods:
        scan(A, ...): scan1(A, ..);
            :
        scan(A, ...): scann(A, ..);
        cost1(A, sf, ..): ..;
            :
        costn(A, sf, ..): ..;
        scanCost(A, sf, ..): min_{i=1}^{n} costi(A, sf, ..);
}
```

In the definition, $scan_1$ and $scan_n$ are methods that implement the elementary retrieving operation *scan* using different searching techniques. To estimate the cost of retrieving any object class, all the cost estimation functions in that class are evaluated, and the lowest cost is returned. The method chosen to retrieve the class is the one associated to the lowest estimated cost. Note that

- Each cost function $cost_i$ associated with method $scan_i$ requires a compulsory argument *sf*, which represents the selectivity factor. The estimation of selectivity factors will be discussed in Section 4.

- Object classes in an OODB are related by superclass-subclass relationships, with the *system* class being the superclass of all other classes. Hence the object classes form a *directed acyclic graph* (DAG). If a particular retrieval method $scan_i$ is shared by all the object classes in a subgraph in the DAG, that method can be factored out and placed in the root of that subgraph. This way all the classes in the subgraph will inherit the method. For example, an indexed retrieval method and a sequential retrieval method, together with their associated cost functions, might be provided in the root class (the root of the object class DAG), and all the other classes can inherit these methods by default.

The optimizer, outlined in Figure 2.3, consists of four modules. Semantic preprocessing transforms queries to reflect the semantics inherent in the object-oriented database, eg. the superclass-subclass relationships and the composite-component relationships between object classes. This module is essential for the correctness of the query results. Semantic optimization uses semantic constraints to modify the queries so that they can be evaluated more efficiently, without changing the semantics of the queries. Access path selection decides on the order to retrieve object classes and the method to use to access each object class. QEP generation produces series of method invocations to execute the queries.

3. Semantic Transformations

3.1 Semantic Preprocessing

The semantic preprocessor uses four preprocessing rules to make semantics inherent in the object-oriented context explicit. This step is necessary for the correctness of the query results.

- Adding Subclasses
  Since instances of an object class include those that belong to its subclasses, subclasses (and in turn their subclasses) of object classes involved in the query must be added to it. Note that the subclasses inherit the selection predicates and join predicates on the object class. The subclasses of each object class can be identified from the schema and is available from the data dictionary during runtime. For example,

  (SELECT {driver.name, vehicle.vehicle#} {} {} {drives} {driver, vehicle})

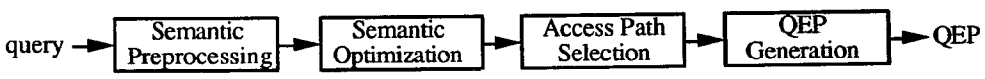


Figure 2.3: Overview of the Optimizer

becomes

(SELECT {{driver.name, supervisor.name}, vehicle.vehicle#} {} {}
{drives} {driver, supervisor, vehicle})

which means *driver.name* and *supervisor.name* will be "unioned" together to form one attribute in the results.

- Identifying Existing Subclasses
Among the classes listed in the query, some might be subclasses of others. For example,

(SELECT {driver.name, supervisor.name, vehicle.vehicle#} {} {}
{drives} {driver, supervisor, vehicle})

As it is, the results would be wrong since *driver.name* and *supervisor.name* get "multiplied" together. A transformation is required to group together the classes related by superclass-subclass relationships:

(SELECT {{driver.name, supervisor.name}, vehicle.vehicle#} {} {}
{drives} {driver, supervisor, vehicle})

- Deep Retrieval
When a composite class is to be retrieved, it is possible to automatically retrieve some of its component classes. Whether this rule is active depends on the value of a system variable. The depth of retrieval, i.e. number of layers of components thus retrieved, can either be computed dynamically based on available display space, or be some pre-determined values that might vary from class to class, or be given as part of the query. For example,

(SELECT {driver.name, vehicle.*} {} {}
{drives} {driver, vehicle})

is transformed to

(SELECT {driver.name, vehicle.*, engine.*} {}
{}{drives, engComp} {driver, vehicle, engine})

There are different ways to present the results. If relations are used, a component class is shown as a "nested" relation in the relation that represents its composite class. Note that we represent a nested object such as vehicle.engine as two objects with a relationship.

