Optimizing Object-Oriented Collection Join Queries through Path Traversals

David Taniar

Department of Computer Science
RMIT University
Australia

taniar@cs.rmit.edu.au

Abstract

Path traversals have been recognized as one of the strengths of object-oriented query processing as object retrieval can be carried out through pointer navigation. Apart from path traversals, explicit join between objects is sometimes necessary in Object-Oriented Databases (OODB), due to the absence of pointer connections or the need for value matching between objects. Like in Relational Databases (RDBMS), join operations are still, if not more, expensive in OODB. In this paper, we propose an optimization strategy for object-oriented collection join queries through the use of path traversals. Our performance evaluation shows that optimization of collection join queries gains some performance benefits.

1. Introduction

Query optimization plays an important role in database systems, without which performance of database systems will not yield very significant improvement. Conventional optimization techniques used in relational database systems were not designed to cope with heterogenous structures and of particular not suitable to handle collection objects [4].

One of the differences between relational and object-oriented databases (OODB) is attributes in OODB can be of a collection type, such as sets, lists, etc, as well as simple types, such as string, numbers, etc. Consequently, explicit join queries in OODB may be based on collection attributes, which we call 'collection join' [14]. Explicit join processing is known to be very expensive not only in relational databases, but also in OODB, and therefore should be avoided whenever possible.

In this paper, we propose an optimization method for collection join queries by exploiting path traversals. Path traversals have been recognized as one of the main benefits of object-oriented query processing because object retrieval can be performed through pointer navigation [1]. Optimization of collection join is done by transforming collection join queries into path expressions. In other words, the optimized query will use pointer navigation and path traversals, instead of join.

The work presented in this paper is actually part of a larger project on object-oriented query optimization. Optimization of collection join is the third and the final stage of the project. The first two stages dealt with optimization of path expression queries, where in stage one efficient path traversal direction is formulated, and in stage two rules for forward and reverse traversals are materialized. We have reported some of the results obtained from the first two stages at ICCI'98 (stage one) [13], and TOOLS 28 (stage 2) [15].

The rest of this paper is organized as follows. Section 2 gives a brief overview of collection join queries in OODB. Section 3 explains two different path traversals, particularly forward and reverse traversals which become the basis for the optimization of collection join queries. Section 4 describes our optimization algorithms which transform collection join operations into path traversals. Section 5 presents some performance evaluation. Section 6 discusses related work, and finally Section 7 gives the conclusions and explains future work.

2. Object-Oriented Collection Join Queries: A Brief Overview

Collection join queries are join queries based on collection attributes. One of the most common collection join predicates is to check whether there is an intersection between the two collection join attributes. This collection join is known as 'collection-intersect' join [11]. An *intersect* predicate can be written by applying an intersection between the two collections and comparing the intersection result with an empty set. It is normally in a form of `(attr1 intersect attr2) != set(nil)`. Attributes `attr1` and `attr2` are collections, not simple type. Suppose the attribute *editor-in-chief* of class *Journal* and the attribute *program-chair* of class *Proceedings* are collection of *Person*. An example of a collection join is to retrieve pairs of Journal and Proceedings, where at least one of the program-chairs of a conference is an editor-in-chief of a journal. The query expressed in OQL (Object Query Language) [3] can be written as follows:

```
Select A, B
From A in Journal, B in Proceedings
Where (A.editor-in-chief intersect B.program-chair) != set(nil)
```

As clearly seen that the intersection join predicates involve the creation of intermediate results through an *intersect* operator. The result of the join predicate cannot be determined without the presence of the intermediate collection result. This predicate processing is certainly not efficient. Therefore, optimization algorithms for efficient processing of such queries are critical if one wants to improve query processing performance. Our primary intention for optimization of such a query is to solve the inefficiency imposed by the original join predicates.

