Query Optimization Techniques Utilizing Path Indexes in Object-Oriented Database Systems

Wan-Sup Cho*, Kyu-Young Whang**, Seung-Sun Lee*, Yong-Ik Yoon*

*Real-time OS Section, ETRI
161 Kajung-Dong, Yusung-Ku, Taejon, 305-350, Korea
**Department of Computer Science, KAIST
Taejon, 305-701, Korea
wscho@mozart.kaist.ac.kr, kywhang@cs.kaist.ac.kr
{ssunlee, yiyoon} @nice.etri.re.kr

Abstract

We propose query optimization techniques that fully utilize the advantages of path indexes in object-oriented database systems. Although path indexes provide an efficient access to complex objects, little research has been done on query optimization that fully utilizes path indexes. We first devise a generalized index intersection technique, adapted to the structure of the path index extended from conventional indexes, for utilizing multiple (path) indexes to access each class in a query. We then propose the query graph reduction algorithm that replaces the classes in the query graph with path index scans; we call the resultant query graph reduced query graph (RQG). We finally present the search algorithm that finds the least-cost evaluation plan from RQG based on a cost model. Proposed query optimization techniques enhance database performance by using path indexes instead of direct accesses to data in the evaluating queries.

Keywords object-oriented databases, query optimization, path indexes, cost model.

1 Introduction

Optimal query processing in object-oriented database systems is an important issue for enhancing the performance of object-oriented database systems [14, 18]. Although most object-oriented database systems provide high-level query languages [11, 13, 18], query optimization techniques are immature yet [3, 5, 6, 14, 18]. Most of the previous query optimization techniques in object-oriented database systems use a variant of relational query optimization techniques for the object-oriented queries that have simple expressive power. Furthermore, the effects of inherent features of object-oriented queries and new access structures are not taken into account. New access structures in object-oriented databases include pointers in complex attributes (reference attributes [21]) and path indexes [1, 17] constructed on path expressions.

Path indexes provide an efficient access to complex objects since a condition on a path expression can be evaluated by using the indexes without fetching any database objects. However, the join algorithms and query optimization techniques that take advantage of the path indexes have not been proposed in the literature; most earlier work focuses on the structure of the path index itself or on the optimal path index configurations for physical database design [1, 2, 17]. To enhance database performance with path indexes, research on all these fields should go together since the decision whether a (path) index matched with a given query is to be used or not is made by the query optimizer and the index is scanned in the processing of the joins.

In this paper, we present query optimization techniques that take advantage of path indexes. Proposed query optimizer finds the least-cost evaluation plan that might use multiple access structures (i.e., pointers, simple indexes, path indexes) for accessing each class in the query. For using multiple access structures, we first devise a generalized index intersection technique [4, 19]; conventional index intersection technique is generalized for handling the extended structure of the path index from that for the simple index. We then present a method for deciding eligible path indexes for a given query and an algorithm for removing the classes that can be processed by the path index scans from the query graph; the resultant query graph is called reduced query graph (RQG). We finally propose a search algorithm that finds the least-cost query evaluation plan from the RQG based on a cost model. Since the search is done on RQG, the search space can be reduced significantly as the number of matched path indexes increases.

Query optimizers that take into account the effect of multiple path indexes in accessing each class in a query have not been proposed. Blakeley et
al.[3] presents the experiences of extensible query optimization in object-oriented database systems. They proposed a query transformation rule called “collapse-to-index-scan” that transforms a part of an algebra tree into a path index scan. However, they do not present the details of the query optimization techniques such as cost modeling, complete query transformation rules with some priority, or search algorithms. Lanzelotte et al.[16] proposes a similar query transformation rule. However, both techniques do not use multiple access structures for accessing each class in the query. Furthermore, since these techniques find the solution based on the query transformation rules, they have to test the applicability of each transformation rule at each step in the query transformation; therefore, they require much testing overhead[10, 23]. In contrast, our technique finds the solution based on the dynamic algorithm with branch and bound pruning, which has been successfully used in Selinger et al.[20, 23].

The paper is organized as follows. In Section 2, we review the concept of object-oriented databases and query languages. In Section 3, we present the query optimization techniques that take full advantage of the path indexes. We describe the generalized index intersection technique, the query graph reduction algorithm, and the search algorithm for finding the least-cost evaluation plan. In Section 4, we present the cost model for the proposed query optimization. Finally, we conclude the paper in Section 5.

