Forward vs. Reverse Traversal in Path Expression Query Processing

David Taniar
Monash University - GSCIT
Australia
David.Taniar@infotech.monash.edu.au

Abstract

Path traversals have been recognised as one of the strengths of object-oriented query processing, as information retrieval can be achieved through pointer navigation. There are two existing path traversal methods, namely "forward" and "reverse traversal". In this paper, we analyse and compare the two traversal methods. Our results show that forward traversal is suitable for path expression queries involving selection operations on the start of the path expression, as the selection operations provide a filtering mechanism. Furthermore, redundant accesses to the associated objects may also be avoided indirectly through filtering. In contrast, reverse traversal is suitable for path expression queries involving selection operations at the end of path expression, since the problem of redundant accesses to the associated objects may be avoided. From our analysis, we formulated two lemmas on path traversals. These lemmas are anticipated to be used as a foundation for future query optimization of general path expression queries involving an arbitrary number of classes connected in relationships.

1 Introduction

Objects are usually not independent, and they can be connected to each other through the *aggregation/association* relationships. This relationship forms complex objects that may include objects from many different classes. A typical object-oriented query is to retrieve objects that satisfy certain predicates. These predicates may appear at any classes connected through aggregation/association relationships. Queries on these hierarchies are known as *path expression queries* [6]. Processing these queries is usually done through path traversal.

Path traversals have been recognised as one of the strengths of object-oriented query processing, as information retrieval can be achieved through pointer navigation. For 2-class path expression queries, there are two types of path traversals: *forward* and *reverse* traversal. A *mixed* traversal between forward and reverse traversals can be applied to complex path expression queries involving more than two classes.

It is the aim of this paper to present a comparative analysis between forward traversal and reverse traversal in object-oriented query processing. The results of this comparison may be used as a guideline for optimising general path expression queries. General path expression queries normally consist of more than 2 classes connected through relationships. As these queries can be built upon multiple 2-class path expressions, the results of our analysis concerning forward and reverse traversals can be applied. The discussion on path expression query optimization is, however, out of scope of this paper.

0-7695-0053-6/98 $10.00 © 1998 IEEE

The rest of this paper is organised as follows. Section 2 and 3 describe forward and reverse traversal techniques in path expression query processing. Section 4 presents a quantitative performance analysis of the two traversal methods. Section 5 presents the experimental results. Section 6 discusses related work. And finally, section 7 gives the conclusions and explains future work.

2 Forward Traversal Processing

Since path expression queries involve multiple classes along aggregation hierarchies, *forward traversal* exploits the associativity within complex objects. All associated objects connected to a root object assemble a complex object. This associative approach views a complex object as a cluster, and consequently processing these objects can be done together.

Figure 1 gives an example of class schema and its instantiations; and a forward traversal of the objects. In this paper, class *A* is referred to as a *root class*, whereas class *B* is called an *associated class*. Further, objects of a root class are *root objects*, and objects of an associated class are *associated objects*. The number of associated objects for a root object is known as the *fan-out* degree of that root object. In this example, the fan-out degree of a_1, a_2 and a_3 are equal to 2, 3 and 1, respectively. Lower case letters are used to indicate OIDs. We use a graph notation to represent class schemas and instantiations. Circles are classes, whereas quarter circles are objects. The lines represent the links/relationships among classes or object. A typical query from this schema is to select objects that satisfy some predicates of both class *A* and *B* [3, 6]. The following is the query which is expressed in OQL (Object Query Language) [4].

```
Select a
From a in A, b in a.rel
Where a.attr1 = constant AND b.attr1 = constant;
```

