AN INNOVATIVE APPROACH FOR OBJECT RETRIEVAL IN OBJECT-ORIENTED DATABASES: DIRECT ACCESS METHOD.

Joshi Bharti1*, Morena Rustom D2.

1Department of Information Technology, Saraswati College of Engineering, Kharghar, Navi Mumbai, India
2Department of Computer Science, Veer Narmad South Gujarat, University, Surat, India

ARTICLE INFO

Corresponding Author:
Joshi Bharti,
Department of Information Technology, Saraswati College of Engineering, Kharghar, Navi Mumbai, India.

ABSTRACT

Query processing remains one of the important challenges of Object-Oriented Database Management Systems. Cost based query optimization involves creating alternative executing plans for a given query and executing the least costly one within a cost model framework. In Object-Oriented Database Management Systems (OODBMSs) objects may store references to other objects (precomputed joins), and path expressions are used in query languages. Although the cost formulas for explicit joins and the selectivity’s of attributes and joins are well-known in the relational model, there is no similar work involving path expressions for OODBMSs. However in order to optimize object-oriented queries involving path expressions, a cost model is essential. This information is necessary for deciding whether to use pointer chasing or to convert the path expressions into explicit joins and also for deciding the execution order of path expressions. In this paper, we propose a query optimization technique that fully utilize the advantages of class Ids and object Ids in object-oriented database systems. We present a new access method, called the Direct Access Method, for supporting contained objects in object-oriented databases. We have taken the advantage of generation of class Ids and object Ids which follows particular pattern while creating the classes and objects. The result shows that this method is significantly better than the existing path index method over a wide range of parameters in terms of retrieval of data. An implementation based on the C#.NET system demonstrates the validity of the results.

INTRODUCTION

The goal of query optimization is to find an execution plan for a specific query in order to minimize the cost of executing the query. The steps involved in this process can be considered at two levels, logical query optimization (query rewriting) using semantic properties of the language in order to find expressions equivalent to the one given by the user, and physical query optimization, based on a cost model to choose the best algorithm for evaluating the query. In calculating the cost of an execution plan for object-oriented queries, estimation of the cost of forward and backward traversals in path expressions is necessary. Although executing precomputed joins is not the best in all situations, estimations of the cost of path traversals is essential to compare their costs with other possible alternatives to choose the best performing access plan. The fact that path traversals must be taken into account when deciding on an execution plan is demonstrated through the following query:

SELECT emp.name, emp.job.name, emp.dept.name
FROM Employee emp
WHERE emp.dept.plant.location='Delhi' and emp.eno=15

Assuming that there is no index in any of the extensions of the classes, this query will require one sequential scan of the extension of the Employee class and 3 disk accesses, (one to fetch the corresponding dept object, one to retrieve the plant object and the last one to fetch the job object) when the path expression is evaluated through pointer chasing. If this query is executed with an explicit join technique it will require the sequential scan of the extensions of the emp, dept, plant, and job classes, and even for very small sized extensions, this cost will exceed the cost of path traversal. Although the cost formulas for explicit joins and selectivity’s of attributes, and joins are well-known in relational model, there is no similar work in calculating the selectivity’s and costs of path expressions for OODBMSs.

In this paper, we discuss the problems with query processing in OODBSs and present the new method, Direct Access Method, which uses class-ID and object Ids for data retrieval. Let’s take one example of an aggregation hierarchy, which consists of four classes “Person” “Person_Name”, “Company” and “Vehicle”. The class Person has one simple attribute ‘age’ and two complex
attributes 'Person', 'Name', and 'Vehicle'. The Person, Name has two simple attributes 'Frame' and 'Name'. Vehicle has one simple attribute 'color' and one complex attribute 'Company'. Company has one simple attribute 'name'. Every object in an OODB is identified by an object identifier (OID). Let's consider the query example.

Query Q1: Retrieve persons whose vehicle company name is "Bajaj".

Our method will directly access the company class object and search the "Bajaj" object and store index in temp list. This list will help in accessing the persons object. No need to access vehicle class objects. It will reduce disk access, which will result in decrease in cost.

The rest of the paper is organized as follows. Section 2 summarizes the access methods proposed in the literature. Section 3 introduces the concept of our method. Finally, we conclude the paper in Section 4.

RELATED WORK

A relational database consists of a group of separated relations, which are related through primitive key values. The Join operation is used to connect these relations. In object-oriented databases, objects of various classes are related by object identifiers, which leads to the special structure of nested objects. Traversal through the bridges built upon OIDs is a natural way of evaluating OODB queries. Therefore, nested queries imply traversal of objects along the path between the target class and the nested attributes. On various traversal methods, we can see that a significant part of the query processing cost is spent on accessing intermediate objects between the target class and the nested attributes. Techniques based on indexing or signature file methods have been proposed to expedite the processing of queries. According to our observation, the essence of these techniques is to reduce physical traversals of intermediate objects between the target class and the predicate classes. The idea behind indexing techniques for nested query processing is to map a value of a certain attribute to some ancestor objects which directly or indirectly own the attribute values. The indexing mechanisms implicitly create a direct reverse link from a nested attribute to an ancestor class. As a result, the goal of bypassing the intermediate objects is achieved by scanning indexes [6]. Indexing techniques are effective and will be efficient as long as the overhead they introduce is smaller than the saving gained from avoiding intermediate object traversal. Unfortunately, most indexing techniques require costly storage overhead and expensive index maintenance. Therefore, they can't be applied on too many attributes. Only some frequently queried target classes and predicate attributes can be chosen to create indexes.