3. Path Traversals

There are two kinds of path traversals, namely *forward* and *reverse traversals* [2, 5]. Basically, path traversals are an operation whereby an object is visited and evaluated, followed by objects of another class associated with the first object visited and evaluated. The difference between the two path traversal methods is in the direction of traversal (object visitation and evaluation), in which forward traversals follow the direction of the relationship from the first class to the associated class, and reverse traversals start from the opposite direction by countering the specified path direction.

3.1. Forward Traversals

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 a class schema and its instantiations; and a forward traversal of the objects. Using a diagrammatic notation, nodes and directed arcs illustrate classes, and path directions, respectively. In the example, the two classes to be traversed are *A* and *B*. To differentiate classes from objects, we use quarter circles for objects. Each object has an OID shown by the lower case letters (eg, a_1, a_2, b_1). A forward traversal of this example is a_1, b_1, b_2, a_2, b_2, b_3, b_4, a_3, and b_4. It is clearly that forward traversal is an *object-based traversal*, as after one object *a*, and all of its associated objects *b*s are visited (possibly using a depth-first search) and evaluated, then another object *a* is evaluated using the same approach. This process is repeated for all objects *a*s.

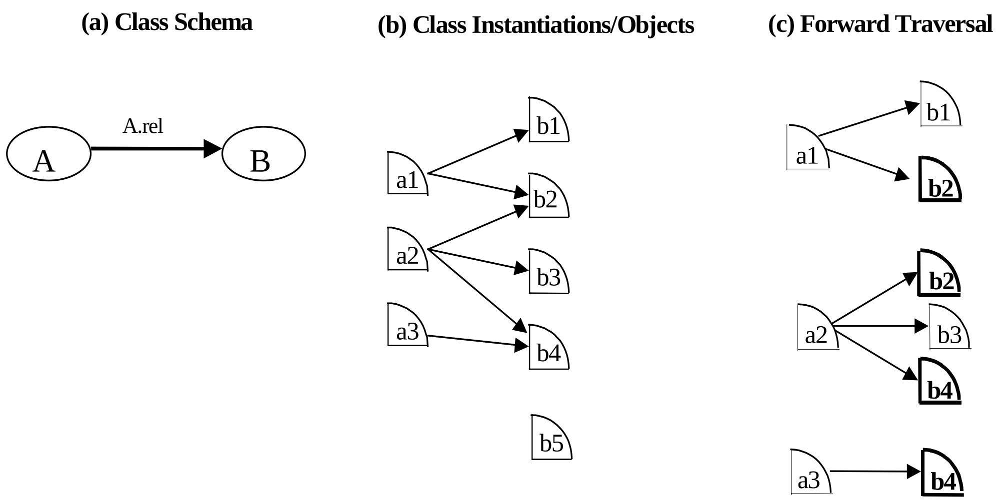

Figure 1. Class Schema, Instantiations, and Forward Traversals

Using this associative approach, objects along the association path that are not reachable from the root object (eg. object b_5) will not be processed. This method is very attractive mainly because of the filtering feature. In this case, objects that do not form a complex object are discarded naturally.

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

3.2. Reverse Traversals

Reverse traversal starts the traversal from the end class and backs up to the root class. Using the example shown previously in Figure 1, the traversal starts from class *B* and then to class *A*. However, unlike forward traversal which is an object-based traversal, *reverse traversal* is a class-based traversal. It means that the traversal starts by visiting and evaluating all objects of class *B*, and then all objects of class *A*. Hence, a reverse traversal of the previous example is

b_1, b_2, b_3, b_4, b_5, a_1, a_2, a_3, and a_4. Reverse traversal does not filter unnecessary objects prior to processing. Non-associated objects (eg. object b_5) will be processed, although these objects will not be part of the query results. This problem will not exist if both classes have *total participation* in the relationship.

Reverse traversal is influenced by the concept of object copying used in object-oriented query processing [9]. 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 normally 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.

Generally, *reverse traversals* are suitable for path expression where there is a selection operation at the end class, whereas *forward traversals* are suitable if there is a filtering process imposed by the root class [15].