- Identifying Existing Components
Among the classes to be retrieved, some might be components of others but these composite-component relationships are not stated in the query. For example,

(SELECT {driver.name, vehicle.vehicle#, engine.capacity} {} {}
{drives} {driver, vehicle, engine})

The results of this query would not be (logically) correct since *engine* and *vehicle* simply get "multiplied" together, and would not show the engine capacity of each vehicle. This preprocessing rule makes the composite-component relationships explicit:

(SELECT {driver.name, vehicle.vehicle#, engine.capacity} {} {}
{drives, engComp} {driver, vehicle, engine})

3.2 Semantic Optimization

Semantic query optimization was first proposed by King [Kin81] and by Hammer and Zdonik [HaZ80]. Three semantic optimization rules - restriction elimination, restriction introduction and class elimination (the equivalence of relation elimination), have been incorporated into our optimizer. In [PLO90], we presented an efficient algorithm to implement these optimization rules and we briefly discuss it here. In our approach, all possible transformations are tentatively applied to the query. Instead of physically modifying the query, the transformation process classifies the predicates in the query into *imperative*, *optional* or *redundant*. An *imperative* predicate is one whose removal will affect the final results. An *optional* predicate is one that, though its inclusion or exclusion does not affect the final results, might affect the execution efficiency of the query by enabling us to make use of indices, or by cutting down the number of instances returned from some object classes and thereby reducing the size of intermediate results. A *redundant* predicate affects neither the final results nor the execution efficiency. At the end of the transformation process, all the *imperative* predicates are retained while the *redundant* predicates are eliminated. Whether an *optional* predicate should be retained depends on the estimated cost savings it can help bring about and the cost of evaluating it. This effectively means the task of choosing the beneficial transformations is delayed until all the possible transformations have been considered. Using this approach, previous transformations do not preclude other transformations and the order of transformations is immaterial. Hence the algorithm is of polynomial complexity. Another advantage of this approach is there is no need to check the profitabilities of all the transformations, eg. those transformations that re-classify predicates to *redundant* should always be carried out. Another issue addressed in the paper is the grouping of semantic constraints to reduce the overhead of retrieving constraints and checking whether each constraint is relevant to the current query. Constraints are grouped according to the object classes they reference. When optimizing a query, only selected groups of constraints are considered. At the same time, constraints are also classified as *intra-* or *inter-class*, depending on the number of object classes each references. This classification is exploited during optimization to reduce the transformation overhead.

4. Access Path Selection

In this section, we use the following notations. $NCARD(O_i)$ is the number of object instances in class O_i, and is available in the data dictionary. D_i refers to the number of instances in O_i that have to be retrieved from the storage device. After the retrieval, the instances might be tested against some selection or join predicates and only those instances which satisfy the predicates are returned. P_i measures the number of such returned instances. Hence $P_i \leq D_i$.

4.1 Estimation of Selectivity Factors

Before the optimizer can calculate the costs of different access paths, it first has to estimate the selectivity factor for each attribute and object class. The selectivity factor is defined as the number of the qualified instances over the total instances. It will be used to estimate the number of instances that need to be

retrieved and the number of instances that will be returned from each object class. When accessing attributes, there are two possible categories of selections.

- *Selection before retrieval*: This is the selection on an attribute via an index such that only instances that satisfy the restriction predicates on the indexed attribute are retrieved. We denote the selectivity factor before retrieval of object O_i based on attribute A_j as SFB_{ij}. When the attribute is not important to the discussion, only SFB_i is used.

- *Selection after retrieval*: This is the selection on attributes which are not part of the index. Whether an instance satisfies the restriction predicates on the attribute can only be determined after the instance has been retrieved. We denote the selectivity factor after retrieval of object O_i based on attribute A_j as SFA_{ij}.

Estimation of selectivity factors before retrieval for sources and selectivity factors after retrieval has been studied by various researchers. The System R optimizer [SAC79] used simple statistics, such as the minimum and maximum values in a given attribute, to estimate selectivity factors of attributes. This scheme produces good selectivity estimates only if the attribute values are uniformly distributed. Equi-depth histograms have been proposed as an alternative for estimating selectivity of attributes whose values are not uniformly distributed [ShC84, MuD88]. In our current prototype, we use a simple scheme that is similar to System R to minimize overhead. With this scheme, the selectivity factor before retrieval, SFB_{ij}, for an intermediate O_i accessed *directly* via pointers from object classes O_1, O_2, .., and O_n is defined as:

$$SF_{B_{ij}} = \prod_{k=1}^{n} SF_{A_{kj}}$$

Selectivity factors are used to estimate the number of object instances to be retrieved, and the number of instances that will be returned. Assuming restriction predicates on different attributes are independent (indeed this condition is satisfied since redundant predicates are removed), the selectivity factor before/after retrieval of an object class O_i, SFB_i/SFA_i, is the product of the selectivity factors before/after retrieval of the attributes in that class. From now on we shall refer only to selectivity factors of object classes. Hence

$$D_i = SFB_i \times NCARD(O_i)$$
$$P_i = SFA_i \times NCARD(O_i)$$

4.2 Determination of Access Path

We now present the algorithm for access path determination. This has been demonstrated to be a hard combinatorial problem [SwG88] and algorithms like simulated annealing and techniques based on local search have been developed [PaS82] to solve it. In our current implementation, only those access paths with one source are considered. This reduces the access path determination problem to the problem of finding the reachability between any two nodes. Hence the algorithm is of polynomial complexity. Our justification for using this simplified approach is, since our selectivity factors are only estimates of the actual values, it is futile to attempt very detailed computations as the errors from estimating the selectivity factors would get magnified.

