

Performance of Query Optimization Heuristics in Object-Oriented Databases

Kyung Chang Kim

Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

Abstract : This paper investigates the performance of query optimization heuristics in object-oriented databases. Query optimization involves the following two stages : (1) Reduce the search space of access plans for a query to be considered during query evaluation using optimization heuristics (2) Estimate the cost of each access plan in (1) and select the cheapest access plan. To compute the cost of each access plan, a cost model is formulated which uses information such as the order of evaluation of classes in a query graph, and the access paths and retrieval algorithms used during the evaluation. The cheapest access plan selected is then used to evaluate the given query. Our performance results show that the optimization heuristics developed are effective and reasonable.

1. Related Work

Some navigational systems [DAYA82, KUCK84] employ primitive heuristics to reduce the search space of access plans before choosing the most efficient one during query optimization. There are also two approaches for query optimization in relational database systems.

In the first approach, each query access plan is generated and evaluated completely before executing the query. However the optimization effort for this approach is high. Therefore, complete planning has been proposed only for restricted environments : two-variable expressions [YAO79]; chain queries [CHIU81] and tree queries [GOUD81]; monadic opera-

tions in horizontally distributed databases [GAVI82]; and completely homogeneous networks [HEVN79]. However, for reasons of complexity, even some of these have to resort to heuristic methods to reduce the number of access plans.

In the other approach, heuristic methods are developed to limit the number of access plans. The well known System R uses heuristics to limit possible join strategies [SELI79]. This is done to make exhaustive search feasible.

Similar to System R, our technique for query optimization in an object-oriented database also employs heuristics to limit the number of access plans to be evaluated. However, by utilizing the semantics of queries in an object-oriented database, we apply a richer set of heuristics to eliminate many of the access plans which may not be eliminated in System R.

This paper discusses query optimization techniques in an object-oriented database. The remainder of the paper is organized as follows. Section 2 discusses the underlying object-oriented query model. In Section 3, we introduce heuristics which are used to reduce the search space of query access plans generated. Section 4 presents the cost model used to assign costs to access plans. In Section 5, extensive performance analysis are conducted to verify the effectiveness of the optimization heuristics. A summary is given in Section 6.

2. Object-Oriented Query Model

As mentioned in [KIM89b], in most existing operational object-oriented database systems, the query model captured in the query language fails to take into account some of the fundamental object-oriented constructs. For example, systems like VBase [ANDR87] merely support the SQL query language. Others, like IRIS [FISH87] and POSTGRES [STON86] provide a new language which is backward compatible with relational query languages SQL and QUEL respectively. The notion of a class hierarchy is not part of a relational data model and hence is not captured in the query model derived from its query language. Even the few query models proposed for object-oriented database systems [COPE84, ROWE87] fail to capture the query complexity of the impacts of the class hierarchy.

The details of the query model based on the model defined in [BANE88] have been discussed in [KIM89a, KIM89b]. In this paper, for ease of understanding, we briefly recap the query model.

A query may be formulated against a class in an object-oriented schema. The class to which a query is directed is called the *target* class. The query will fetch instances of the target class which satisfy certain search criteria and output only specified attributes of the instances fetched as the result of the query. This is analogous to the selection and projection operation for a single-relation query in a relational database. However, there are several major differences.

The concept of a class hierarchy in object-oriented databases which captures the IS-A abstraction suggests that the target class may be specialized into subclasses in the schema. Depending on the access scope of the query, the query will either fetch instances of the target class or the instances of the class hierarchy rooted at the target class which satisfy certain search criteria as the result of the query.

