Parallel Sub-collection Join Algorithm for High Performance Object-Oriented Databases

David Taniar° J. Wenny Rahayu+

° Monash University - GSCIT, Churchill, Vic 3842, Australia
+ La Trobe University, Dept. of Computer Sc. & Comp. Eng., Bundoora, Vic 3083, Australia

1 Introduction

In *Object-Oriented Databases* (OODB), although path expression between classes may exist, it is sometimes necessary to perform an explicit join between two or more classes due to the absence of pointer connections or the need for value matching between objects. Furthermore, since objects are not in a normal form, an attribute of a class may have a collection as a domain. Collection attributes are often mistakenly considered merely as set-valued attributes. As the matter of fact, *set* is just one type of collections. There are other types of collection. The Object Database Standard *ODMG* (Cattell, 1994) defines different kinds of collections: particularly *set, list/array*, and *bag*. Consequently, object-oriented join queries may also be based on attributes of any collection type. Such join queries are called *collection join queries* (Taniar and Rahayu, 1996).

Our previous work reported in Taniar and Rahayu (1996, 1998a) classify three different types of collection join queries, namely: *collection-equi join*, *collection-intersect join*, and *sub-collection join*. In this paper, we would like to focus on sub-collection join queries. We are particularly interested in formulating a parallel algorithm based on the sort/merge technique for processing such queries. The algorithms are non-trivial to parallel object-oriented database systems, since most conventional join algorithms (e.g. hybrid hash join, sort-merge join) deal with single-valued attributes and hence most of the time they are not capable of handling collection join queries without complicated tricks, such as using a loop-division (repeated division operator).

Sub-collection join queries are queries in which the join predicates involve two collection attributes from two different classes, and the predicates check for whether one attribute is a sub-collection of the other attribute. The sub-collection predicates can be in a form of *subset*, *sublist*, *proper subset*, or *proper sublist*. The difference between proper and non-proper is that the proper predicates require both join operands to be properly sub-collection. That means that if both operands are the same, they do not satisfy the predicate. The difference between subset and sublist is originated from the basic different between sets and lists (Cattell, 1994). In other words, subset predicates are applied to sets/bags, whereas sublists are applied to lists/arrays.

2 Parallel Sort-Merge Join Algorithm for Sub-collection Join

Parallel join algorithms are normally decomposed into two steps: *data partitioning* and *local join*. The partitioning strategy for the parallel sort-merge sub-collection join query algorithm is based on the *Divide and Partial Broadcast* technique.

The Divide and Partial Broadcast algorithm proceeds in two steps. The first step is a *divide* step, where objects from both classes are divided into a number of partitions. Partitioning of the first class (say class *A*) is based on the first element of the collection (if it is a list/array), or the smallest element (if it is a set/bag). Partitioning the second class (say class *B*) is exactly the opposite of the first partitioning, that is the partitioning is now based on the last element (lists/arrays) or the largest element (sets/bags).

The second step is the *broadcast* step. In this step, for each partition *i* (where *i*=1 to *n*) partition *Ai* is broadcasted to partitions *Bi* .. *Bn*. In regard to the load of each partition, the load of the last processor may be the heaviest, as it receives a full copy of *A* and a portion of *B*. The load goes down as class *A* is divided into smaller size (e.g., processor 1). Load balanced can be achieved by applying the same algorithm to each partition but with a reverse role of *A* and *B*; that is, divide *B* based on the first/smallest value and partition *A* based on the last/largest value in the collection.

After data partitioning is completed, each processor has its own data. The join operation can then be carried out independently. The local joining process is made of a simple sort-merge and a nested-loop structure. The sort operator is applied to each collection, and then a nested-loop construct is used in joining the objects through a merge operator. The algorithm uses a nested-loop structure, because of not only its simplicity but also the need for all-round comparisons among all objects.

In the merging process, the original join predicates are transformed into predicate functions designed especially for collection join predicates. Predicate functions are the kernel of the join algorithm. Predicate functions are boolean functions which perform the predicate checking of the two collection attributes of a join query. The join algorithms use the predicate functions to process all collections of the two classes to join through a nested-loop. Since the predicate functions are implemented by a merge operator, it becomes necessary to sort the collections. This is done prior to the nested-loop in order to avoid repeating the sorting operation.

References

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

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

Taniar, D. and Rahayu, J.W., "A Taxonomy for Object-Oriented Queries", a book chapter in *Current Trends in Database Technology*", Idea Group Publishing, 1998a (in press).

Taniar, D. and Rahayu, J.W., "Parallel Collection-Equi Join Algorithms for Object-Oriented Databases", *Proceedings of International Database Engineering and Applications Symposium IDEAS'98*, IEEE Computer Society Press, Cardiff, UK, July 1998b (to appear).