Any object class in the query can be used as the source in the access path. We define a selectivity matrix Ψ which contains selectivity factor on the object id in each object class as follows. For a query involving m object classes,

$$\Psi = (\psi_{ij}), \qquad 1 \le i,j \le m$$

$$\psi_{ij} = \begin{cases} 1 & \text{if } i = j \\ SFA_i & \text{if } O_i \text{ and } O_j \text{ are connected by an arc} \\ 0 & \text{otherwise} \end{cases}$$

When evaluating a query, the selection on object classes accessed previously might reduce the number of instances in the current object class that need to be retrieved. Assuming an access path of O_1, .., O_m. The selectivity factors on O_1, .., O_{i-1} need to be propagated to O_i. Hence we define Ψ^{2n}, $n \ge 1$ as

$$\Psi^{2n} = (\psi_{ij}^{2n})$$

where

$$\psi_{ij}^{2n} = \min_{1 \le k \le m} \psi_{ik}^{n} \times \psi_{kj}^{n}$$

The selectivity matrix Ψ^n can be interpreted as follows. If we start from O_i and access intermediates via object ids, the selectivity factor before retrieval of an object class O_j linked by at most n relationships to O_i is ψ_{ij}^{n}. Hence for a query involving m object classes, we need to compute Ψ^n, $n \ge m\text{-}1$ to know the selectivity factor before retrieval of any object class given an arbitrary source. The number of matrix multiplications is thus $p = \lceil \log_2 m\text{-}1 \rceil$, and the total number of primary operations needed is $m^3 \times \lceil \log_2 m\text{-}1 \rceil$.

Having computed the selectivity matrix Ψ^p, we compute the cost matrix Φ as follows.

$$\Phi = (\phi_{ij})$$

where

$$\phi_{ij} = scanCost(j, \psi_{ij}^{p}, ...)$$

Every $\phi_{ij} \in \Phi$ gives us the minimum cost of retrieving object class O_j if we use O_i as the source. Hence we choose as source O_i such that the total cost of retrieving all the object classes is the lowest.

$$Cost_{query} = \min_{1 \le i \le m} \sum_{j=1}^{m} \phi_{ij} \qquad \phi_{ij} \in \Phi$$

We use as access path the one from which we derive the lowest retrieval cost using the chosen source. Figure 2.2 shows a possible access path with *engine* as the source.

The most expensive part of our access path selection algorithm is the computation of the selectivity matrix. Hence the time

complexity of the algorithm is $o(m^3 \log_2 m)$, where m is the number of object classes in the query. An access path chosen by our algorithm is essentially a tree with the source as root. We prefer such paths because they do not contain syncs, so each branch of the tree can be retrieved in parallel without having to wait for another branch at a sync.

5. QEP Generation

After the access path has been determined, the QEP can be generated from the query using QEP generation rules. We define predicate $\Delta(j)$ such that $\Delta(j)$ evaluates to true if all the attributes involved in join predicate j have already been retrieved; otherwise $\Delta(j)$ is false. In addition, we define function Γ as follows:

- $\Gamma(\rho)$ returns the set of object classes referenced by ρ, where ρ is a restriction predicate, i.e. join predicate or selection predicate.
- $\Gamma(\alpha)$ returns the set of object classes that contain attribute α.

The query is first converted to algebraic form using the following rule to ease the generation process.

(SELECT e1 e2 e3 e4 e5) →
(REQUIRE e1 (LINK {t_0} {} e2 e3 e4 e5
resultClass))

Note that a dummy source, t_0, has been introduced into the term LINK. This is to facilitate generation of QEP and will be removed at the end of the process. *resultClass* will contain the results of the query and is initialized to an empty class.