Multiple Index [5], [14] is the first of the indexing techniques for OODBs. It creates an index for each edge on the path from a nested attribute to the target class. It is like creating a reverse link for each edge along the path. To answer a query involving the indexed attribute and target class, index scans may be used to replace physical access to intermediate objects for backward traversals from the nested attribute to the objects in target class. Although several indexes are created for a given path and thus many index scans are necessary for a query evaluation, this organization is flexible for creating indexes sharing a path without introducing much duplicated overhead.

Nested Index and Path Index [5] map a specific nested attribute to the target class and to the classes located along the given path, respectively. Like multiple index, they separately create implicit reverse links from the nested attribute to the target class and the classes appearing on the path. Only one index scan is needed to reach the target classes from the nested attribute. Although both techniques are very effective, they require high storage cost and expensive update maintenance. Thus, they are very expensive when many attributes are indexed. Further, the nested index requires system-supported reverse links among objects in the path to efficiently update the index [5]. These indexing techniques cannot support all of the nested queries.

Field Replication Technique [15], as its name suggested, replicates attributes of nested objects into their ancestor objects. Therefore, nested attribute values that would normally be accessed through forward traversal are replicated such that expensive traversals may be avoided. The problems with field replication are that it imposes a structure change to the original database and that its update cost is expensive. In order to improve the update performance, inverted path was introduced to implement reverse links along the path [15]. The idea of inverted path is similar to that of multiple indexes. Therefore, this organization can be used to support backward traversals and some of the forward traversals where the nested attributes are replicated.

An access support relation [8] is a generalization of the join indices for OODBs. Instead of supporting traversal (or join) of two connected classes (relations), access support relations support the traversal along a path of arbitrary length. The relations may be created by joining all of the classes along the path and project the object identifiers from the classes on the path. Similar to join indices, two copies of an access support relation are stored and clustered correspondingly on the OIDs of objects in the two end classes of the path. Therefore, traversals from either end class of the path to any class on the path can be supported.

Direct Links [13] maintain links connecting objects in two separate classes for fast object traversal. Since objects in the intermediate classes between the target class and the predicate class usually are not directly related to the query, much computing cost will be saved if they are not accessed during query processing. Therefore, the direct links between two classes provide short cuts for object traversals. The direct links are similar to projecting the OIDs of the end classes on the access support relations. Thus, it may go from one end of the path to the other end efficiently. Moreover, in order to facilitate associative search of the direct links, indexes can be built to map attributes of either end classes to the direct links organization. Therefore, both forward and backward traversals are supported with reasonable storage overhead. However, like nested index, system-supported reverse links among objects in the path is needed to efficiently update the direct links.

Our method is using symmetric pattern generation of class ID and Object ID to directly access the data of nested class. Which will reduce all traversing from predicate class to target class and no extra storage space is required to access the data.

STRUCTURAL DESIGN OF OUR OODB

To improve extensibility we have classified the elements of the object-oriented database into three types Meta-class, Meta-object, and object [2]. All of them reside in three different spaces. The database has logical
storage structures [16] [19] [20].

Class contains required metadata for class of object to be persisted. Before actual object is persisted, metadata of its class must be persisted. Each class must have unique identity. After acquiring unique identity, all the components of class like attributes, constructors, properties, methods, and events are retrieved by using object to be persisted. While obtaining components of class, relationships of that class with other classes must also be identified and maintained. Class maintains collection to preserve metadata of elements like attributes, constructors, methods and events. It stores semantics of a set of user objects. Relationship types considered are Is-a relationship and Has-a relationship [2].

Relationship cardinality is maintained while adding relationship of any of the above three types. It can be one to many, one to one, etc. Main component of the relationship are members in the relationship (classes), type of relationship and its cardinality. Relationship is maintained in the database schema using relationship collection [17].

Object is the element used to sustain user objects and their related objects. Objects are stored in object space. Each object has unique identity. Our system maintains a unique object identifier (OID) within each Object which distinguishes one Object form other Object. The OID is system generated, the program that created the objects did not set the OIDs. The OID is usually a number, and is not visible to the user or programmer [18]. OID is invariant across all possible modifications of an object’s state, address, structure or name. OID is also used to maintain relationship of current Object with other Objects. Class Id indicates unique id of the class associated with it within the database.