2 Object-Oriented Databases

In this section, we describe the schema graph, queries, and indexes in object-oriented databases that are necessary to discuss the remainder of this paper.

We use a generally accepted object-oriented data model described in the literature[13]. Objects are uniquely identified by object-identifiers (OID). Objects having the same properties (attributes and methods) are classified into a class. The definition of a class forms a two dimensional rooted directed graph of classes called the schema graph for that class. Figure 1 shows the schema graph for the class Person in a database.

Most of the query languages in object-oriented databases are an extension of the relational query SQL for supporting object-oriented concepts. We assume the query model proposed by Kifer et al.[11]. Consider the following query:

Select students who own a red-colored car, made by Ford, whose horse power is over 100.

Q1: SELECT * FROM Student S WHERE S.own[Car].color = “Red” // pred1

Figure 1: An Object-Oriented Database.

Figure 2: Query Graph for the Query Q1.

query processor checks semantic errors by using catalog information. The query optimizer determines the indexes, the join order, and the join algorithm that will be used in query processing. Note that although a given query refers to a class (or an attribute) more than once, the class (or the attribute) appears only once in the query graph. For example, the class Student and the attribute own in the query Q1 are referred to four times by the query, they are represented only once in the query graph.

In object-oriented databases, conventional indexing techniques are extended for efficient pro-
cessing of conditions on path expressions. A path index provides a set of qualified OID tuples of the classes in the path expression on which the index is defined. For example, the path index defined on the path expression Person.own.manuf.name, denoted by \(IDX(Person.own.manuf.name)\), has the leaf node consisting of the following two fields:

- key field: Company.name
- pointer field: OID tuples of the classes Person (including Employee, Student), Vehicle (including Car, Truck), and Company

Figure 3 (a) shows the structure of \(IDX(Person.own.manuf.name)\). In the leaf nodes of the index in

![Figure 3: Index Structure.](image)

Figure 3, the OID tuple \([\text{Cho, car1, Company}]\) consists of the OIDs of the classes Person, Vehicle, and Company; it represents that the person Cho has the car car1 made by the company Ford (i.e., Company). In contrast, simple indexes are constructed for the path expressions of unit length. For example, the index \(IDX(Person.age)\) shown in Figure 3(b) has the key attribute age and the pointer field Person OID. As shown in Figure 3 (a) and (b), the path index extends the pointer field of the simple index so as to have OID tuples.

3 Query Optimization Techniques in Object-oriented Database Systems

In this section, we present query optimization techniques that take advantage of the path indexes in object-oriented database systems. For the given query, proposed query optimizer selects multiple access structures for each class in the query, produces the reduced query graph, and finds the least-cost evaluation plan based on a cost model. In addition, we present the generalized index intersection technique modified for path indexes in object-oriented databases.

### 3.1 Access Path Selection

In this section, we propose a criterion for selecting access structures that will be used in query processing. The access structures considered in this paper are pointers, simple indexes, and path indexes. We first define terminology and then propose the selection criterion.

(Definition 1) Eligible indexes for the given query \(Q\), denoted by \(EI(Q)\), are the indexes that are useful in query processing; for the condition \(\text{path}_i \theta \text{value}\), eligible indexes are the indexes constructed on ‘any subpath’ of the path expression \(\text{path}_i\). If the condition \(\text{path}_i \theta \text{value}\) (or \(\text{subpath}_i \theta \text{value}\)) can be processed by a (path) index, we call the condition an index processible predicate (IP) [22]. Predicates that cannot be processed by indexes are called residual predicates (RP) [22].

Figure 4 shows an example query graph used in the remainder of this paper. In the figure, the link (i.e., the attribute) that connects the classes \(C_i\) and \(C_j\) is represented by the symbol \(a_{ij}\). The dotted arrow denotes the path index constructed on the corresponding path expression. In the query graph, since the predicate \(\text{pred2}(: C_1.a_{1/2}.a_{2/3}.a_{3/9} \theta \text{value})\) can be processed by using the path index \(IDX(C_1.a_{1/2}.a_{2/3}.a_{3/9})\), \(\text{pred2}\) becomes an index processible predicate and \(IDX(C_1.a_{1/2}.a_{2/3}.a_{3/9})\) becomes an eligible index.