Example 5.1: The algebraic form for the query "get the names of those drivers whose license classifications equal the classifications of the vehicles they are driving, and whose vehicles have engine capacity greater than 1000cc".

(SELECT {{driver.name, supervisor.name}}
{driver.licenseClass = vehicle.class,
supervisor.licenseClass = vehicle.class}
{engine.capacity > 1000}
{drives, engComp}
{driver, supervisor, vehicle, engine})

is transformed to

(REQUIRE {{driver.name, supervisor.name}}
(LINK {t_0} {}
{driver.licenseClass = vehicle.class,
supervisor.licenseClass = vehicle.class}
{engine.capacity > 1000}
{drives, engComp}
{driver, supervisor, vehicle, engine}
resultClass))

The generation of QEPs can be divided into three steps. In the first step, the optimizer transforms the query to reflect the access path chosen. The notations for the QEP generation rules used in this step are listed in Table 5.1.

- When the sink in the access path has been reached, *Rule 5.1* terminates the step by replacing the LINK operation with a null list.

Rule 5.1: (LINK F L J S R T resultClass) $\rightarrow^{C1}$

where C1 = (F = L).

- The following two rules generate a SCAN operation for the first object class in the access path, and the selection predicates on this object class are moved into the SCAN operation. The object class that is linked to this first object class will in turn be the first among the remaining classes. The difference between the two rules is the treatment of join predicates that involve the first object class. If a join predicate also involves another object class which has yet to be retrieved, *Rule 5.2* keeps it with the LINK operation; otherwise *Rule 5.3* is used to move it into the SCAN operation.

Rule 5.2: (LINK {t_f} L J∪{j_f} S∪{s_f} R∪{r}
T∪{t_1} resultClass $\rightarrow^{C2 \wedge \text{not}(\Delta(j_f))}$
((SCAN t_f {} {} {s_f} resultClass)
(LINK {t_1} L J∪{j_f, r ∈ t_f.r} S R T
resultClass))

Rule 5.3: (LINK {t_f} L J∪{j_f} S∪{s_f} R∪{r}
T∪{t_1} resultClass $\rightarrow^{C2 \wedge \Delta(j_f)}$
((SCAN t_f {} {j_f} {s_f} resultClass)

Table 5.1: Notations for QEP Generation Rules

Notation	Meaning
F	a set of sources, F = {} or {t_0}
L	a set of sinks, L = {} or {t_1}
J	a list of join predicates
S	a list of selection predicates
R	a list of relationships
T	a list of object classes
Ω	a set of values retrieved from attribute r in the previous SCAN
s_f	the selection predicates on object class t_f
j_f	the join predicates that refer to object class t_f

(LINK $\{t_1\}$ L J$\cup\{r \in t_f.r\}$ S R T resultClass))

where C2 = (r links t_f and t_1)

- The next two rules generate a SCAN operation for the first object class in the access path and divides the remaining object classes (together with their selection predicates and relationships) into sub-blocks that are evaluated in parallel. Each of these sub-blocks contains an object class that is linked to the first object class, and which will in turn be the first in the sub-block. The difference between *Rule 5.4* and *Rule 5.5* is the treatment of join predicates that involve the first object class, and is similar to the difference between *Rule 5.2* and *Rule 5.3*.

Rule 5.4: (LINK $\{t_f\}$ L J$\cup\{j_f\}$ S$\cup\{s_f\}$ R$\cup\{r_1,..,r_n\}$ T$\cup\{t_1,..,t_n\}$ resultClass $\rightarrow^{C3 \wedge \text{not}(\Delta(j_f))}$
((SCAN t_f { } { } $\{s_f\}$ resultClass)
(FORK
(LINK $\{t_1\}$ L $J_1\cup\{j_f, r_1 \in t_f.r_1\}$ S_1 R_1 T_1 resultClass)
...
(LINK $\{t_n\}$ L $J_n\cup\{j_f, r_n \in t_f.r_n\}$ S_n R_n T_n resultClass)))

Rule 5.5: (LINK $\{t_f\}$ L J$\cup\{j_f\}$ S$\cup\{s_f\}$ R$\cup\{r_1,..,r_n\}$ T$\cup\{t_1,..,t_n\}$ resultClass $\rightarrow^{C3 \wedge \Delta(j_f)}$
((SCAN t_f { } $\{j_f\}$ $\{s_f\}$ resultClass)
(FORK
(LINK $\{t_1\}$ L $J_1\cup\{r_1 \in t_f.r_1\}$ S_1 R_1 T_1 resultClass)
...