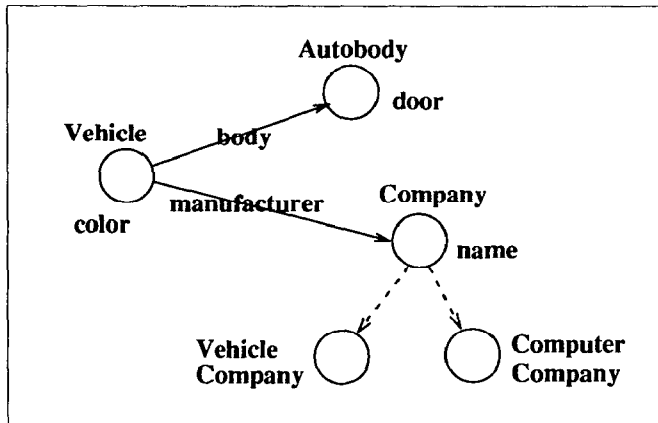
The query restricts the instances of the target class to be fetched by specifying predicates against any attributes of the class. The value of a *simple* attribute is of a simple data type like an integer or a character string while the value of a *complex* attribute is an instance(s) of another class. To fetch an instance of a class, the class and all direct and indirect domain classes of the complex attributes of the class must be recursively evaluated.

A predicate applied on a simple attribute of the target class is called a *simple predicate* while a predicate applied on a complex attribute is called a *complex predicate*. A complex predicate may also involve attributes of the direct and indirect domain classes of the complex attribute on which it is applied initially. A query that involves only simple predicates is called a *simple query* and one that involves simple predicates and one or more complex predicates is called a *complex query*.

A simple or complex query can be represented in the form of a directed graph called a *query graph*. Informally, each node in a query graph corresponds to a class involved in the query and an arc from a node A to a node B means that the domain of the complex attribute of class A is class B. If B is the root of a class hierarchy in the schema, its subclasses are also included as nodes in the query graph. In essence, the query graph contains the target class of the query which, depending on the access scope of the query, may include its subclasses. For those attributes on which predicates are applied, all direct and indirect domain classes of the attributes are also included. The query graph may also contain cycles.

As an example, the query "Select all blue 2-doored vehicles manufactured by Ford" is expressed below using the ORION query language syntax [BANE88] followed by the corresponding query graph form.

(Vehicle select :V (:V Color = 'blue'
and (:V Body Door = 2
and (:V Manufacturer Name = 'Ford'))))



The query is directed against class Vehicle which becomes the target class. For the sake of argument, we assume that the access scope of the query is class Vehicle. To restrict the instances of class Vehicle to be fetched as the result of the query, predicates on simple attribute Color (i.e. Color = 'blue'), complex attribute Body (i.e. Body Door = 2) and complex attribute Manufacturer (i.e. Manufacturer Name = 'Ford') of class Vehicle are applied. The predicate on complex attribute Body also extends to simple attribute Door of the domain class Autobody of attribute Body. In addition, the predicate on attribute Manufacturer also involves simple attribute Name of its domain class Company. The query evaluates three classes namely Vehicle, Autobody and Company. In addition, classes Vehicle Company and Computer Company which are subclasses of class Company in the schema are also evaluated.

3. Query Optimization Heuristics

In optimizing an n-relation join query in relational databases, the heuristic to be used is to postpone cross products of a relation and the intermediate result of the query [SELI79]. If the next relation to be joined is not connected by a join column with any of the relations already considered, that relation forms

a cross product with the intermediate result, and is not a candidate to be the next relation if there exists at least one other relation which does have a join column connecting it to one or more of the relations already considered.

In optimizing a query in object-oriented databases, the semantics of queries avoids cross products altogether. In addition, the semantics of queries in object-oriented databases make it possible to eliminate many of the access plans that may not be eliminated in evaluating relational queries. In this section, we will consider a number of heuristics which we have identified as effective and reasonable in reducing the computational complexity of query optimization in an object-oriented database.

Before we go on, we discuss two fundamental ways of traversing the classes in an access plan for a query. They are forward traversal and reverse traversal. Briefly, nodes (classes) on a query graph are traversed in a parent-to-child node order in forward traversal and child-to-parent node order in reverse traversal.

3.1. Simple-Predicate-First Heuristic (the SPF heuristic)

Rule: *The first node in an access plan, and hence the first class to be evaluated, must be a class associated with one or more simple predicates.*

In other words, the condition for being the first class in an access plan is to have predicates on the simple attributes of the class. This heuristic applies for any given query when index is supported during evaluation. The justification for the SPF heuristic is simple.

It is easy to see that if a class is associated with a simple predicate, the number of instances of the class that need to be kept for further evaluation is a fraction of the total number of instances of the class depending on the selectivity of the predicate.

