Collection-Intersect Join Algorithms for Parallel Object-Oriented Database Systems

David Taniar[1] and J. Wenny Rahayu[2]

[1] Monash University - GSCIT, Churchill, Vic 3842, Australia
David.Taniar@fcit.monash.edu.au
[2] La Trobe University, Dept. of Computer Sc. & Comp. Eng.,
Bundoora, Vic 3083, Australia
wenny@latcs1.lat.oz.au

Abstract. One of the differences between relational and object-oriented databases (OODB) is that attributes in OODB can of a collection type (e.g. sets, lists, arrays, bags) as well as a simple type (e.g. integer, string). Consequently, explicit join queries in OODB may be based on collection attributes. One form of collection join queries in OODB is collection-intersect join queries, where the joins are based on collection attributes and the queries check for whether there is an intersection between the two join collection attributes We propose two algorithms for parallel processing of collection-intersect join queries. The first one is based on sort-merge, and the second is based on hash. We also present two data partitioning methods (i.e. simple replication and "divide and partial broadcast") used in conjunction with the parallel collection-intersect join algorithms. The parallel sort-merge algorithm can only make use of the divide and partial broadcast data partitioning, whereas the parallel hash algorithm may have a choice which of the two data partitioning to use.

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* [1] 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* [9]. Our previous work reported in [9] classified 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 collection-intersect join queries. We are particularly interested in formulating parallel algorithms for processing such queries. The algorithms are non-trivial to

D. Pritchard, J. Reeve (Eds.): Euro-Par'98, LNCS 1470, pp. 505–512, 1998.

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 suitable to handle collection join queries.

Collection-intersect join queries are queries that join two classes based on an attribute of a collection type. The join predicates check for whether there is an intersection between the two collection join attributes. An *intersect* predicate can be written by applying an intersection between the two sets 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 of type set. If one or both of them are of type *bag*, they must be converted to sets. Suppose the attribute *editor-in-chief* of class *Journal* and the attribute *program-chair* of class *Proceedings* are of type sets of *Person*. An example of a collection-intersect join is to retrieve pairs of Journal and Proceedings, where the program-chairs of a conference are intersect with the editors-in-chief of a journal. The query expressed in OQL (Object Query Language) [1] 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. In a collection-intersect join query, the original subset predicate has to produce an intermediate set, before it can be compared with an empty set. This process checks for the smaller set twice: one for an intersection, the other for an equality comparison. Therefore, optimization algorithms for efficient processing of such queries are critical if one wants to improve query processing performance. In this paper, we present two parallel join algorithms for collection-intersection join queries. The primary intention is to solve the inefficiency imposed by the original join predicates.

An interest in *parallel OODB* among database community has been growing rapidly, following the popularity of multiprocessor servers and the maturity of OODB. The emerging between parallel technology and OODB has shown promising results [3], [5], [7], [11]. However, most research done in this area concentrated on path expression queries with pointer chasing. Explicit join processing exploiting collection attributes has not been given much attention.

The rest of this paper is organized as follows. Section 2 explains two data partitioning methods for parallel collection-intersect join algorithms. Section 3 describes a proposed parallel join algorithm for collection-intersect join queries based on a sort-merge technique. Section 4 introduces another algorithm, which is based on a hash technique. Finally, section 5 draws the conclusions and explains the future work.

2 Data Partitioning

Parallel join algorithms are normally decomposed into two steps: *data partitioning* and *local join*. Data partitioning creates parallelism, as it divides the data to multiple processors, so that the join can then be performed locally in each processor without interfering others. For collection-intersect join queries, it is not possible to have non-overlap partitions, due to the nature of collections which may be overlapped. Hence, some data needs to be replicated. Two non-disjoint partitioning methods are proposed. The first is a simple replication based on the value of the element in each collection. The second is a variant of Divide and Broadcast [4], called "*Divide and Partial Broadcast*".

2.1 Simple Replication

Using a simple replication technique, each element in a collection is treated as a single unit, and is totally independent of other elements within the same collection. Based on the value of an element in a collection, the object is placed into a particular processor. Depending on the number of elements in a collection, the objects that own the collections may be placed into different processors. When an object is already placed at a particular processor based on the placement of an element, if another element in the same collection is also to be placed at the same place, no object replication is necessary.

As a running example, consider the data shown in Figure 1. Suppose class A and class B are *Journal* and *Proceedings*, respectively. Both classes contain a few objects shown by their OIDs (e.g., objects a to i are Journal objects and objects p to w are Proceedings objects). The join attributes are *editor-in-chief* of Journal and *program-chair* of Proceedings; and are of type collection of *Person*. The OID of each person in these attributes are shown in the brackets. For example a(250,75) denotes a Journal object with OID a and the editors of this journal are Persons with OIDs 250 and 75.

Figure 2 shows an example of a simple replication technique. The **bold** printed elements are the elements which are the basis for the placement of those objects. For example, object $a(250, \mathbf{75})$ in processor 1 refers to a placement for object a in processor 1 because of the value of element 75 in the collection. And also, object $a(\mathbf{250}, 75)$ in processor 3 refers to a copy of object a in processor 3 based on the first element (i.e., element 250). It is clear that object a is replicated to processors 1 and 3. On the other hand, object $i(80, 70)$ is not replicated since both elements will place the object at the same processor, that is processor 1.