(LINK $\{t_n\}$ L $J_n\cup\{r_n \in t_f.r_n\}$ S_n R_n T_n resultClass)))

where C3 = $(\overset{n}{\underset{i=1}{\forall}}\, j \in J_i, t \in T_i\cup\{t_i\}$ for some $t \in \Gamma(j))$
$\wedge\, (\overset{n}{\underset{i=1}{\forall}}\, s \in S_i, \Gamma(s) = t$ where $t \in T_i\cup\{t_i\})$
$\wedge\, (\overset{n}{\underset{i+1}{\cup}} J_i = J) \wedge (\overset{n}{\underset{i=1}{\cup}} S_i = S) \wedge (\overset{n}{\underset{i=1}{\cup}} R_i = R) \wedge (\overset{n}{\underset{i=1}{\cup}} T_i = T)$
$\wedge\, (\overset{n}{\underset{i=1}{\forall}}\, r_i$ links t_f and $t_i))$

- The last two rules are similar to *Rule 5.4* and *Rule 5.5*, except there is a sync at the end of the parallel sub-blocks.

Rule 5.6: (LINK $\{t_f\}$ L J$\cup\{j_f\}$ S$\cup\{s_f\}$ R$\cup\{r_1,..,r_n\}$ T$\cup\{t_1,..,t_n,t_{n+1}\}$ resultClass $\rightarrow^{C4 \wedge \text{not}(\Delta(j_f))}$
((SCAN t_f { } { } $\{s_f\}$ resultClass)
(FORK
(LINK $\{t_1\}$ $\{t_{n+1}\}$ $J_1\cup\{j_f, r_1 \in t_f.r_1\}$ S_1 R_1 T_1 resultClass)
...
(LINK $\{t_n\}$ $\{t_{n+1}\}$ $J_n\cup\{j_f, r_n \in t_f.r_n\}$ S_n R_n T_n resultClass))
(LINK $\{t_{n+1}\}$ L $J_{n+1}\cup\{j_f\}$ S_{n+1} R_{n+1} T_{n+1} resultClass))

Rule 5.7: (LINK $\{t_f\}$ L J$\cup\{j_f\}$ S$\cup\{s_f\}$ R$\cup\{r_1,..,r_n\}$ T$\cup\{t_1,..,t_n,t_{n+1}\}$ resultClass $\rightarrow^{C4 \wedge \Delta(j_f)}$
((SCAN t_f { } $\{j_f\}$ $\{s_f\}$ resultClass)
(FORK
(LINK $\{t_1\}$ $\{t_{n+1}\}$ $J_1\cup\{r_1 \in t_f.r_1\}$ S_1 R_1 T_1 resultClass)
...
(LINK $\{t_n\}$ $\{t_{n+1}\}$ $J_n\cup\{r_n \in t_f.r_n\}$ S_n R_n T_n resultClass))
(LINK $\{t_{n+1}\}$ L J_{n+1} S_{n+1} R_{n+1} T_{n+1} resultClass))

where C4 = $(\overset{n+1}{\underset{i=1}{\forall}}\, j \in J_i, t \in T_i\cup\{t_i\}$ for some $t \in \Gamma(j))$
$\wedge\, (\overset{n+1}{\underset{i=1}{\forall}}\, s \in S_i, \Gamma(s) = t$ where $t \in T_i\cup\{t_i\})$
$\wedge\, (\overset{n+1}{\underset{i+1}{\cup}} J_i = J) \wedge (\overset{n+1}{\underset{i=1}{\cup}} S_i = S) \wedge (\overset{n+1}{\underset{i=1}{\cup}} R_i = R) \wedge (\overset{n+1}{\underset{i=1}{\cup}} T_i = T)$
$\wedge\, (\overset{n}{\underset{i=1}{\forall}}\, r_i$ links t_f and $t_i))$

In the second step of QEP generation, each attribute specified in REQUIRE is moved into the SCAN operations for the object classes that contain the attribute, and finally the term REQUIRE is removed.

(REQUIRE ( ) (.. ..)) $\rightarrow$ (.. ..)

(REQUIRE {..α..} (..(SCAN t (.. ..) J S resultClass)..)) $\rightarrow^{C5}$
(REQUIRE {.. ..} (..(SCAN t (..α..) J S resultClass)..))

where C5 = ($t \in \Gamma(\alpha)$).