For a forward traversal evaluation method, this suggests that only a fraction of the instances of each class need be evaluated. Likewise, for a reverse traversal evaluation method, only a fraction of instances of each class need to be evaluated. The reason is that if a class A is the first class in the access plan and is a class which has a simple predicate in the query, only a fraction of instances of class A are kept for further evaluation. This in turn requires only the qualified instances of the parent class of A to be accessed and so on.

3.2. Depth-first Instantiation Heuristic (the DFI heuristic)

Rule: *Once a node N of a query graph is included in an access plan, each node on the subtree of the query graph which is rooted at node N must be included in the access plan in a depth-first search order. If there are no child nodes of N, the parent node of N is included next.*

This implies that, once a node C appears in an access plan, any one of its child nodes is a candidate as the next node to be included; and if there are no child nodes, the parent node of C is included next. When used in conjunction with the SPF heuristic, this implies that, starting from a class associated with a simple predicate, all classes in the subtree rooted at the class are included in the access plan in a depth-first search order. An extension to this heuristic is an access plan which results in a reverse traversal of the nodes in a query graph.

The justification for the DFI heuristic is as follows. If a node (class) N has just been selected to be included in an access plan, we argue that the next node to select is a child node of N. The explanation is as follows.

Since node N is along a path from the root node to a leaf node in the query graph, two choices are made during query evaluation. (1) is to evaluate parent nodes of N successively all the way to the root node and then to evaluate all the child nodes of N successively to a leaf node in the query graph. (2) is

to first evaluate all the child nodes of N successively until a leaf node is encountered and then to evaluate the parent nodes of N successively all the way to the root node. It is more likely that (2) will be cheaper than (1) because the semantics of queries guarantee simple predicates only on leaf classes in a query graph.

The number of qualified instances for further evaluation goes down when a leaf class is evaluated. If we choose strategy (1) first, it is possible that the size of the intermediate result will not change since no simple predicates may be associated with any of the non-leaf classes. The DFI heuristic guarantees that we always get to a leaf node in the query graph as soon as possible by traversing the nodes, starting from node N, in a depth-first manner.

3.3. Forward Traversal Heuristic (the FT heuristic)

Rule: *In evaluating a cyclic query, or if indexes are not supported during evaluation, the forward traversal is probably at least as good as the other traversal methods.*

The justification is as follows. In evaluating an acyclic query, consider a node (class) C in a query graph. Reverse traversal of C implies that its child node D must be traversed and based on the instances of class D that are selected, a set of qualified instances of C must be selected. Since there are no indexes on the attributes of C, it is necessary to scan all instances of C. This implies that a complete scan of all instances of C is always required during reverse traversal of C.

On the other hand, a forward traversal of C implies that its parent node B must be traversed and based on the instances of B that are selected, a set of qualified instances of C must be selected. Since an instance of B directly selects an instance of C through its attribute value, either all instances of C or only a subset of all instances of C are selected regardless of the existence of an index. Hence, the maximum cost of forward

traversal of node C is a scan of all instances of C. This cost is not greater than the cost of reverse traversal of C, which always requires a complete scan of all instances of C.

In evaluating cyclic query, it is easy to show why forward traversal is as good as other traversal methods. Consider a node (class) C in a query graph. A reverse traversal of C implies that its child node D must be traversed and based on an instance of D that is selected, a set of qualified instances of C must be selected. If there are indexes on the attributes of C, it is necessary to do an index lookup using the selected instance of D as the key value, to select the qualified instances of C. If no index is supported on attributes of C, a complete scan of all instances of C is necessary.

On the other hand, a forward traversal of C implies that only one instance of C is selected regardless of whether an index is supported or not. Generalizing to all the nodes (classes) in a cyclic query graph, we easily see that forward traversing the nodes is always cheaper than reverse traversing the nodes.

4. Cost Model for Query Optimization

Given a query, the query optimizer first selects all reasonable access plans for the query using the heuristics. It then computes the cost of each of the access plans selected and choose the access plan with the minimum expected processing cost.