4. Transformation Process

In this section, we are going to explain how collection join queries are transformed into path traversals. Since there are two types of path traversals: forward and reverse traversals, we will explain two types of transformations, particularly transformation of collection join to forward traversals and to reverse traversals.

4.1. Collection Join to Forward Traversal Transformation

A collection join operation on a class domain is actually formed by two forward traversals meeting at the joining class. When *Journal* is to be joined with *Proceedings* on *Person*, we have two forward traversals: *Journal-Person* and *Proceedings-Person*, and then perform a join. A transformation from a collection join operation to a forward traversal may be achieved by changing the direction of one of the paths, so that a complete forward path traversal can be formed. The main restriction of such transformation is that at least one of the paths must be bi-directional.

Figure 2 gives an illustration of the transformation process. The shaded nodes indicate the starting nodes of the traversals. In the initial query (join query), the two forward traversals (FT) start from *A* and *C*, each, to *B*. In the transformed query (path navigation), there is only one forward traversal, which starts from *A*, to *B*, and then to *C*.

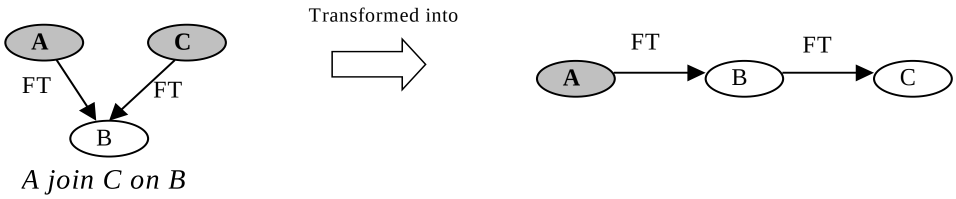

Figure 2. Collection Join Transformation to Forward Traversal

4.2. Collection Join to Reverse Traversal Transformation

A collection join to reverse traversal transformation may be applied to collection join queries where the two paths are uni-directional. In the case where an inverse relationship exists, transformation to a forward traversal is preferable, as unreachable associated objects along the path are discarded naturally. In this transformation we consider three cases, particularly a selection operation exists in the join class, selection operations exist in the root classes, and no selection operation is involved (see Figure 3).