Example 5.2: For the query in Example 5.1, if the access path in Figure 3.2 is chosen, the corresponding QEP would be

((SCAN engine { } { } {capacity > 1000} resultClass)
(SCAN vehicle { } {engComp $\in$ resultClass.engine.engComp} { } resultClass)
(FORK
(SCAN driver {name}
{licenseClass = vehicle.class, drives $\in$ resultClass.vehicle.drives}
{ } resultClass)
(SCAN supervisor {name}
{licenseClass = vehicle.class, drives $\in$ resultClass.vehicle.drives}
{ } resultClass)))

Table 6.1 Database Sizes

	DB1	DB2	DB3	DB4
Number of Object Classes	5	5	5	5
Average Class Cardinality	52	104	208	208
Number of Relationships	6	6	6	6
Average Relationship Cardinality	77	154	308	616

Strictly speaking, attributes (including those that represent relationships) in an object class that are involved in selection or join predicates should appear in the attribute list of the corresponding SCAN operation, so that they will be in *resultClass* for evaluation of the predicates. Furthermore, if such attributes are not among those required in the final results, they should be dropped from *resultClass* immediately after the relevant predicates have been evaluated. However, we leave out such attributes to simplify the presentation of our QEP generation rules and examples.

While determining the access path, the optimizer had to decide on the cheapest method $scan_i$ to retrieve each object class i. In the last step, this is reflected in the QEP by changing each SCAN operation to $scan_i$.

6. Implementation and Empirical Analysis

Our prototype query optimizer was implemented on a SUN-3/160 workstation. As different parts of our object-oriented database system (OODB) are currently being developed, the effectiveness of our optimizer was evaluated as follows. A sample database whose schema is shown in Figure 2.1 was built. All possible paths in this schema were identified, where a path consists of a series of interconnecting object classes and relationships, and no object class or relationship appears more than once. A "sensible" query was formulated for each such path and thus a set of queries was generated. From this set of queries, 40 test queries were randomly chosen and sent to the optimizer. After optimization, each pair of original and optimized query was sent to a DBMS to be executed. A relational DBMS, Oracle 5.0 running on an IBM RT, was used to simulate the cost ratios of the optimized and original queries. Although relational DBMS and OODB might use different access mechanisms, we believe an improvement in execution efficiency in the relational context would most likely indicate a similar improvement in OODB context.

In our experiments, each object class had an average of 3 semantic constraints attached to it. The database was populated with randomly generated data, the amount of which was varied to test the performance of the optimizer under different database sizes. Here we present the performance under four database instances, each of increasing size. Some statistics of the database instances are given in Table 6.1.

Table 6.2 shows the ratio of the cost of optimized query (including query transformation time) and the cost of original query for each pair of queries and each database instance. For DB1, the smallest database we used, performance worsened for 40% of the queries, and the extra overheads were limited to about 10%. Only 34% of the queries executed faster after optimization, out of which 20% were significant improvements. This result is expected because when the database is small, execution times of the original queries are low (in our experiment most of them took 1 to 2 seconds). Unless the outputs can be obtained without going to the database, the overheads of optimization usually more than offset the little savings derived. As the database grew in size, semantic optimization became much more effective. For DB4, the largest database we used, 67% of the queries executed faster after optimization. Furthermore, some queries (27%) which originally either took hours to execute or could not be executed at all (because the system ran out of resources) were able to be executed significantly faster after optimization.

We conclude, as one may expect, that it is probably not worth doing semantic optimization when the database is small or when the query execution cost is expected to be low, i.e. the semantic optimizer should be disabled (in fact it has a monitoring mechanism which would cause it to disable itself automatically). However when the database is large or when the query execution cost is expected to be high, the usefulness of the semantic optimizer is apparent.

Since the optimizer does not exhaustively check all the possible access paths, the QEPs generated might not always be optimal. However, our search strategy gives us access paths that are optimal or near-optimal in most cases. We evaluated 100 different queries with average selectivity factors equal to 0.25, 0.5, and 0.75 in turn and calculated the ratio of the cost of the optimal access path to the cost of the best one-source access path. The results are shown in Table 6.3. We observe that, for 70% of the queries, we were only able to reduce the execution costs by at most another 20% if we had continued to search for the optimal paths instead of stopping after getting the best one-source paths. These potential savings did not justify the extra computation and space overheads involved, especially when the query execution costs were relatively low. Hence in our current implementation we feel it suffices to consider only one-source paths. However, if a stop criterion for the searching of the optimal path can be determined, then search space is worth enlarged by considering multiple source access paths.