Object data includes all the values of current object to be persisted and other objects related with current object. In this algorithm, it is intended to take the advantage of symmetric pattern of class ids and object ids, which are generated by our software. This method uses bottom-up approach. Thus, it will reduce the path propagation from base class to nested derived class.

In our OODBS structure Data Dictionary is created, in which each class has its associated object ids with them. B+ Tree is also there for all attributes and root address is stored in an array.

Algorithm:
1. Accept query from the user.
2. Extract the given attribute name and required attribute name.
3. Get the class name of both the attributes.
4. Find the class id of both the classes using getclassld function (refer (A)).
5. Access B+ Tree of required attribute.
6. Search the objects from given class Ids and make a set of object Ids and indexes of objects.
7. Using this set and required class name see the data dictionary and fetch the objects with the new object Ids.
8. End of the algorithm.

(A) Getting Class Id of Stored Class

To retrieve the unique class Id of the stored class GetStoredClassID method of the clsDatabase is used. In it string containing the name of class is passed. objDb.GetStoredClassID(“Person”)

(B) Getting Detail of Stored Class based on the class Id

To retrieve the details of the class based on the class Id GetClass method of clsDatabase is used. It takes class Id as argument and object of the class as result. clsClasa objClass=clsDb.GetClass(0)

(C) Getting data from the object id

To retrieve the data from the object function ReadObjectsbyObjectID(long objected) is used . It takes objectld as argument and returns details of the object.

This method will directly retrieve the attribute without going through the long path traversal. Algorithm will take less time to get the retrieval of attributes in complex environment.

Proof: Taking above class structure example

Suppose I have created three objects(X1, X2, X3) (Example of data Dictionary)

<table>
<thead>
<tr>
<th>Object Ids----&gt; Class Ids</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 0 (Attribute : age, person_name and Vehicle owns)</td>
<td>440</td>
<td>838</td>
<td>1236</td>
</tr>
<tr>
<td>Person_Name 1 (Attribute : lname,fname)</td>
<td>1A1</td>
<td>5B1</td>
<td>9DJ</td>
</tr>
<tr>
<td>Vehicle 2 (Attribute :color, company object)</td>
<td>2Red</td>
<td>6Blue</td>
<td>10Red</td>
</tr>
<tr>
<td>Company 3 (Attribute name)</td>
<td>3Bajaj</td>
<td>7Maruti</td>
<td>11Bajaj</td>
</tr>
</tbody>
</table>

Query is: Find the first name of the person whose Vehicle company name is “Bajaj”. According to our algorithm:

(1)(2) Given attribute is company name and required one is first name of the person.
(3) Class name of both the attribute is Company and Person_Name.
(4) Class Id of both the classes is 3 and 1.
(5) Access B+ tree based on given attribute company “name”.
(6) After matching set will contain only (3, 11) and index (1, 3)
(7) According to data dictionary required class Id is 1. Phi(r) will contain (1, 9)
(9) Get the data from Phi(r) = (A and D)

Thus it is proved that with the help of Class Id, Object Id data can be accessed directly without traversing the complicated path.

CONCLUSION AND FUTURE SCOPE

Our Method has used B+ Tree and class ID and Object Ids. B-tree is an index organization for Object-Oriented Databases which supports range queries indexing several classes along aggregation and inheritance hierarchies. B Tree indexing technique is compared with the other indexing techniques to overcome the limitation of the later. B-tree organization offers the best retrieval performance in most cases. It is generally required to store enormous data and retrieve data from the database in a shorter duration. It is shown that this technique performs better than most widely used indexing techniques. Our method does not require full path traversal. Directly from class id, object id...
and B+Tree indexing we can retrieve the data. It will solve many complex nesting processes easily.

Using above example following short analysis has been done, we have used following statistics
N no of objects in a class
B no of blocks containing the objects
S size of objects
F blocking factor, the no of objects of a class that fit into one block
No of blocks b = n/f
V (A, Class c) no of distinct values of A in class c.

Analysis

We have four classes:
Person   age
Person_Name  f name, lname
Vehicle                 color
Company  name

Query: Find lname of person whose company name is "Bajaj".

Let no of person objects are 100 and company objects are also 100
N (p) = 100 and N(c) = 100
S (p) = 2 bytes and S(c) = 4 bytes
F (p) = 50 and F(c) = 25
B (p) = 100/50=2 and B(c) = 100/25=4
Linear Search: 2+4= 6 Blocks read
Binary Search: if V (name, Company) = 10 then 100/10=10 objects are for “Bajaj”
    10/25 = 1 Block
    If binary search to find first record is log base 2 (4) =2
    Total 2+2+1= 5 blocks read
B+Tree If height is 1 then max 1 block read for company object
    Total 2+1 = 3 Blocks read.
We can conclude that our method will save time. This time saved is more if level of aggregation is more. Our plan is to do experimental modelling to prove our method that it is cost effective method for complicated nested queries also.

REFERENCES