![Figure 4: A Query Graph.](image)

The problem of determining eligible indexes in the query optimization has exponential time complexity [19] like the problem of physical database design. Therefore, we use a simple index selection heuristic: "all eligible indexes and pointers are selected." For example, in Figure 4, all three indexes and all pointers in the reference attributes are selected; the pointers can be used in the joins of adjacent classes. Since multiple joins and a selection operation can be evaluated by a single probing of the matched path index, the benefit from path indexes is significant in most cases. We use as many pointers as possible since they provide direct access to objects to be joined. We regard the reverse join...
that has a reverse pointer as the forward join since we can fetch the objects to be joined via reverse pointers.

Note that the paper's goal, that is taking full advantage of the path indexes, is not compromised by the proposed index selection heuristic. The reason is that if we find the (sub) optimal index set for a given query in the future, the proposed optimizer would simply use the (sub) optimal index set (i.e., the index set obtained from the above heuristic); therefore, the query optimizer is unchanged. We are now devising a polynomial time algorithm that finds the optimal index set for a given query under some restrictions.

### 3.2 Generalized Index Intersection

In this section, we describe conventional index intersection technique (IIT) [4, 19] and extend it for different structures of path indexes in object-oriented databases.

IIT is essential for using more than one index in the access of a single table. Conventional IIT acts as follows. When a predicate involving key fields of an index exist, the RIDS list of the tuples that satisfy the predicate is found by using the index. When there is another conjunctive predicate on the key fields of another index, the technique generates another RIDS list by using the second index, intersects the two RIDS lists, and fetches the qualified records by using the resultant RIDS list. Disjunctive predicates can be processed in a similar way by using index unions. The benefit of IIT is presented in detail in the literature [4, 19]. Most of the early database management systems did not use IIT in fear of index access overhead. However, the benefit of IIT is proved lately and several commercial database management systems such as IBM DB2 Family [9, 19] and NonStop SQL [5] use this technique. Figure 5 shows an example of the index intersection technique. The conjunctive predicates Employee.age = 25 AND Employee.salary = 25,000 can be processed by the intersection of two indexes IDX(Employee.age) and IDX(Employee.salary).

For accommodating path indexes, conventional IIT should be extended since the structure of the path index is different from that of the conventional simple index. Figure 3 shows the difference; the pointer field of the path index contains not a set of OIDs but a set of OID tuples (i.e., a relation) of degree n (≥ 1). Therefore, we use the natural join operation instead of the intersection operation for obtaining qualified OID tuples. In the remainder of this paper, the index intersection means the natural join of two sets of OID tuples obtained from the eligible (path) indexes. Figure 6 shows the process of the index intersection in

![Figure 5: Index intersection for simple indexes.](image)

![Figure 6: Index intersection for path indexes.](image)
3.3 Query Graph Reductions

In this section, we present an algorithm that determines the classes that are replaced by the index scans and removes them from the query graph. The resultant query graph becomes the input data structure of the search algorithm described in Section 3.4.

We use Higraph\[10\] for modeling the process of the query graph reduction. Compared with conventional graph, Higraph has one extra element called supernode that contains one or more subnodes (classes). The query graph reduction algorithm consists of the following three steps:

1. For query graph $QG$, determine the set of eligible indexes $EI(QG)$.

2. For each $IDX(path_i) \in EI(QG)$
   1) remove all primitive classes and edges in $path_i$.
   2) create a new supernode that contains all user defined classes in $path_i$; the supernode denotes OID tuples of its subnodes that satisfy the predicates matched with $IDX(path_i)$.

3. If two supernodes (relations) $T_1$ and $T_2$ have a common subnode, perform natural join for them. The join result is denoted by another supernode $T_{12}$ and the nodes $T_1$ and $T_2$ are removed. We repeat this step until no more supernodes exist in the query graph that share a subnode.