2.2 Divide and Partial Broadcast

The Divide and Partial Broadcast algorithm, shown in Figure 3, proceeds in two steps. The first step is a *divide* step, and the second step is a *partial broadcast* step. We divide class B and partial broadcast class A. The *divide* step is explained as follows. Divide class B into n number of partitions. Each partition of class B is placed in a separate processor (e.g. partition $B1$ to processor 1, partition

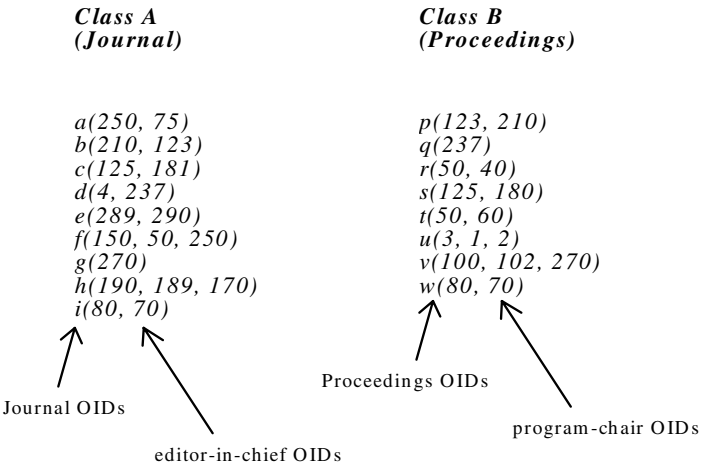

Fig. 1. Sample data

Class A	Class B	
a(250, **75**) d(**4**, 237) f(150, **50**, 250) i(**80**, 70)	r(**50**, 40) t(**50**, 60) u(**3**, 1, 2) w(**80**, 70)	Processor 1 (range 0-99)
b(210, **123**) c(**125**, 181) f(**150**, 50, 250) h(**190**, 189, 170)	p(**123**, 210) s(**125**, 180) v(**100**, 102, 270)	Processor 2 (range 100-199)
a(**250**, 75) b(**210**, 123) d(4, **237**) e(289, **290**) f(150, 50, **250**) g(**270**)	p(123, **210**) q(**237**)	Processor 3 (range 200-299)

Fig. 2. Simple replication

$B2$ to processor 2, etc). Partitions are created based on the largest element of each collection. For example, object $p(123, 210)$; the first object in class B, is partitioned based on element 210, as element 210 is the largest element in the collection. Then, object p is placed on a certain partition, depending on the partition range. For example, if the first partition is ranging from the largest element 0 to 99, the second partition is ranging from 100 to 199, and the third partition is ranging from 200 to 299, then object p is placed in partition $B3$, and subsequently in processor 3. This is repeated for all objects of class B.

```
Procedure DividePartialBroadcast
Begin
  Step 1 (divide):
    1. Divide class B based on the largest element in each collection.
    2. For each partition of B (i = 1, 2, ..., n)
         Place partition Bi to processor i
       End For

  Step 2 (partial broadcast):
    3. Divide class A based on the smallest element in each collection.
    4. For each partition of A (i = 1, 2, ..., n)
         Broadcast partition Ai to processor i to n
       End For
End Procedure
```

Fig. 3. Divide and Partial Broadcast Algorithm

The *partial broadcast* step can be described as follows. First, partition class A based on the smallest element of each collection. Clearly, this partitioning method is exactly the opposite of that in the divide step. Then for each partition Ai where i=1 to n, broadcast partition Ai to processors i to n. This broadcasting technique is said to be partial, since the broadcasting goes down as the partition number goes up. For example, partition $A1$ is basically replicated to all processors, partition $A2$ is broadcast to processor 2 to n only, and so on. In regard to the load of each processor, the load of the last processor may be the heaviest, as it receives a full copy of class A and a portion of class 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* A and *partial broadcast* B. It is beyond the scope of this paper to evaluate the Divide and Partial Broadcast partitioning method. This has been reserved for future work. Some preliminary results have been reported in [10].

3 Parallel SORT-MERGE Join Algorithm

The partitioning strategy for the parallel sort-merge collection-intersect join query algorithm is based on the *Divide and Partial Broadcast* technique. The use of the Divide and Partial Broadcast is attractive to collection joins because of the nature of collections where disjoint partitions without replication are often not achievable. After data partitioning is completed, each processor has its own data. The join operation can then be done independently. The overall query results are the union of the results from each processor.