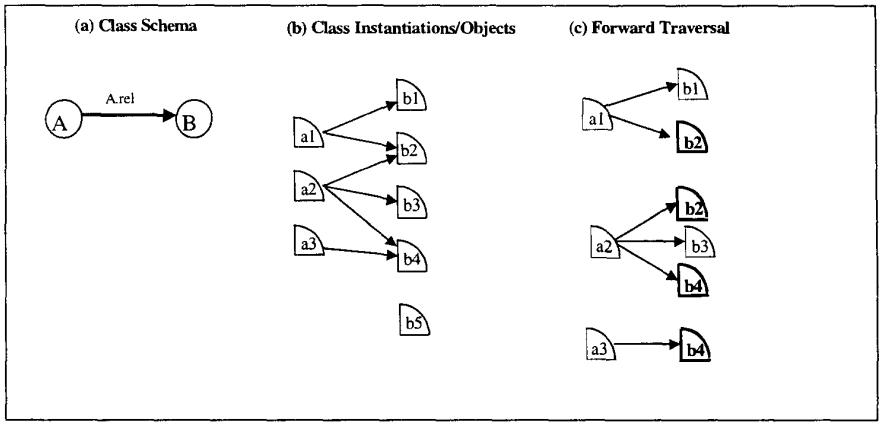


Figure 1. Class Schema, Instantiations, and Forward Traversals

Using this associative approach, objects along the association path that are not reachable from the root object will not be processed (e.g. object *b*5). This method is very attractive mainly because of the filtering feature. In this case, objects that do not form a complex object described in the query predicate are discarded naturally.

A problem of forward traversal is that when the cardinality of the association is *many-many* or *many-one*, associated objects referred by more than one root object will need to be visited more than once. In the example, object b_2 and b_4 are visited twice, each.

3 Reverse Traversal Processing

Reverse traversal consists of two phases: *selection* phase and *consolidation* phase. Reverse traversal is influenced by the concept where the predicate of each class is invoked independently regardless of the associative relationship. The selection phase contains selection operations, whereas the consolidation phase gathers the results from the selection phase for final presentation. In the selection phase, all associated objects are evaluated against the selection predicate. The non-selected associated objects will make the links to this associated object destroyed from the working space. In the consolidation phase, all root objects are read, and checked whether or not the links to the associated class exist. If exist, the object is selected and is put in the query result. Otherwise, this root object is not selected, as the corresponding associated objects have been discarded in the selection phase.

Reverse traversal is influenced by the concept of object copying used in object-oriented query processing [10]. The process is basically as follows. Initially, both classes together with the links are copied to a temporarily working space. Each selection predicate selects or destroys objects as it scans and processes each object. If an object is selected, the object remains in the working space. The links, that this object has, are untouched. However, if the object is not selected, it is removed from the working space. Removing the object also implies that the directed links that the object has are also removed. But, the associated objects are not automatically deleted, as their life times are independent to other objects. At the end, only objects that have been selected by the selection predicate exist in the working space. The final query result is basically those selected root objects that exist in the working space and still maintain at least one link out to an associated object that also exists in the working space.

Reverse traversal does not filter unnecessary objects prior to processing. Non-associated objects will be processed, although these objects will not be part of the query results. The processing performance of a class will be down graded by $(1-\alpha)$ times 100% percent, where α is a probability of an object of having an association with objects from a different class. This problem will not exist if both classes have *total participation* in the association relationship ($\alpha = 1$).

4 Quantitative Performance Evaluation

To compare performance of the two traversal methods, it is necessary, firstly, to describe the behaviour of each traversal method in terms of cost models, and secondly, to perform analytical performance comparison between the two methods. The cost models presented in this section are merely for relative performance comparison, not for estimating the query response time. Hence, common factors (such as processing unit cost) are left out. We measure the effectiveness of each model by counting the number of objects involved in the query processing. The notations used in the cost equations are given in Table 1. We assume that the data is already retrieved from the disk. This main memory based structure for high performance databases is increasingly common, especially in OODB, because query processing in OODB requires substantial pointer navigation, which can be easily accomplished when all objects present in the main memory [9, 11].

Variables	Descriptions
r_1	Number of root objects
r_2	Number of associated objects
r'_2	Number of accesses to the associated objects
λ_1	The average *fan-out* degree of the root class
σ_i	Selectivity which gives the probability (or proportion) that a given object of the *i*th class is selected.

Table 1. Notations used in the cost equations

4.1 Cost Models for Forward Traversal

The cost for forward traversal is the sum of the root class cost and the associated class cost. The root class cost is equal to the number of root objects r_1 multiply the processing unit cost *tp*.

$$r_1.tp \tag{1}$$

The processing unit cost *tp* is a cost for processing an object. There are two main components in the processing costs: *reading/loading time* and *predicate evaluation time*. The reading/loading time is influenced by the size of object, whereas the evaluation time is determined by the length of selection predicates in each class.