In Step 2, we do not remove the user-defined classes on $path_i$ since residual predicates may exist for them. For example, the residual predicate $pred1$ exists for the subnode $C_1$ in $T_{13}$. Since $pred1 : C_1.a_1 \neq a_2.a_3$ can be evaluated by using the index $IDX(C_1.a_1,a_2,a_3)$, we remove the primitive class $C_1$ and the edges $a_1, a_2, a_3$ from the query graph. Then we make the supernode $T_{13}$ that includes $C_1, C_2$, and $C_3$. $T_{13}$ contains OID tuples of its subclasses ($C_1, C_2, C_3$) that satisfy the predicate $pred1$; in the query processing, these OID tuples are obtained from the index $IDX(C_1.a_1,a_2,a_3)$. Figure 7(a) shows the resultant reduced query graph.

2. Since the join of $C_4$ and $C_5$ can be evaluated by using the index $IDX(C_4,a_4,a_5)$, we create the supernode $T_{45}$ that includes $C_4$ and $C_5$. Similarly, we generate the supernode $T_{14}$ since $pred6$ can be evaluated by using the index $IDX(C_1.a_1,a_2,a_3)$. Figure 7(b) shows the resultant reduced query graph.

3. In Figure 7(b), since the supernodes $T_{123}$ and $T_{14}$ have the common subnode $C_1$, natural join for them is done. The nodes $T_{123}$ and $T_{14}$ are replaced by the new supernode $T_{1234}$. Figure 6 shows this natural join in detail. $T_{1234}$ contains OID tuples of its subnodes that satisfy both $pred2$ and $pred6$; in the query processing, query processor obtains $T_{1234}$ by the intersection of the indexes $IDX(C_1,a_1,a_2,a_3,a_4,a_5)$ and $IDX(C_1.a_1,a_2,a_3,a_4,a_5)$. Figure 7(c) shows $T_{1234}$.

4. In Figure 7(c), since two supernodes $T_{1234}$ and $T_{45}$ have the common subnode $C_4$, natural join for them is done, and the nodes $T_{1234}$ and
$T_{45}$ are replaced by the new supernode $T_{12345}$. Figure 7(d) shows $T_{12345}$.

In Figure 7(d), the supernode $T_{12345}$ represents the resultant OID tuples of the intersection of the indexes $IDX(C_1.a_1/a_2/a_3/a_9/9)$, $IDX(C_4.a_4/a_5/a_6/a_7/a_8)$, and $IDX(C_1.a_1/a_4/a_5/a_6/a_7/a_8)$, therefore, $T_{12345}$ contains OID tuples that satisfy $pred2$, $pred6$, and the join of $C_4$ and $C_5$.

3.4 Generation of the Least-Cost Evaluation Plan

In this section, we present the search algorithm that constructs the least-cost query evaluation plan from the reduced query graph (RQG). The search algorithm uses dynamic algorithm with branch and bound pruning technique as in Selinger et al. [20]. The search algorithm always finds the optimal solution, but the search space extends exponentially as the number of classes in the query increases. Since most of the queries in real environments include small number of classes, many query optimizers use this search algorithm. Especially, the proposed search algorithm is done for the reduced query graph (RQG), the more the number of path indexes matched, the less the number of classes to be searched.

In this paper, we exclude the join orders that include Cartesian products. Although this kind of join orders can be the optimal solution, it is rare; therefore, most literature excludes them. The search algorithm generates all possible join orders (or alternative plans) from the RQG, and then estimates evaluation cost for each join order, and finally chooses the least-cost join order based on the cost model. In the following, we present the process of the search algorithm for Figure 7(d).

(1) Generation of Search Tree

Search tree describes possible join orders generated from the RQG. Figure 8 shows the search tree for Figure 7(d). A branch from the root to a leaf node denotes a join order that does not Cartesian products. Since every node (user-defined classes or supernodes) in Figure 7(d) can be the first class of the join order, nodes $T_{12345}$, $C_6$, and $C_7$ can be at level-1. After accessing $T_{12345}$, either $C_6$ or $C_7$ can be accessed without Cartesian products (level-2). After accessing $C_6$ (or $C_7$), $C_7$ (or $C_6$) can be accessed without Cartesian products (level-3). Sub-trees for $C_6$ and $C_7$ at level-1 can be made in a similar way.

(2) Cost Estimation and The Least-cost Evaluation Plan Generation

After generating the search tree, the search algorithm estimates the cost of each branch, and finds the least-cost evaluation plan for the given query based on the cost model presented in Section 4. The costs for the branches are evaluated in the breadth first way as in Selinger et al. [20]. After calculating costs in each level, equivalent solutions (actually partial solutions) are determined and all solutions except the least-cost are removed. For example, at level-2 of Figure 8, equivalent partial solutions are as follows.