The cost of an access plan is the cost of evaluating the classes in the order determined by the access plan using a data retrieval algorithm. In this paper, we consider two retrieval algorithms; nested-loop and sort-domain, which are one record at a time and a set of records at a time evaluation algorithms respectively. The cheapest access plan for a given query computed by the query optimizer is presumably the access plan which results in minimum query evaluation time. The optim-

izer also determines the query evaluation algorithm to be used for this access plan as well as all efficient access paths for the plan.

The paper gives a flavor of the cost model used by the query optimizer. The comprehensive cost model to compute the cost of each of the selected access plans is given in [KIM89].

4.1. Statistics for the cost model

The statistics used in the cost model are information on selected classes and selected attributes of the classes. In a relational database, statistics are needed on individual relations and the values on individual attributes of tuples. In an object-oriented database, statistics must be gathered not only on a specific class but also on the subclasses of the class if exist. Hence the query optimizer must have access to more complex statistics, such as the sum of the number of instances of a class C and the number of instances of all subclasses of the class C.

An important parameter used in the cost model is selectivity. In object-oriented databases, selectivity can be divided into selectivity of a simple predicate, complex predicate or a class. The selectivity of a complex predicate is the selectivity of its domain class. The selectivity of a class is the combined selectivity of all the predicates involved in the class (and its subclasses) and selectivity of all resolved predicates corresponds to the ones that have already been applied during current query evaluation.

4.2. Cost Model

Costs are estimated in terms of disk page accesses. There are three categories in which cost can be computed for each of the nodes in any query graph. (1) is to compute the cost of accessing the first node to be traversed during evaluation. The

cost of evaluating the class is the cost of accessing all qualified instances of the class (and its subclasses) whose number corresponds to the selectivity of the class. (2) is to compute the cost of a particular node which is forward traversed. The cost of evaluating a class, say D, in this category is the cost of accessing qualified instances of class D based on the selectivity of another class, say C, in which class D is the domain of a complex attribute of class C. (3) is to compute the cost of a particular node which is reverse traversed. The cost of evaluating a class, say C, in this category is the cost of accessing qualified instances of C (and its subclasses) based on the selectivity of another class, say D, in which the domain of a complex attribute of C is D.

The three categories reflect all the situations under which cost can be computed for any particular node in a query graph during query evaluation. By having a generic cost model for each of the three categories, the cost of evaluating each node in a query graph and consequently the cost of any access plan can be computed.

5. Effectiveness of Query Optimization Heuristics

In this section, we analyze the effectiveness of the query optimization heuristics. The methodology we will use is as follows. First, we apply the heuristics to generate access plans for sample query types. The access plans generated is a reduced set of all access plans for the query type. Next, we test to see if the cheapest access plan for each query type corresponds to one of the access plans selected using the heuristics.

5.1. Query Test Types

To study the effectiveness of the query optimization heuristics, we identify fifteen query types as the basis for our performance tests. Some of the parameters used to vary typical query patterns are; predicates on simple attribute of a class,

predicates on complex attribute of a class, predicates on simple and complex attribute of a class, logical connectivity between predicates on a class and the existence of class hierarchy rooted at a class.

The query graphs of the fifteen generic queries are shown in Figure 1. For each generic query, we use symbols $C1$, $C2$, $C3$, $C4$, $C5$, $S1$ and $S2$ to be the generic classes in a schema. The labels on the directed arcs of each query graph, i.e. a , b , c or d , are complex attributes involved with query predicates. The labels on the nodes of each query graph, i.e. i , j , k or l , are simple attributes involved with query predicates.