Processing associated objects can be accomplished by traversing each selected root object to all of its associated objects. The number of accesses to the associated objects is determined by r'_2, which is given by

$$r'_2 = r_1.\lambda_1 \tag{2}$$

The symbol r'_2 is differentiated from r_2 (the original number of objects), since some associated objects are processed/visited more than once. Redundant accesses to the associated objects occur only in *many-many* and *many-one* relationships. The original degree of redundancy is determined by the degree of coupling between the root class and the associated class, partly shown by the fan-out degree of the root class λ_1. For example, if r_1=100, r_2=200, and λ_1=5, the number of accesses to the associated objects (r'_2) is equal to 500. It shows that the redundancy factor is more than double. Moreover, if the fluctuation of λ_1 is high, the redundancy factor will even be higher.

If the root class includes a selection operation, not all associated objects will need to be accessed. The selection factor is shown by σ_1, which is the probability of a root object to be selected by the selection operation. By incorporating equation (2) with the selectivity factor σ_1, the cost for processing an associated class becomes:

$$(\sigma_1.r'_2).tp \tag{3}$$

The sum of equations (1) and (3) gives the total processing cost for a 2-class path expression using a forward traversal technique.

$$[r_1 + (\sigma_1.r'_2)]tp \tag{4}$$

For path expression involving *i* classes ($i \geq 2$), the total cost using a forward traversal becomes:

$$r_1.tp + \sum_{i=2}^{m} (\sigma_{(i-1)}.r'_i)tp \tag{5}$$

4.2 Cost Models for Reverse Traversal

The cost for reverse traversal is determined by the selection cost and the consolidation cost. The processing cost for the selection phase depends on whether there is one or two-class selection in the query. If both classes contain a selection operation, the selection cost is:

$$(r1+r2).tp \tag{6}$$

The consolidation cost varies depending on whether the query involves a selection on the root class and where the target class is. When the root class does not include a selection operation, the consolidation cost is the cost for going through all root objects which is given by:

$$r1.tp \tag{7}$$

However, when there is a selection operation in the root class, the consolidation cost will be influenced by the selectivity factor $\sigma 1$. Hence, the consolidation cost becomes:

$$\sigma 1.r1.tp \tag{8}$$

If the associated class becomes the target class (i.e., projection operation on the associated class), the consolidation cost must be added to further retrieval cost of the associated objects for projection, which is given by:

$$\sigma 1.r'2.tp \tag{9}$$

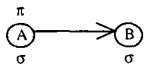
Because the selection phase and the consolidation phase shows an interdependency, in which the consolidation phase cannot start before the selection phase finishes, the total cost for reverse traversal is the sum of the selection cost and the consolidation cost.

4.3 Forward Traversal vs. Reverse Traversal

Forward traversal is not only simple but also attractive, and is particularly good when there are no redundant accesses to the associated classes. Reverse traversal, in contrast, is especially suitable for highly coupled association relationships. Since each class is processed independently, the selection process is free from any associativity independency.

It becomes essential to compare performance of forward traversal and reverse traversal. In a two-class path expression query, basically there are three different cases: 2 selections (selections on the root class and the associated class), 1 selection (a selection on the associated class), and 1 selection on the root class.

4.3.1 Case 1:




```
OQL:  Select a
      From a in A, b in a.rel
      Where a.attr = constant
      And b.attr = constant
```

We shall determine the condition, under which the cost of forward traversal is lower than that of reverse traversal, i.e.,

Forward traversal cost < *Reverse traversal cost* (10)

From equations (4), (6) and (8), condition (10) is equivalent to

$$r1 + \sigma 1.r'2 \quad < \quad r1 + r2 + \sigma 1.r1 \tag{11}$$

The processing cost *tp* has been factored out. And for $r'2 = r1.\lambda 1$, condition (11) becomes

$$\sigma 1.r'1.\lambda 1 \quad < \quad r2 + \sigma 1.r1$$

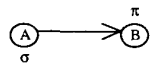
Now as for $\sigma 1.r1.\lambda 1 = x.r2$; where x represents the replication factor, the above becomes

$$\begin{aligned} & x.r2 < r2 + \sigma 1.r1 \\ \Rightarrow \quad & x.r2 < r2 + (x.r2)/\lambda 1 \\ \Rightarrow \quad & x < 1 + \frac{x}{\lambda 1}. \end{aligned} \tag{12}$$

In the case where the relationship between the root class and the associated class is *one-one*, the values of x and $\lambda 1$ are equal to 1. Therefore, condition (12) is trivially satisfied. If the relationship is *one-many*, where x=1 and $\lambda 1$=*m*, condition (12) is also satisfied since the left hand side is equal to 1 while the right hand side is greater than 1. If the relationship is *many-one* where *x*=*many* and $\lambda 1$=1, condition (12) is true since the right hand side is always 1 more than the left hand side.