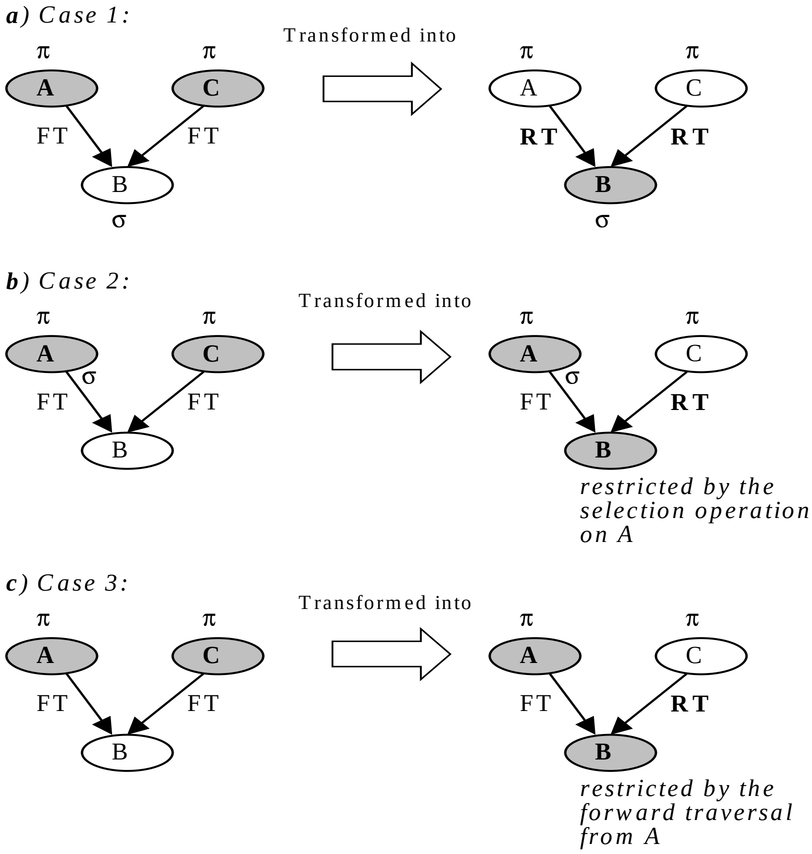

Figure 3. Collection Join Transformation to Reverse Traversal

Case 1: Since the join class contains a selection operation, path traversal should start from this class. Because the paths are uni-directional, both paths are done in a reverse traversal (RT).

Case 2: Since a root class contains a selection operation, that path should perform a forward traversal operation. Although originally the join class does not contain a selection operation, due to the selection operation done by the forward traversal, the filtering has been carried out indirectly to the join class. Therefore, the second path is performed in a reverse traversal operation. The 'to reverse traversal' transformation is actually a mixed traversal where one path is a forward traversal (FT), and the other is a reverse traversal (RT).

Case 3: Like case 2, one of the paths is carried out in a forward traversal. The join class is now restricted by the forward traversal operation done to one of the paths. The other path is then carried out in a reverse traversal operation.

Case 2 and case 3 have shown the effect of indirect filtering through selection operations on previous classes and through forward traversal operations. Also in these two cases, an optimization of collection join by transforming it to a *mixed* traversal is demonstrated.

5. Performance Evaluation

The implementation 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. 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 [8, 10]. The experimental program was written in C++. In the experimentations, the class size varies from 100 to 500 objects. The average collection size is 3 objects.

5.1. Performance Measurement of Collection Join to Forward Traversals

As stated above, the transformation to forward traversal is applicable to collection join queries and is done by changing one of the paths so that complete path expressions are formed.

Figure 4 shows that at all times collection join operation is much more expensive than path expression operation through the transformation optimization. As the join selectivity degree increases, processing costs for both operations also increase. The increase in processing cost of the collection join operation is caused by the additional comparison of the elements of the collection join attributes. Likewise, the increase in processing cost of the forward traversal is due to the traversal cost imposed by the inverse relationship where more objects need to be accessed. With a lower selectivity degree, most path traversal do not form a complete path traversal.

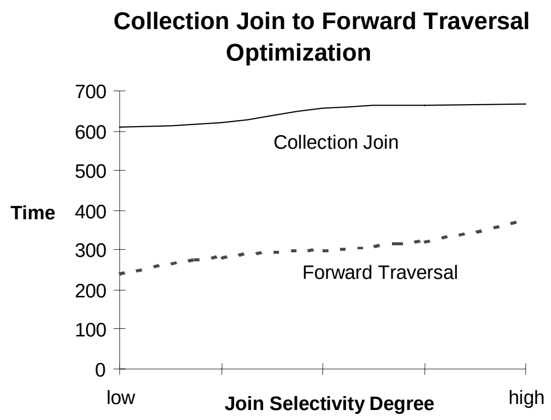

Figure 4. Performance of the Collection Join to Forward Traversal Transformation

5.2. Performance Measurement of Collection Join to Reverse Traversals

The collection join to reverse traversal transformation is applicable when it is impossible to do a collection join to forward traversal transformation due to the absence of an inverse relation. The objective remains the same; that is to avoid join operation whenever possible. In the previous section, we describe three cases of this transformation: case 1 is a transformation to two reverse traversals, and cases 2 and 3 are a transformation to a mixed traversals (one forward traversal and one reverse traversal). Since the last two cases are a mixed traversal, we only present a performance evaluation of cases 1 and 2 only.

Case 1: Figure 5 shows that the selectivity degree does not affect performance of the reverse traversal since all objects from the three classes need to be accessed. Performance of the collection join operation, however, is affected by the increase of the selectivity degree which incurs additional cost for comparison of the elements of the collection join attributes.