Table 6.2: Ratio of Optimized Cost and Original Cost

	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	110%
DB1	_	20	_	_	_	_	_	_	7	7	26	40
DB2	_	20	_	_	_	_	_	_	27	7	13	33
DB3	13	13	7	_	_	_	7	7	13	_	40	_
DB4	27	13	_	_	7	_	_	7	13	_	33	_

Table 6.3: Optimality of One-Source Paths

Cost of optimal path / Cost of best 1-source path	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Percentage of queries	6%	6%	9%	9%	-	24%	3%	43%

7. Conclusion

This paper reports the design and implementation of a query processor for our OODBMS. In this work, we clearly describe the process of translating a declarative query that possibly involves many classes and relationships into a sequence of method invocations that can be evaluated directly by the OODBMS. Each method invocation activates a method that retrieves objects in the class which the method belongs to. Several issues peculiar to the query optimization in OODBMS were discussed, in particular, the optimization of using pointers maintained in the objects to answer the query. In order to support new data types and new methods, the program transformation approach is adopted to generate the QEP. Some simulation studies have been carried out on this prototype query processor. The initial result shows that the optimizer is effective.

Our current work in this area is to enhance the access path determination algorithm to incorporate joins, instead of doing only pointer-chasing. Recall that in this work, if class B precedes class A in the chosen access path, objects in B can be retrieved only via pointers from A. Under certain circumstances, it might be cheaper to retrieve all the objects in A and then perform the join operation between these objects and the retrieved objects from B. Of course, incorporating joins complicates the algorithm considerably but we feel it is a possibility worth exploring.

Acknowledgements

We would like to thank Desai Narasimhalu and TokWang Ling for helping to improve an earlier version of this paper, and anonymous referees for useful comments.

References

[Ba*87] J. Banerjee, et al., *Data Model Issues in Object Oriented Applications*, ACM Transactions on Office Information System, Vol. 5, No. 1, Mar 1987.

[BGW81] P.A. Bernstein, N. Goodman, E. Wong, et al, *Query Processing in a System for Distributed Database*, ACM Trans. on Database Systems, 6(4):602-625, Dec 1981.

[CFM84] U.S. Chakravarthy, D.H. Fishman, J. Minker, *Semantic Query Optimization in Expert Systems and Database Systems*, Proc. of 1st Int. Workshop on Expert Database Systems, Oct 1984.

[CMG86] U.S. Chakravarthy, J. Minker, J. Grant, *Semantic Query Optimization: Additional Constraints and Control Strategies*, Proc. of the 1st Int. Conf. on Expert Database Systems, Apr 1986.

[Fre86] J.C. Freytag, *Rule-Based Translation of Relational Queries into Iterative Programs*, Proc. of the ACM SIGMOD 1986 Annual Conference.

[Fre87] J.C. Freytag, *A Rule-Based View of Query Optimization*, Proc. of the ACM SIGMOD 1987 Annual Conference.

[GrD87] G. Graefe, D. DeWitt, *The EXODUS Optimizer Generator*, Proc. of the ACM SIGMOD 1987 Annual Conference.

[HaZ80] M.M. Hammer, S.B. Zdonik, *Knowledge Based Query Processing*, Proc. of the 6th Int. Conf. on Very Large Data Bases, Sep 1980.

[Hue80] G. Huet, *Confluent Reductions: Abstract Properties and Applications of Term Rewriting Systems*, Journal of the ACM 27,4 (Oct 1980) pp. 797-821.

[Kin81] J.J. King, *QUIST: A System for Semantic Query Optimization in Relational Databases*, Proc. of the 7th Int. Conf. on Very Large Data Bases, Sep 1981.

[Kor88] H.F. Korth, *Optimization of Object-Retrieval Queries*, Proc. of the 2nd Int. Workshop on Object-Oriented Database Systems, Sep 1988.

[LNO89] H. Lu et al *On the Design of a Multimedia Object-Oriented Database System*, Working Paper, Institute of Systems Science, National University of Singapore, Dec 1989.

[LoM87] J. Lobo, J. Minker, *A Metaprogramming Approach to Semantically Optimize Queries in Deductive Databases*, Proc. of the 2nd Int. Conf. on Expert Database Systems, Apr 1988.

[Mal86] C.V. Malley, *A Knowledge-Based Approach to Query Optimization*, Proc. of the 1st Int. Conf. on Expert Database Systems, Apr 1986.

[MuD88] M. Muralikrishna, D. DeWitt, *Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries*, Proc. of the ACM SIGMOD 1988 Annual Conference.

[Osb88] S.L. Osborn, *Identity, Equality and Query Optimization*, Proc. of the 2nd Int. Workshop on Object-Oriented Database Systems, Sep 1988.

[PaS82] C.H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.

[PLO90] H. Pang, H. Lu, B. Ooi, *An Efficient Semantic Query Optimization Algorithm*, IEEE Int. Data Engineering Conf., Japan, 1991 (to appear).