In the case of *many-many* relationships, the validity of condition (12) will be determined by the values of both x and $\lambda 1$. The replication factor x is very much influenced by the selectivity degree $\sigma 1$, which serves as a filter to the whole process. Hence, the value of x is expected to be small (can even be less than 1, if the total accesses to the associated objects are smaller than the original number of objects in the associated class). If the participation of the associated class to the relationship is *partial*, the replication factor x can be greatly reduced. In these cases, condition (12) can be expected to be satisfied.

4.3.2 Case 2:




```
OQL:  Select b
      From a in A, b in a.rel
      Where a.attr = constant
```

We shall determine that the cost of forward traversal is lower than that of reverse traversal, by showing that

Forward traversal cost < *Reverse traversal cost* (13)

From equations (4), (6), (8) and (9), condition (13) is equivalent to

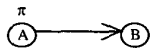
$$r1 + \sigma 1.r'2 \quad < \quad r1 + \sigma 1.r1 + \sigma 1.r'2 \tag{14}$$

Note that the selection phase consists of selection operation on the root class only. Now, condition (14) can be derived to

$$0 \quad < \quad \sigma_1.r_1.$$

In the case where the selection operation σ_1 obtains some objects r_1, the right hand side becomes positive, and condition (14) is satisfied. Consequently, condition (13) is also satisfied.

4.3.3 Case 3:




```
OQL:  Select a
      From a in A, b in a.rel
      Where b.attr = constant
```

The cost of reverse traversal is *normally* lower than that of forward traversal, i.e.,

$$\textit{Forward traversal cost} > \textit{Reverse traversal cost} \tag{15}$$

From equations (4), (6) and (7), condition (15) is equivalent to

$$r_1 + r'_2 \quad > \quad r_2 + r_1$$

Note that here, there is no selection operation on the root class. Hence, the variable σ_1 in the forward traversal cost is eliminated. Also, the selection cost in the reverse traversal consists of the selection cost for the associated class only, whereas the consolidation cost is the cost to go through all root objects. Thus, the above becomes

$$r'_2 \quad > \quad r_2$$

And for $r'_2 = r_1.\lambda_1$, we have

$$r_1.\lambda_1 \quad > \quad r_2,$$

for $\lambda_1 \geq 1$, this implies

$$\lambda_1 \quad > \quad \frac{r_2}{r_1}. \tag{16}$$

If $r_2 \leq r_1$, condition (16) is satisfied.

If $r_2 > r_1$, and if as in case 1 $\lambda_1 = \dfrac{x.r_2}{\sigma_1.r_1}$, we have

$$\lambda_1 \quad > \quad \frac{r_2}{r_1}$$

$$\Rightarrow \quad \frac{x.r_2}{\sigma_1.r_1} \quad > \quad \frac{r_2}{r_1}$$

$$\Rightarrow \quad x \quad > \quad \sigma_1$$

Since $\sigma_1 = 1$, the condition becomes

$$x \quad > \quad 1$$

If the number of accesses to the associated class is larger than the original number of associated objects, the above condition is satisfied.

Based on the three cases discussed above, two lemmas about the forward traversal and the reverse traversal are given as follows.

LEMMA 1 (FORWARD TRAVERSAL).

Forward traversal is particularly good when there is a selection operation on the root class.

LEMMA 2 (REVERSE TRAVERSAL).

Reverse traversal is suitable for path expression queries especially when filtering is not possible and the number of accesses to the associated class through path traversal from the root class is greater than the original number of associated objects.

The first lemma is concerned with cases 1 and 2, while the second lemma relates to case 3 above.

5 Experimental Performance Evaluation

The experimental environment was a Dec Alpha 2100 model running under Digital Unix operating system. The size of main memory was 2Gb. The processors are based on the 64-bit RISC technology. The 64-bit technology breaks the 2-gigabytes limitations imposed by conventional 32-bit systems. Subsequently, the usage of very large memory is common to Digital Alpha servers. Very large memory systems significantly enhance the performance of very large database applications by caching key data into memory. The experimental program was written in C++. Two-class path expressions were created.