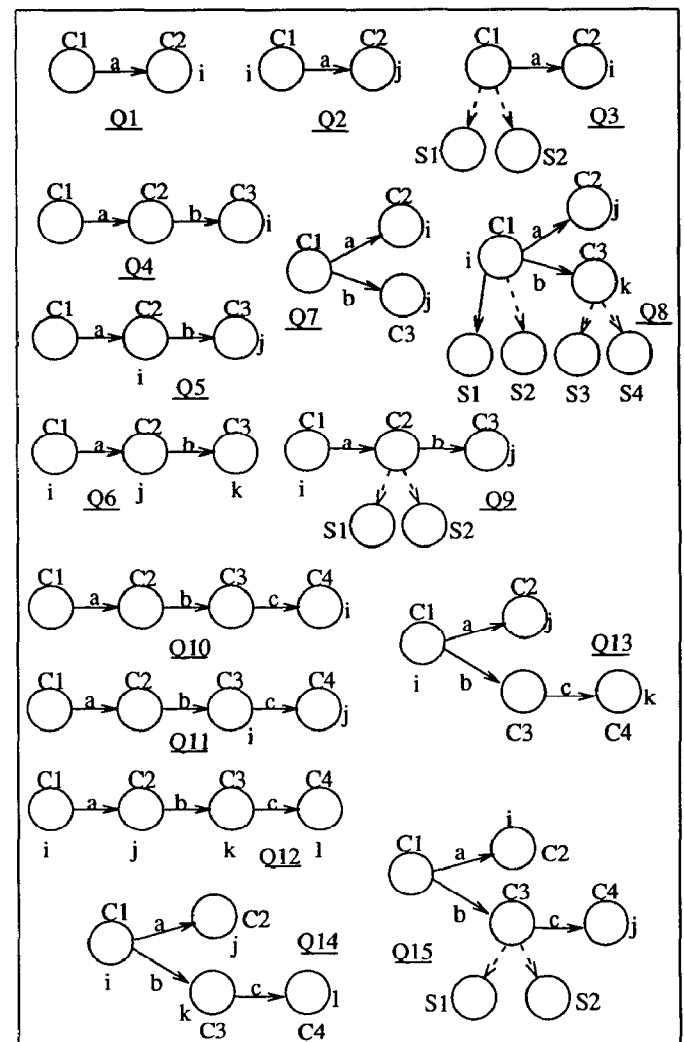


Figure 1 : Generic Query Types

As examples, in Q1 for instance, class C1 is the target class and a predicate is applied on complex attribute a. The predicate also involves a simple attribute i of class C2. Query Q2 is similar to Q1. The only difference is that a simple predicate is also applied on class C1. Query Q3 is also similar to Q1. The difference is that the access scope of the query is a class hierarchy rooted at C1. In Q1, the access scope of the query is just C1. Queries Q1, Q2 and Q3 are typical 2-class queries that can be formulated on any schema.

The test queries we propose are by no means complete. However, we believe that they represent typical query patterns for the respective query types.

5.2. Access Plans Generated

The methodology for generating access plans for each query is as follows. For each selected query, we ; (1) generate all access plans, (2) generate the reduced set of access plans using heuristics whenever applicable assuming that index is supported and (3) generate the reduced set of access plans using heuristics whenever applicable assuming that no index is supported. As examples, we select and generate access plans for queries Q1, Q9 and Q13.

Q1:

- (1) (C1, C2) (C2, C1)
- (2) (C2, C1)
- (3) (C1, C2)

Q9:

- (1) (C1, {C2,S1,S2}, C3) ({C2,S1,S2}, C1, C3)
(C2,S1,S2}, C3, C1) (C3, {C2,S1,S2}, C1)
- (2) (C3, {C2,S1,S2}, C1) (C1, {C2,S1,S2}, C3)
- (3) (C1, {C2,S1,S2}, C3)

Q13:

- (1) (C1, C2, C3, C4) (C1, C3, C4, C2) (C1, C3, C2, C4)
(C2, C1, C3, C4) (C3, C1, C2, C4) (C3, C1, C4, C2)
(C3, C4, C1, C2) (C4, C3, C1, C2) (C1, C1, C4, C3, C1)
- (2) (C1, C2, C3, C4) (C1, C3, C4, C2) (C2, C1, C3, C4)
(C4, C3, C1, C2) (C2, C1, C4, C3, C1)
- (3) (C1, C2, C3, C4) (C1, C3, C4, C2)

5.3. Performance Tests

The cheapest access plan for a query is the one which will take the least query processing time. For the heuristics to be effective, the cheapest access plan for any query should correspond to one of the access plans generated by the heuristics. Our claim is that this is indeed true in most cases.