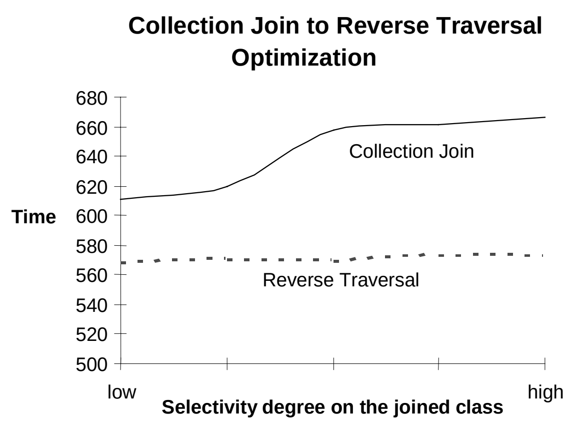

Figure 5. Performance of the Collection Join to Reverse Traversal Transformation – Case 1

Case 2: Figure 6 shows that when the selectivity degree is low, performance of the collection join is quite good. This reflects that the join cost is affected by the size of classes to be joined. If one of the classes is very small, the join cost will not be that expensive. As the size of the class having the selection grows (due to the increase of the selectivity degree), the join cost expands. The selectivity degree also plays an important role after the transformation. The degree of selectivity of the forward traversal also brings the total processing cost down, as the processing cost for the reverse traversal is reduced by the filtering mechanism carried out by the forward traversal.

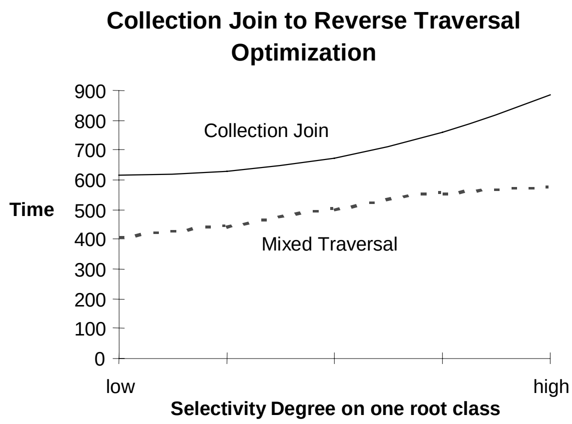

Figure 6. Performance of the Collection Join to Reverse Traversal Transformation – Case 2

6. Related Work

Collection join queries are unique to OODB, and have not been explored widely in the literature. In fact, a taxonomy for collection join is first proposed by the author and published in TOOLS PACIFIC 1996 [14]. The taxonomy is then revised and published in 1999 [11]. Subsequently, there is no pre-existing work in collection join processing, let alone the optimization. However, for a completeness of this paper, we will discuss published work which may be related to collection join processing, or explicit join in OODB in general. The first related work we would like to mention is pointer-based join, and the second in our previous work on formulating special parallel algorithms for collection join queries.

6.1. Pointer-Based Join

A number of pointer-based join algorithms (i.e. hash-loops, probe-children, hybrid-hash) have been proposed [7]. All of them are based on hash join. Pointer-based join algorithms were designed for an implicit join between two associated classes. Hence, it is a strategy for path expression queries not explicit joins. Pointer-based join gives an alternative processing strategies to path expression queries, and should not merely rely on path traversals. Path expression queries involving multiple classes are decomposed into multiple binary operations and each operation is a pointer-based join.

The main difference between pointer-based join and path traversal is that pointer-based join is a binary operation in which the path direction is not taken into account, whereas path traversal makes use of traversal direction in processing the query. Using the pointer-based join approach, query optimization is carried out by operations permutation (modification) and expansion [6]. On the other hand, path traversal is based on pointer navigation by following the path in a forward or a reverse direction.