5.1 Validation of the Cost Models

5.1.1 Validation of the Forward Traversal Cost Models

A number of experimentations have been carried out to validate the forward traversal cost models. In the experimentations, the fan-out degree of the root class is varied from 1-10. A random number generator is used to generate the fan-out degree. The selectivity degree is 1%, 5%, 10%, and 20%. The values of the selection attributes are between 1-100. They are distributed randomly to all objects. Selectivity of 1% refers to an exact match of any one value of the selection attribute. Selectivity of 5% nominates any value within a range of 5 (e.g., selection attribute<=5). Using this principle, it can be approximated an arbitrary selection degree. The unit processing cost for an object is 5.9 μsec. This includes the cost for handling the relationship. The results of using each of this selectivity degree are presented in Figure 2.

A number of observations can be made based on the experimental results. First, performance modelling through analytical models has proven to be a difficult task. It is often impossible to achieve a zero percent error rate, due to unseen overheads, which deal with lower level architecture. To show whether an analytical model is reliable, a 10% tolerance is often set. Second, the error rate is within the tolerance of 10% for all cases. For the selectivity of 20%, the average error rate is below 5%. Third, filtering is not as good as predicted. This is most probably caused by a fixed overhead, which is not influenced by the degree of selectivity. Nevertheless, the accuracy of the model is still acceptable as shown by the minimum error rate.

The results from the experimentations validate the forward traversal models as shown by the minimum error rate. Since the basic forward traversal models are justified, the analytical models can be used to perform complex queries, which employ forward traversal as the basic building block.

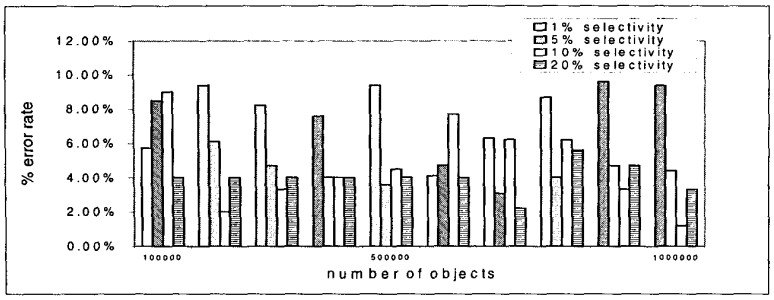


Figure 2. Validation of the Forward Traversal Cost Models

5.1.2 Validation of the Reverse Traversal Cost Models

In validating the reverse traversal cost models, a number of experimentations are carried out. In the experimentations, the associated class contains objects ranging from 50,000 to 100,000 objects. The selectivity degrees are 1%, 5%, and 10%. The fan-out degree of the root class is between 1-10, and is distributed randomly to all root objects. On average, the fan-out degree is around 4. The processing unit cost is equal to 7.6 μsec. It is higher than the processing unit cost for the forward traversal, since in the reverse traversal, the writing cost for the temporary results from the selection phase is incorporated. Figure 3 shows the experimental results.

A number of observations can be made. First, in a very few cases, the error rate is above the limit of 10%. In the majority, the error rate is well below 10%. The analytical models for the reverse traversal are shown to be quite reasonable. Second, the impact of the degree of selection on the elapsed time is not drastic, meaning that the processing cost is mainly for accessing all objects.

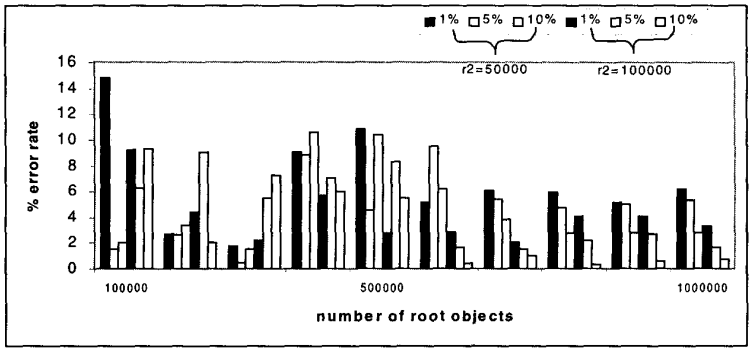


Figure 3. Validation of the Reverse Traversal Cost Models

5.2 Further Experimental Results

Further experimentations were carried out to validate the analytical comparative results of the three cases presented earlier.