To test the effectiveness of the heuristics, the following methodology is used. First, for each generic query, the cost of each of the access plans generated for the query is computed using the cost model. The cheapest access plan can then be selected. Next, the cheapest access plan for the query is compared with the cheapest access plan from the subset of access plans generated using the heuristics. To make performance tests and results as accurate as possible, we formulate the fifteen generic queries on different test databases.

The test databases are selected to reflect different database situations. The methodology used to create each test database is based upon a set of information for each class of the database. All test databases have the same number of classes. However, they will have a different set of characteristics for their containing classes. The following information is used to describe the characteristics of a test database and to distinguish one test database from another test database.

1. The number of instances of a class.
2. Number of disk pages containing instances of a class.
3. Indexes may not be supported or may be fully/partially supported on attributes of classes.
4. Selectivity for predicates on attributes of classes.

The characteristics of the test databases are shown in Table 2. For each database, there are three different entries for each of the classes. As indicated in the tables, the first entry is the number of instances, the second entry is the number of disk pages containing the instances, and the third entry is the

Test Database	class					Index
	1. no. of instances 2. no. of disk pages 3. selectivity					1. attr. 2. size 3. height
	C1	C2	C3	C4	C5	
DB 1	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	all 40 3
DB 2	30000 (930) 1/1000	30000 (930) 1/1000	30000 (930) 1/1000	30000 (930) 1/1000	30000 (930) 1/1000	all 40 3
DB 3	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	complex 40 3
DB 4	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	30000 (930) 1/100	none 0 0
DB 5	20000 (200) 1/50	20000 (150) 1/20	20000 (300) 1/10	20000 (100) 1/5	20000 (80) 1/4	all 30 3
DB 6	20000 (200) 1/5	20000 (150) 1/10	20000 (300) 1/20	20000 (100) 1/50	20000 (80) 1/100	all 30 3
DB 7	20000 (200) 1/50	20000 (150) 1/20	20000 (300) 1/10	20000 (100) 1/5	20000 (80) 1/4	simple 30 3
DB 8	20000 (200) 1/500	20000 (150) 1/500	20000 (300) 1/500	20000 (100) 1/500	20000 (80) 1/500	complex 30 3

Table 2 : Test Databases

selectivity. In addition, information on index support are given for each database. Index can be supported on all attributes of a class, on complex attributes of a class, on simple attributes of a class or not supported at all. When index is supported, the size and the height of the index are also given.

For example, in test database 1, the number of instances for each class is 30000 and the number of disk pages to store the instances is 930. In addition, index is supported on all attributes of each class. The index size for each index supported is 40 index pages and the height of each index is assumed to be 3. The selectivity factor for each class is assumed to be 1/100. In comparison, test database 2 is similar to the characteristics of test database 1 except that the selectivity factor of each class in test database 2 is 1/1000.

5.4. Performance Analysis

In this section, we present and analyze the results of our performance tests on the effectiveness of the heuristics during query evaluation. For each query formulated on a specific test database, the cheapest access plan from a whole set is compared with the cheapest access plan from a reduced set. Access plans in the whole set are all access plans which are generated for the query. Access plans in the reduced set are the subset of access plans in the whole set which are generated using the heuristics mentioned in Section 3.

For clarity, the results of performance tests for the queries on test database 1 are described in Table 3. Each Table 3 entry contains the cheapest access plan from the whole set as well as the cheapest access plan from the reduced set for each of the fifteen generic queries considered.

For each table entry in Table 3, additional information, associated with the cheapest access plan, is; the evaluation algorithm used to evaluate the access plan and the I/O cost incurred. Symbol S means that sort-domain algorithm is used to execute the access plan and symbol N means that nested-loop algorithm is used. The I/O cost incurred is measured in terms of the number of disk pages fetched into memory. For example, the entry for the cheapest access plan from the whole set in query Q1 is (C2, C1); S; 43. This means that (C2, C1) is the cheapest access plan for Q1 and based on the access plan, it requires 43 disk page fetches to evaluate Q1 using sort-domain algorithm.