Pointer-based join is also designed for non-collection join. Hence, it is not applicable to collection join, without necessary modifications to the original algorithms. In this paper, our optimization of collection join is done through operation transformation, not through an introduction of new join algorithms.

6.2. Special Parallel Algorithms for Collection Join

We have initiated and proposed special algorithms for collection join queries for *Parallel Object-Oriented Databases* [12]. These algorithms are based on sort-merge and hash join algorithms, which have been modified to suit collection join properties. The are algorithms are complex in nature, due to the complexity of the collection join predicates itself. Furthermore, the algorithms are designed especially to run in parallel machines, and thus limiting the applicability in general. Based on this experience, we feel the need for exploring other possibilities for collection join processing, particularly through the use of path traversal.

7. Conclusions and Future Work

Optimization of collection join queries in OODB can be achieved through a transformation to path traversals. As there are two types of path traversals, particularly *forward* and *reverse traversals*, we presented two transformation techniques: the first is a transformation of collection join to forward traversals, and the second is a transformation to reverse traversals.

The transformation to forward traversals is achieved by changing the direction of one of the paths. As a result, the collection join, which is a join of two forward traversals, is turned into a single forward traversal.

The transformation to reverse traversals is carried out by changing one of the forward traversals in the join to become a reverse traversal. As a result, the transformed query is actually a mixed traversal with one forward traversal and one reverse traversal. Since there is no change of path direction, this transformation does not require any path to be bi-directional. Hence, it is less restrictive compared to the transformation to forward traversals.

Generally, the tasks of query optimization can be highlighted into two major areas, particularly basic operation optimization, and access plans. We have identified that basic operations of object-oriented queries comprise path traversals and collection join [11]. The three-stage query optimization project mentioned in the beginning of this paper basically focuses on basic operation optimization, which includes optimization of path traversals [13, 15] and optimization of collection join queries (this paper). Our future work is to extend this project by focusing the optimization of the second major task namely access plan. We expect that the results the optimization of path traversals and collection join will become a foundation for optimization for general object-oriented queries.

References

[1] Banerjee, J., Kim, W. and Kim, K-C., "Queries in Object-Oriented Databases", *Proceedings of the Fourth International Conference on Data Engineering*, pp. 31-38, February 1988.

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

[3] Cattell, R.G.G. (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

[4] Cluet, S. and Delobel, C., "A General Framework for the Optimization of Object-Oriented Queries", *Proceedings of the ACM SIGMOD Conference*, pp. 383-392, 1992.

[5] Kim, W (1990) *Introduction to Object-Oriented Databases*, MIT Press, Cambridge, Mass.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] Taniar, D. and Rahayu, J.W., "Chapter 5: A Taxonomy for Object-Oriented Queries", *Current Trends in Database Technology*, A. Dogac, M.T.Ozsu, and O.Ulusoy (eds.), Idea Group Publishing, 1999.

[12] Taniar, D. and Rahayu, J.W., "Collection-Intersect Join Algorithms for Parallel Object-Oriented Database Systems", *Euro-Par'98 Parallel Processing*, Springer-Verlag LNCS 1470, D.Pritchard and J.Reeve (eds.), pp. 505-512, 1998.

[13] Taniar, D. and Rahayu, J.W., "Query Optimization Primitive for Path Expression Queries via Reversing Path Traversal Direction", *Proceedings of the Ninth International Conference on Computing and Information ICCI'98*, Winnipeg, Manitoba, Canada, pp. 69-76, 1998.

[14] Taniar, D. and Rahayu, W., "Object-Oriented Collection Join Queries", *Proceedings of the International Conference on Technology Object-Oriented Languages and Systems TOOLS PACIFIC'96*, Melbourne, pp. 115-125, 1996.

[15] Taniar, D., "Forward vs. Reverse Traversal in Path Expression Query Processing", *Proceedings Technology of Object-Oriented Languages and Systems TOOLS 28*, IEEE Computer Society Press, pp. 127-140, 1998.