5.2.1 Case 1:

Figure 4(a) shows a comparison between forward traversal and reverse traversal for query type 1 (i.e., 2 selections). Performance of forward traversal is demonstrated to be better than that of reverse traversal. As the selectivity factor increases, the cost for forward traversal also increases. This is due to the reduction of the filtering mechanism in the forward traversal. On the other hand, performance of the reverse traversal seems to be not much affected by the degree of the selectivity, since the major component of the processing is the cost for evaluating all root and associated objects.

Figure 4(b) shows that the forward traversal cost almost remains steady, until the fan-out degree is closing to a high degree. This shows that the filtering mechanism is not much affected by the fan-out of the root class, because the selection operator of the root class eliminates most of the associated objects. The trend of the reverse traversals is also similar to that of forward traversal. This is particularly because the cost is influenced by the size of the two classes. It is interesting to notice that the difference in performance between 10% replication and 20% replication of the reverse traversal is insignificant. This is due to the consolidation cost which focuses on the root class. The difference is reflected only by the selection cost of the associated class.

Overall, the results show a support for Lemma 1, where the influence of the selection operation in the root class plays an important role in bringing the forward traversal cost down.

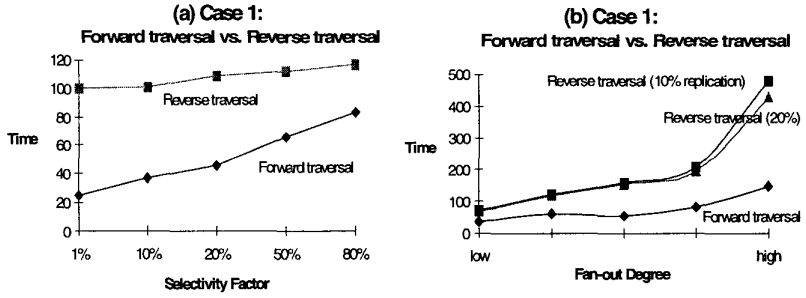


Figure 4. Case 1: Forward traversal vs. Reverse traversal

5.2.2 Case 2:

Figure 5(a) presents a comparative result for query type 2 (1 selection on the root class). Forward traversal still shows its superiority over the reverse traversal. As the selectivity factor of the root class increases, the costs for both traversal models also escalate. Moreover, performance of the reverse traversal becomes worse when the selectivity factor is more than 50%. This is because, the purpose of the consolidation process for query type 2 is to evaluate all selected root objects and their associated objects. The latter is needed, as the selected associated objects are to be projected and presented to users. This process is much influenced by the selectivity factor.

Figure 5(b) shows that the difference between the two traversal models is almost steady, regardless the fan-out degree of the root class. Performance of the two models relies heavily on the number of associated objects, which is partly shown by the fan-out degree of the root class.

It can also be concluded that for query type 2, performance of the forward traversal is better than that of reverse traversal, due to the filtering mechanism provided by the selection operator in the root class.

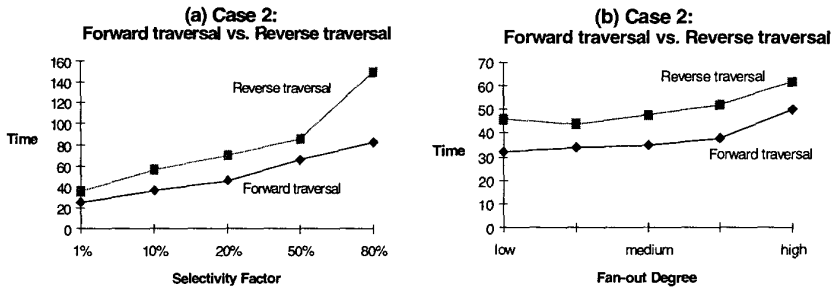


Figure 5. Case 2: Forward traversal vs. Reverse traversal

5.2.3 Case 3:

Figure 6(a) gives the results for query type 3 (1 selection on the associated class). As the selection operation is absent from the root class, the reverse traversal shows its superiority over the forward traversal, even when the size of the associated class is larger than the size of the root class. The number of accesses to the associated class significantly determines the performance of the forward traversal.