Some of the query types in Table 3 are associated with *. This shows that, using heuristics, we have failed to select the cheapest access plan for the query. For example, the cheapest access plan for Q6 is (C2, C1, C3) whose I/O cost is 49 disk pages. On the other hand, the I/O cost for the cheapest access plan (C3, C2, C1) from the reduced set is 58 disk pages. This shows that the optimizer, using heuristics, has failed to choose

the cheapest access plan for Q6. Table 3 shows that the optimizer fails to choose the cheapest access plan for queries Q6, Q12.

Query Types	Cheapest Access Plan from Whole Set	Cheapest Access Plan from Reduced Set
	Algorithm ; Disk Pages	Algorithm ; Disk Pages
Q1	(C2,C1) ; S ; 43	(C2,C1) ; S ; 43
Q2	(C2,C1) ; S ; 46	(C2,C1) ; S ; 46
Q3	(C2,C1) ; S ; 43	(C2,C1) ; S ; 43
Q4	(C3,C2,C1) ; S ; 83	(C3,C2,C1) ; S ; 83
Q5	(C3,C2,C1) ; S ; 55	(C3,C2,C1) ; S ; 55
Q6*	(C2,C1,C3) ; S ; 49	(C3,C2,C1) ; S ; 58
Q7	(C3,C2,C1) ; S ; 86	(C3,C2,C1) ; S ; 86
Q8	(C2,C1,C3) ; S ; 52	(C2,C1,C3) ; S ; 52
Q9	(C3,C2,C1) ; S ; 86	(C3,C2,C1) ; S ; 86
Q10	(C4,C3,C2,C1) ; S ; 123	(C4,C3,C2,C1) ; S ; 123
Q11	(C4,C3,C2,C1) ; S ; 64	(C4,C3,C2,C1) ; S ; 64
Q12*	(C2,C1,C3,C4) ; S ; 50	(C4,C3,C2,C1) ; S ; 64
Q13	(C2,C1,C3,C4) ; S ; 52	(C2,C1,C3,C4) ; S ; 52
Q14	(C2,C1,C3,C4) ; S ; 50	(C2,C1,C3,C4) ; S ; 50
Q15	(C2,C1,C4,C3,C1) ; S ; 126	(C2,C1,C4,C3,C1) ; S ; 126

Table 3 : Results on Test Database 1

When the optimizer fails to select the best access plan, our results show some common characteristics. First, a simple predicate is always applied on an attribute of the root (target) class as well as on an attribute of a non-leaf class in the query graph of the query. In addition, the selectivity of a predicate on a class is such that the number of qualified instances selected for the class is always greater than the size, in number of index pages, of any index supported on the class. Also, in the query graph for each such query, the selectivity of a predicate may be higher in a parent class than in its child class but not vice versa. It is easy to see that these characteristics favor reverse traversing classes during evaluation. Since the optimizer uses the DFI heuristic, it favors forward traversing classes during evaluation and heuristics may not be effective in queries having the characteristics mentioned.

For acyclic queries formulated on databases which do not support indexes, the test results show that the optimizer is not very effective in selecting the best access plan. Some of the

characteristics of such queries are as follows. One characteristic is that no simple predicate is associated with the root (target) class in the query graph for the query. The other is that the number of qualified instances of the target class is always greater than the number of disk pages containing instances of classes. In addition, the number of instances of the target class is always equal or greater than the number of instances of some class on which a simple predicate is applied.

Under these situations, it is easy to see why the optimizer may fail to select the cheapest access plan. The characteristics mentioned above favor reverse traversing the classes as opposed to forward traversing because less number of qualified instances are retained for each class evaluated. In the absence of indexes, the optimizer uses the FT heuristic to generate access plans for each query and hence the forward traversal of classes during evaluation.

Table 4 summarizes the success rate of optimizing the generic query types formulated on the test databases. It shows that the heuristics are effective in optimizing queries formulated on databases which always support indexes. Specifically, the optimizer selects the best access plan in over 90 % of the

Test Database	Success Rate using Heuristics
Database 1	90 %
Database 2	100 %
Database 3	96.7 %
Database 4	46.7 %
Database 5	93.4 %
Database 6	100 %
Database 7	100 %
Database 8	93.4 %

Table 4 : Effectiveness of Heuristics

queries formulated on the databases. Table 4 also shows that the optimizer is not as effective in optimizing queries formulated on databases which do not support indexes. Specifically, the optimizer succeeds in selecting the cheapest access plan in only about half the queries formulated on these databases.