- $E_1 = \{ T_{12345} - C_6, C_6 - T_{12345} \}$
- $E_2 = \{ T_{12345} - C_7, C_7 - T_{12345} \}$

All elements in an equivalent solution have the same subtree. For example, all elements in $E_1$ have the subtree rooted at $C_7$. In the sets $E_1$ and $E_2$, elements except the least-cost are removed. After finishing the cost estimation for the branches, the optimizer constructs the evaluation plan for the least-cost branch.

4 Cost Model

In this section, we present a cost model for estimating the cost of each branch in the search tree.

Since the cost of index intersection is equivalent irrelevant to the join orders, we ignore it. Table 1 shows the parameters used in the cost model.

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OIDL</td>
<td>OID length (bytes)</td>
</tr>
<tr>
<td>PSize</td>
<td>page size (bytes)</td>
</tr>
<tr>
<td>n(C)</td>
<td>number of objects in C</td>
</tr>
<tr>
<td>p(C)</td>
<td>number of pages of C</td>
</tr>
<tr>
<td>bf(C)</td>
<td>blocking factor of C</td>
</tr>
<tr>
<td>sel(pred)</td>
<td>selectivity for pred</td>
</tr>
<tr>
<td>sel(C)</td>
<td>selectivity for C</td>
</tr>
<tr>
<td>LP(C)</td>
<td>local predicates of C</td>
</tr>
<tr>
<td>RP(C)</td>
<td>residual predicates of C</td>
</tr>
<tr>
<td>INP(C)</td>
<td>index processable nested predicates of C</td>
</tr>
</tbody>
</table>

(Theorem 1) [Yao77] Given n records grouped into m blocks ($1 \leq m \leq n$), each containing $bf = n/m$ records. If k records are randomly selected from the n records, the expected number of blocks hit (blocks with at least one record selected) is given by
\[ b(m, bf, k) = m \times \left[1 - \frac{n-bf}{k} \right] \] (1)

In the next, we present the cost estimation for an arbitrary branch \(< C_1, C_2, ..., C_n >\) in the search tree. Here, \(C_i\) is either a user-defined class or a set of OID tuples produced by index intersections. We do not distinguish the set of OID tuples and intermediate join results since both have OID tuples as elements. Since the joins of the branch \(< C_1, C_2, ..., C_n >\) can be processed by the sequence of binary joins, we present the cost formula only for the binary join. We also assume pointer-based sort-merge join algorithms [12, 21]; cost formulas for other join algorithms [12, 21] can be modeled similarly. Formula (2) shows the cost for the binary join of \(C_i\) and \(C_{i+1}\).

\[
\text{cost}(C_i \bowtie C_{i+1}) = \text{cost}(C_i) + \text{sort}(C_i, a_i) + \text{cost}(C_{i+1}) \tag{2}
\]

(1) \text{cost}(C_i)
This is the access cost of \(C_i\) objects for processing residual predicates of \(C_i\). When \(C_i\) is a user-defined class (i.e., \(C_i\) has no eligible indexes), \(p(C_i)\) pages are fetched. When \(C_i\) is the set of OID tuples (i.e., \(C_i\) is an intermediate result), the query processor fetches each page of \(C_i\) and checks whether the OID tuples in the page satisfy the residual predicates. In order to find the qualified OID tuples, objects on which at least one residual local predicate are applied should be fetched. Therefore, the number of pages to be fetched is \(p(C_i) + n(C_i) \times \{\text{number of classes on which the residual local predicates are applied}\}\). We can avoid repeated page access by sorting the OIDs in the set of OID tuples according to the physical address before fetching individual objects.