Figure 6(b) shows a comparison between the forward traversal and the reverse traversal according to the ratio between the size of the root class and the size of the associated class. The lower the associated class size, the better the performance of the reverse traversal. Performance of the reverse traversal degrades only when a lot of objects from the second class do not have any association with any root objects. This is when the associated class has a partial participation to the relationship between the root class and the associated class.

Both experimentation results shown in Figure 6 support Lemma 2, where it states, that in the absence of the filtering mechanism, forward path traversal will not enhance the performance. Therefore, the reverse traversal model is much more feasible for query type 3.

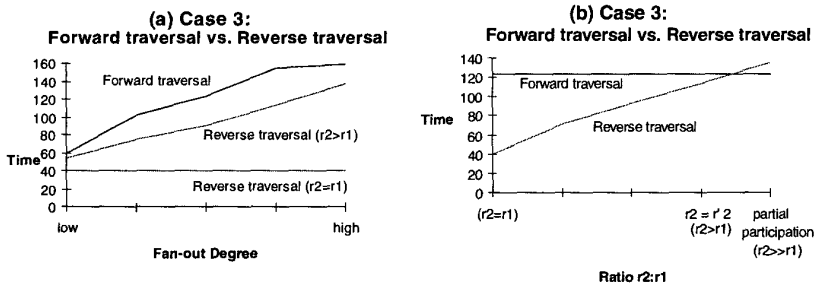


Figure 6. Case 3: Forward traversal vs. Reverse traversal

6 Related Work

Related work on path expression query processing can be divided into several categories, particularly *nested-loop* and *sort-domain* traversals, *path indices*, and *pointer-based joins*. They are described as follows.

The work by KimKC et al. [5] introduces four techniques for forward and reverse traversals, namely: *Nested-Loop Forward Traversal* (NLFT), *Sort-Domain Forward Traversal* (SDFT), *Nested-Loop Reverse Traversal* (NLRT), and *Sort-Domain Reverse Traversal* (SDRT). Extensive cost models of these four traversal techniques are reported in Bertino and Martino [2]. However, it lacks of conclusive comparison among these techniques. Our work reported in this paper has presented a comparative analysis of forward and reverse traversal, and has described two lemmas relating to the strengths of each of the traversal methods. Our aim is to bring forward rules that can be used by query optimization for optimizing complex path expression queries involving more than two classes. In optimizing such queries, the queries can be decomposed into a number of path expression sub-queries. For each sub-query, it will be determined which traversal technique is more efficient, taking the two lemmas formulated in this paper into consideration.

Path index is an index structure for path expression queries [1]. Path indices can be used to simulate reverse traversals. The reverse traversal process starts from the end class. By looking up the value of each entry of the indexed attribute, it can be determined the objects that have links to the end class. Path indices are particularly suitable for queries having a selection predicate on the indexed attribute of the end class. They were designed to simplify forward traversal technique, which is to start from a root class. If path indices are not available, traversal of such queries must start from the root object and follow the path that links to the end class. Ultimately, the selection predicate on the end class can be performed. With the availability of a path index based on the attribute of the selection predicate, there is no need for a traversal from the root class.

Path indices are designed with long path expression queries in mind. The application of such indices is also limited, since only queries having selection predicates on the indexed attribute of the end class can make use of path indices. The decision on which attribute to be indexed is known to be a classical secondary index selection problem. However, there is no doubt that when an index exists, the query that makes use of this index will be benefited.

Path expression queries are often processed by means of join, particularly *pointer-based join* [8]. The use of join in path expression queries, especially in reverse traversal, is influenced by the structure of the path expression queries expressed in OQL. Forward traversal can be easily expressed in OQL, since OQL provides a mechanism to phrase forward traversal. This is done through a dot notation. For example:

```
Select A
From A in ClassA, and B in A.rel
Where A.attr1 = constant AND B.attr1 = constant
```

The clause **`B in A.rel`** shows that the domain of relationship *rel* of *A* is pointed by *B*. This is a manifestation of forward traversal from *A* to *B*.