6. Summary

The semantics of queries in an object-oriented database make it possible to eliminate many of the access plans which may not be eliminated in evaluating relational queries. In this paper, we introduced a number of heuristics which we identified as reasonable and effective in reducing the computational complexity of query optimization in object-oriented databases. These heuristics were used to select a reduced set of access plans from all access plans generated for a given query. We then introduced a cost model to compute the cost of an access plan for a query. We conducted extensive performance tests to verify the effectiveness of the heuristics. This was done by comparing the cheapest access plan for a query with the cheapest access plan from the set of access plans generated by the heuristics.

Analysis of the test results show the effectiveness of the heuristics in optimizing queries formulated on databases which always support indexes. The results show that the heuristics are not as effective in selecting the best access plan for queries formulated on databases which do not support indexes. However, for each such query, our results show that the increase in I/O cost is not significant.

We do not claim that our test results are always reliable. This is because we did not consider every possible query type and database situation. However, the test results and analysis should still be useful because every attempt was made to select generic query types and database situations and we identified general situations under which the heuristics might fail to select the best access plan for a given query.

References

- [ANDR87] Andrews, T., and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment," in Proc. 2nd Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, Oct. 1987
- [BANE88] Banerjee, J., Won Kim and Kyung-Chang Kim, "Queries in Object-Oriented Databases", Proc. 4th. International Conference on Data Engineering, L.A. Feb. '88
- [CHIU80] Chiu, D.M, and Y.C. Ho, "A methodology for interpreting tree queries into optimal semi-join expressions," Proc. ACM SIGMOD, Santa Monica, 1980, 169-178
- [CHIU81] Chiu, D.M, et. al., "Optimizing chain queries in a distributed database system," TR-01-81, Harvard University, 1981
- [COPE84] Copeland, G. and D. Maier, "Making Smalltalk a Database System," Proc. ACM SIGMOD, June 1984, pp. 316-325
- [DAYA82] Dayal, U. and N. Goodman, "Query Optimization for CODASYL Database System," Proc. ACM-SIGMOD, June 1982, pp. 138-150
- [FISH87] Fishman, D., et. al., "IRIS: An Object-Oriented Database Management System," ACM Transaction on Office Information Systems, vol. 5 no.1 Jan 87, pp 48-69
- [GAVI82] Gavish, B., and A. Segev, "Query optimization in distributed computer systems," Akoka J (ed), Management of Distributed Data Processing, North Holland, 1982
- [HEVN79] Hevner A.R., and S.B. Yao, "Query processing in distributed database systems," IEEE Trans. on Software Engineering, SE-5,3, May 1979, 177-187
- [KIM89a] Kim, Kyung-Chang, Won Kim and Al Dale, "Cyclic Query Processing in Object-Oriented Databases," Proc. 5th. International Conference on Data Engineering, L.A. Feb. '89
- [KIM89b] Kim, Won, "A Model of Queries for Object-Oriented Databases," in Proc. 15th. Intl. Conf. on Very Large Data Base, Amsterdam, 1989
- [KUCK84] Kuck, S.M. and Y. Sagiv, "Links in Relational Databases," Proc. of Canadian Conf. on Information Processing Session, May 1984
- [ROWE87] Rowe, L., and M. Stonebraker, "The POSTGRES Data Model," in Proc. Intl. Conf. on VLDB, Brighton, Sept. 1987, pp. 83-95
- [SELI79] Selinger, P.G., et al., "Access Path Selection in a Relational Database Management System," Proc. ACM SIGMOD, 1979, pp. 23-34
- [STON86] Stonebraker, M. and L. Rowe, "The Design of POSTGRES," in Proc. ACM SIGMOD Intl' Conf. on Management of Data, May 1986
- [YAO79] Yao, S.B., "Optimization of Query Evaluation Algorithms," ACM TODS, 4(2), June 1979, pp. 133-155