(2) \text{sort}(C_i, a_i)
This is the cost of sorting \(C_i\) objects according to the values of the attribute \(a_i\). When \(C_i\) is a user-defined class, only qualified objects satisfying the index processible predicates and the local residual predicates are to be sorted. Therefore, the number of pages to be sorted is estimated as follows:

\[ n(C_i) \times \text{sel}(LP(C_i), INP(C_i)) \times 2 \times \text{OIDL} \times \frac{\text{PSize}}{\text{PSize}} \]

\[ n(C_i) \times \text{sel}(RP(C_i)) \times (\text{degree of } C_i) \times \text{OIDL} \times \frac{\text{PSize}}{\text{PSize}} \]

Here, \(\text{sel}(LP(C_i), INP(C_i))\) is the product of the selectivities of the local predicates and the index processible predicates for \(C_i\). On the other hand, when \(C_i\) is the set of OID tuples, since only qualified OID tuples satisfying the residual predicates are sorted, the number of pages to be sorted, \(p(C_i)\), is estimated as follows:

Therefore, by [15] the sorting cost becomes \(2 \times p(C_i) + 2 \times p(C_i) \times \log_2 [p(C_i)]\).

(3) \text{cost}(C_{i+1})
This is the cost of fetching \(C_{i+1}\) objects for evaluating the residual predicates of \(C_{i+1}\) and joining \(C_{i+1}\) with \(C_i\). When \(C_{i+1}\) is a user-defined class, since the objects are fetched according to the physical address, the number of pages to be fetched becomes \(b(p(C_{i+1}), b(C_i), N_{i+1})\), where \(N_{i+1}\) is the number of distinct \(C_{i+1}\) objects that are to be fetched; \(N_{i+1}\) can be estimated as follows [8].

\[ N_{i+1} = n(C_{i+1}) \times \left[1 - \frac{n(C_i) - E_{i+1,i}}{n(C_i)} \right] \times \text{sel}(INP(C_{i+1})) \tag{3} \]

The subitems in Eq. (3) are explained as follows:

1. \(\binom{n(C_i)}{n(C_i)}\): the number of subsets when we choose \(n(C_i)\) objects from \(C_i\); here \(n(C_i) = n(C_i) \times \text{sel}(C_i)\).

2. \(\binom{n(C_i) - E_{i+1,i}}{n(C_i)}\): the number of subsets when we choose \(n(C_i)\) objects from \(n(C_i) - E_{i+1,i}\) objects. Here, \(E_{i+1,i}\) is the average number of \(C_i\) objects that refer to the same \(C_{i+1}\) object (i.e., sharing degree [1]) and \(n(C_i) - E_{i+1,i}\) is the number of objects in \(C_i\) that do not refer to a \(C_{i+1}\) object, say \(O\).

3. \(\frac{n(C_i) - E_{i+1,i}}{n(C_i)}\): the probability that the \(C_{i+1}\) object \(O\) is not referred to by any \(C_i\) objects.

4. \(\frac{1 - n(C_i) - E_{i+1,i}}{n(C_i)}\): the probability that the \(C_{i+1}\) object \(O\) is referred to by at least one \(C_i\) object; i.e., the probability that the \(C_{i+1}\) object \(O\) is selected.

On the other hand, when \(C_{i+1}\) is a set of OID tuples, \(p(C_{i+1})\) pages are to be fetched for each \(C_i\) object in the worst case since the OID tuples of \(C_{i+1}\) are not sorted according to the join attribute. We can avoid repeated page accesses by sorting the OIDs in the OID tuples according to the physical address before fetching individual objects.

5 Conclusions
We have presented query optimization techniques in object-oriented database systems. Proposed techniques construct the least-cost evaluation plan that fully takes advantage of path indexes. Previous literature on the path index have focused on the efficiency of the index structure itself and on the optimal (path) index configurations in physical database design.
Contributions of the paper are as follows. First, we have proposed a generalized index intersection technique adapting to the extended structure of the path index. The key idea is to use the natural join operation instead of the set intersection operation. Second, we have proposed an efficient query graph reduction algorithm that determines classes to be replaced by index scans. The process of the graph reduction is modeled by using Higraph. Third, we present a search algorithm that generates a search tree from the reduced query graph and finds the least-cost evaluation plan from the search tree based on the cost model. Our search algorithm is not based on the query transformation rules, but is based on the dynamic algorithm with branch and bound pruning. Finally, we have presented a cost model appropriate to the proposed optimization techniques. In the future, we plan to devise an algorithm that finds the optimal (path) index set from the eligible index set for the given query. We also are working on extending the proposed query optimization algorithm to handle cyclic queries with disjunctive conditions.

References