In the local joining process, each collection is sorted. Sorting is done within collections, not among collections. After the sorting process is completed, the merging process starts. We use a nested loop structure to compare the two collections from the two operand objects. Figure 4 shows a parallel sort-merge join algorithm for collection-intersect join queries.

```
Program Parallel-Sort-Merge-Collection-Intersect-Join
Begin
  Step 1 (data partitioning):
     Call DividePartialBroadcast

  Step 2 (local joining): In each processor
    a. Sort phase
       For each object a(c1) and b(c2) of class A and B, respectively
         Sort collection c1 and c2
       End For
    b. Merge phase
       For each object a(c1) of class A
         For each object b(c2) of class B
           Merge collection c1 and c2
           If TRUE Then
             Concatenate objects a and b into query result
           End If
         End For
       End For
End Program
```

Fig. 4. Parallel Sort-Merge Collection-Intersect Join Algorithm

4 Parallel HASH Join Algorithm

In this section we introduce a parallel join algorithm based on a *hash* method for collection-intersect join queries. Like the previous Parallel Sort-Merge algorithm, Parallel-Hash algorithm is also divided into data partitioning and local join phases. However, unlike the parallel sort-merge algorithm, data partitioning

for parallel hash algorithm is available in two forms: *Divide and Partial Broadcast* and *Simple Replication*. Once data partitioning is complete, each processor has its own data, and hence local join process can proceed.

The local join process itself is divided into two steps: hash and probe. The hashing is carried out to one class, whereas the probing is performed to the other class. In the hashing part, it basically runs through all elements of each collection in a class. The probing part is done in similar way, but is applied to the other class. Figure 5 shows the pseudo-code for parallel hash join algorithm for collection-intersect join queries.

```
Program Parallel-Hash-Collection-Intersect-Join
Begin
  Step 1 (data partitioning):
    Divide and Partial Broadcast version:
      Call DivideAndPartialBroadcast partitioning
    Simple Replication version:
      Call SimpleReplication partitioning

  Step 2 (local joining): In each processor
    a. Hash
       For each object a(c1) of class A
         Hash collection c1 to a hash table
       End For
    b. Probe
       For each object b(c2) of class B
         Hash and probe collection c2 into the hash table
         If there is any match Then
           Concatenate obj b and the matched obj a into query result
         End If
       End For
End Program
```

Fig. 5. Parallel Hash Collection-Intersect Join Algorithm

5 Conclusions and Future Work

The need for join algorithms especially designed for collection-intersect join queries is clear, as collection-intersect join predicates normally require intermediate results to be generated, before the final predicate results can be determined. This is certainly not optimal. In this paper, we present two algorithms especially design for collection-intersect join queries, namely *Parallel Sort-Merge* and *Parallel Hash* algorithms. These two algorithms are designed especially for collection-intersect join queries in object-oriented databases. Data partitioning

methods, which create parallelism, are based on either simple replication or "divide and partial broadcast". Parallel Sort-Merge can make use only the divide and partial broadcast method, whereas Parallel Hash can have a choice between the two partitioning methods. Once a data partitioning method is applied, local join is carried out by either a sort-merge operator (in the case of parallel sort-merge algorithm) or a hash function (in the case of parallel hash). Local join process is therefore much straightforward, as each processor performs sequential sort-merge or hash operations. Hence, the critical element is the data partitioning method.

Our future work includes evaluating the two data partitioning methods (and possibly other partitioning methods) for parallel collection-intersect join algorithms, since data partitioning plays an important role in the overall efficiency of parallel collection join algorithms.

References

1. Cattell, R.G.G. (ed.), The Object Database Standard: ODMG-93, Release 1.1, Morgan Kaufmann, 1994. 505, 506
2. DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", Communication of the ACM, **35**(6), pp. 85-98, 1992.
3. Kim, K-C., "Parallelism in Object-Oriented Query Processing", Proceedings of the Sixth International Conference on Data Engineering, pp. 209-217, 1990. 506
4. Leung, C.H.C. and Ghogomu, H.T., "A High-Performance Parallel Database Architecture", Proceedings of the Seventh ACM International Conference on Supercomputing, pp. 377-386, Tokyo, 1993. 507
5. Leung, C.H.C., and Taniar, D., "Parallel Query Processing in Object-Oriented Database Systems", Australian Computer Science Communications, **17**(2), pp. 119-131, 1995. 506
6. Mishra, P. and Eich, M.H., "Join Processing in Relational Databases", ACM Computing Surveys, **24**(1), pp. 63-113, March 1992.
7. Taniar, D., and Rahayu, W., "Parallelization and Object-Orientation: A Database Processing Point of View", Proceedings of the Twenty-Fourth International Conference on Technology of Object-Oriented Languages and Systems TOOLS ASIA'97, Beijing, China, pp. 301-310, 1997. 506
8. 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 1998.
9. Taniar, D. and Rahayu, J.W., "A Taxonomy for Object-Oriented Queries", a book chapter in Current Trends in Database Technology, Idea Group Publishing, in press (1998). 505
10. Taniar, D. and Rahayu, J.W., "Divide and Partial Broadcast Method for Parallel Collection Join Queries", High-Performance Computing and Networking, P.Sloot et al (eds.), Lecture Notes in Computer Science 1401, Springer-Verlag, pp. 937-939, 1998. 509
11. Thakore, A.K. and Su, S.Y.W., "Performance Analysis of Parallel Object-Oriented Query Processing Algorithms", Distributed and Parallel Databases **2**, pp. 59-100, 1994. 506