Reverse traversal mechanism is not explicitly declared in OQL. A reverse traversal can be expressed as a join. The above query can be re-written as follows.

```
Select A
From B in ClassB, A in ClassA
Where B in A.rel
AND A.attr1 = constant AND B.attr1 = constant
```

The query assumes that the relationship between *A-B* is where class *B* is at the *many* side (*one-many* or *many-many*) of the relationship. The clause **`B in A.rel`** checks whether each object of class *B* is a *member* of objects pointed by relationship *rel* of class *A*. If class *B* is at the *one* side, the clause **`B in A.rel`** can be written as **`B = A.rel`**. This checks whether each object of class *B* is *the same as* the object pointed by relationship *rel* of class *A*. The **`B = A.rel`** clause suggests that it is actually a join operation between class *A* and class *B*. The join is carried out between the OID pointed by relationship *rel* of class *A* and the OID of object of class *B*. Further investigation on reverse traversal in OQL can be very useful. This is reserved for future work.

7 Conclusions and Future Work

Simple path expression query processing is available in two forms: *forward traversal* and *reverse traversal*. An analysis and comparison between these traversal techniques have been described in this paper. Several points can be made which outline the feature of each traversal method.

Forward traversal is suitable for path expression queries involving a selection operation on the start of path traversal, as selection operation provides as a filtering mechanism. Redundant accesses to the associated objects may also be avoided indirectly through the filtering mechanism.

Reverse traversal is suitable for path expression queries involving a selection operation at the end of path traversal, since the problem of redundant accesses to the associated objects may be avoided through class independent processing. Because filtering is not performed, the forward traversal model is not a good choice for this particular query, and consequently, the reverse traversal model is the only option.

One main assumption used throughout this paper is that the path direction between the two classes in a path expression is *uni-directional*. A forward traversal is performed by following the path direction, that is starting from the root class and then following the path to the associated class. A reverse traversal is carried out by processing the associated class first and then the root class. The latter does not at all follow the path direction. In the case where the path is *bi-directional*, the forward traversal technique must choose where to start the pointer navigation. This matter is out of scope of this paper, and the issues regarding bi-directional paths have been discussed in our previous work (see [12]).

Understanding the strengths and weaknesses of each traversal technique is important for further work on optimization of general path expression queries involving arbitrary number of classes connected in relationships. In processing these complex queries, several steps may be involved. The *first step* is to decompose such complex queries into a number of sub-queries, in which each of the sub-queries forms a 2-class path expression. The query optimizer must formulate the decomposition process. The *second step* is to choose the most appropriate path traversal strategy for each of the sub-query. The two lemmas on path traversals established in this paper can be employed in deciding which path traversal technique to be used for each sub-query. Due to this important role of the lemmas in query optimization, it is very critical to understand the behaviour of each path traversal technique. This paper has not only described the

two path traversal methods, but also analyzed and evaluated them. Hence, these lemmas are anticipated to become a foundation for optimization of general path expression queries.

References

[1] Bertino, E., "A Survey of Indexing Techniques for Object-Oriented Database Management Systems", *Query Processing for Advanced Database Systems*, J.C.Freytag, et al. (eds.), Morgan Kaufmann, pp. 384-418, 1994.

[2] Bertino, E. and Martino, L., *Object-Oriented Database Systems: Concepts and Architectures*, Addison-Wesley, 1993.

[3] Bertino, E., et al., "Object-Oriented Query Languages: The Notion and the Issues", *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 3, 1992.

[4] Cattell, R.G.G. (editor), *The Object Database Standard ODMG-93*, Release 1.1, Morgan Kaufmann Publishers, 1994.

[5] Kim, K.C., et al., "Acyclic Query Processing in Object-Oriented Databases", *MCC Technical Report*, No: ACA-ST-287-88, 1988.

[6] Kim, W., "A Model of Queries for Object-Oriented Databases", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pp. 423-432, Amsterdam, 1989.

[7] Lanzelotte, R.S.G., et al., "Optimization of Nonrecursive Queries in OODBs", *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'91*, Munich, pp. 1-21, December 1991.

[8] Lieuwen, D.F., DeWitt, D.J. and Mehta, M., "Parallel Pointer-based Join Techniques for Object-Oriented Databases", *AT&T Technical Report*, 1993.

[9] Litwin, W. and Risch, T., "Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates", *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 517-528, December 1992.

[10] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1988

[11] Moss, J.E.B., "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 657-673, August 1992.

[12] Taniar, D., and Rahayu, J.W., "Query Optimization Primitive for Path Expression Queries via Reversing Path Expression Direction", *Journal of Computing and Information*, 1998 (to appear).