[SAC79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price, *Access Path Selection in a Relational Database Management System*, Proc. of the ACM SIGMOD 1979 Annual Conference.

[ShC84] G.P. Shapiro, C. Connell, *Accurate Estimation of the Number of Tuples Satisfying a Condition*, Proc. of the ACM SIGMOD 1984 Annual Conference.

[ShO87] S.T. Shenoy, Z.M. Ozsoyoglu, *A System for Semantic Query Optimization*, Proc. of the ACM SIGMOD 1987 Annual Conference.

[Sie88] M.D. Siegel, *Automatic Rule Derivation for Semantic Query Optimization*, Proc. of the 2nd Int. Conf. on Expert Database Systems, Apr 1988.

[SwG88] A. Swami, A. Gupta, *Optimization of Large Join Queries*, Proc. of the ACM SIGMOD 1988 Annual Conference.

[Zdo89] S.B. Zdonik, *Query Optimization in Object-Oriented Databases*, IEEE Proc. of the 22nd Annual Hawaii Int. Conf. on System Sciences, 1989 (Vol II).

Appendix

In this Appendix, we show how we derive the QEP in Example 5.2 from the query in Example 5.1, using the access path in Figure 3.2.

```
(REQUIRE {(driver.name, supervisor.name)}
    (LINK {t0} {}
          {driver.licenseClass = vehicle.class,
           supervisor.licenseClass = vehicle.class}
          {engine.capacity > 1000}
          {drives, engComp}
          {driver, supervisor, vehicle, engine}
          resultClass))
                ↓   by Rule 5.2
(REQUIRE {(driver.name, supervisor.name)}
    ((SCAN t0 {} {} {} resultClass)
     (LINK {engine} {}
          {driver.licenseClass = vehicle.class,
           supervisor.licenseClass = vehicle.class}
          {engine.capacity > 1000}
          {drives, engComp}
          {driver, supervisor, vehicle}
          resultClass)))
                ↓   by Rule 5.2
(REQUIRE {(driver.name, supervisor.name)}
    ((SCAN t0 {} {} {} resultClass)
     (SCAN engine {} {} {engine.capacity > 1000}
       resultClass)
     (LINK {vehicle} {}
     {driver.licenseClass = vehicle.class,
      supervisor.licenseClass = vehicle.class,
      engComp ∈ resultClass.engine.engComp}
     {}
     {drives}
     {driver, supervisor}
     resultClass)))
          ↓   by Rule 5.5
(REQUIRE {(driver.name, supervisor.name)}
    ((SCAN t0 {} {} {} resultClass)
     (SCAN engine {} {} {engine.capacity > 1000}
       resultClass)
     (SCAN vehicle {} {engComp ∈
       resultClass.engine.engComp} {}
            resultClass)
     (FORK
          (LINK {driver} {}
               {driver.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                {} {} resultClass)
          (LINK {supervisor} {}
               {supervisor.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                {} {} resultClass))))
                ↓   by Rule 5.3
(REQUIRE {(driver.name, supervisor.name)}
    ((SCAN t0 {} {} {} resultClass)
     (SCAN engine {} {} {engine.capacity > 1000}
       resultClass)
     (SCAN vehicle {} {engComp ∈
       resultClass.engine.engComp} {}
            resultClass)
     (FORK
          ((SCAN driver {}
               {driver.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                resultClass)
           (LINK {} {} {} {} {} {} resultClass))
          ((SCAN supervisor {}
               {supervisor.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                resultClass)
           (LINK {} {} {} {} {} {} resultClass)))))
                ↓   by Rule 5.1
(REQUIRE {(driver.name, supervisor.name)}
    ((SCAN t0 {} {} {} resultClass)
     (SCAN engine {} {} {engine.capacity > 1000}
     resultClass)
     (SCAN vehicle {} {engComp ∈
     resultClass.engine.engComp} {}
            resultClass)
     (FORK
          (SCAN driver {}
               {driver.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                resultClass)
          (SCAN supervisor {}
               {supervisor.licenseClass = vehicle.class,
                drives ∈ resultClass.vehicle.drives} {}
                resultClass))))
          ↓   project attributes, remove dummy scan
((SCAN engine {} {} {capacity > 1000} resultClass)
 (SCAN vehicle {} {engComp ∈
  resultClass.engine.engComp} {} resultClass)
 (FORK
     (SCAN driver {name}
          {licenseClass = vehicle.class, drives ∈
           resultClass.vehicle.drives} {}
                resultClass)
     (SCAN supervisor {name}
          {licenseClass = vehicle.class, drives ∈
           resultClass.vehicle.drives} {}
                resultClass)))
```