PARALLEL DOUBLE SORT-MERGE Algorithm for
Object-Oriented Collection Join Queries

David Taniar
Department of Computer and Math. Sc.
Victoria University of Technology
P.O. Box 14428 MC, Melbourne 8001
Australia
taniar@matilda.vut.edu.au

Wenny Rahayu
School of Comp. Sc. and Comp. Eng.
La Trobe University
Bundoora 3083, Victoria
Australia
wenny@latcs1.lat.oz.au

Abstract

In object-oriented databases (OODB), although path expressions through pointer connections may exist, it is sometimes necessary to perform an explicit join operation between two classes. Since a class may contain collection attributes, as well as simple attributes, join queries in OODB may be based on collections. A need for collection join algorithms arises, since conventional join algorithms, such as hybrid-hash join, GRACE join, were not designed for collection join queries. A new algorithm, so called "Parallel Double Sort-Merge" join algorithm, is proposed. This algorithm plays an important role in parallel object-oriented query processing, due to its superiority over the conventional join methods.

1. Introduction

In relational databases, a join operation connects two or more tables based on a common attribute [7]. In object-oriented databases (OODB), although path expression between classes may exist, it is sometimes necessary to join two or more classes due to an absence of pointer connections or to a 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. Consequently, object-oriented join queries may also be based on collection attributes [9]. If we indicate a node as a class, a typical join operation between two classes is shown as in Figure 1. The query is to retrieve pairs of Journal and Conference having the same co-editor and co-chair. As the co-editor and co-chair attributes are not simple types (in this case set of Person), the join predicate checks for full equality of the two sets, that is all co-editors of a journal must be equal to all co-chairs of a conference.

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. In this paper, we consider a broad range of collection types. The Object Database Standard ODMG [1] defines different kinds of collections: particularly set, list/array, and bag. Unlike in lists/arrays, elements in a set or a bag are not in a particular order. Besides, lists/arrays and bags allow duplicate values to exist. This makes join operations in OODB much more complex than that of relational and nested-relational databases.

![Figure 1. Object-Oriented Join Query](image)

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 [3,6,10]. However, most research done in this area concentrated on path expression queries with pointer chasing. Join processing exploiting collection attributes has not been given much attention. It is the aim of this paper to present a parallel join algorithm especially designed for collection join queries. It is called PARALLEL DOUBLE SORT-MERGE join algorithm. This algorithm is non-trivial to Parallel OODB, since most conventional join algorithms (eg.,
hybrid hash join, sort-merge join) deal with single-valued attributes and hence most of the time they are not capable to handle collection join queries without complicated tricks, such as using loop division (repeated division operator) and intersection.

The rest of this paper is organized as follows. Section 2 describes briefly collection join queries. Section 3 presents the "Parallel Double Sort-Merge" join algorithm. Section 4 explains how collection join queries may be processed using relational division operator. Section 5 presents a quantitative analysis. Section 6 shows simulation results. Finally, section 6 gives the conclusions.

2. Collection Join

Collection Join Queries contain join predicates in a form of standard comparison using relational operators, such as =, !=, <, >, and ≥. Unlike join queries in relational databases, operand of collection join needs not to be of simple attributes. The comparison can also be on collections, as relational operators work with non-primitive values. This is true even in relational databases. For example, a string which is implemented in an array of characters, can be compared with another string. The join predicate may look something like this: (Student.Suburb < Lecturer.Suburb).

One characteristic of collection join is that the join result maybe determined by the first element in the collections. For each pair of objects to compare, a negative answer is obtained if the first elements of the collections are not matched. The opposite is not applied as the comparison for subsequent elements is required.

A typical collection join query is to compare two collections for an equality. Suppose the attribute co-editor of class Journal and the attribute co-chair of class Conference are of type arrays of person. To retrieve conferences chaired by all editor-in-chief of a journal, the join predicate becomes (co-editor = co-chair). Only pairs having an exact match between the join attributes will be retrieved. The query expressed in OQL [1] can be written as follows:

OQL

```
Select x, y
From x in Journal, y in Conference
Where x.co-editor = y.co-chair
```

Using an example shown in Figure 2, the result of collection join is provided in Figure 3.

Relational operators are overloaded functions. This feature is not new to object-oriented join queries, because long before OODB exists, relational operators in relational joins have shown this capability. For example, it is possible to compare an integer with a real number. One of the operand is automatically converted to the type of the other operand (in this case, integer to real). Casting a collection, however, must be done explicitly in the join predicate. Using the previous example, if co-editor is a list and co-chair is a set, the equality predicate becomes (List_to_Set(co-editor) = co-chair), where the co-editor is converted from a list to a set. Comparing two sets/bags can be done easily by sorting them prior to the actual comparison.

![Figure 2. Sample data](image)

![Figure 3. Collection join query results](image)

3. PARALLEL DOUBLE SORT-MERGE Join Algorithm

Parallel Double Sort-Merge join algorithm proceeds in two steps. The first step is the partitioning step which produces disjoint partitions, and the second step is the joining step. Since the partitions are disjoint, the join operation in each partition can be done independently.

3.1. Disjoint Partitioning Step

Data partitioning method employed by the Parallel Double Sort-Merge join algorithm is much influenced by common practises of arrays/sets comparison in programming. An array can be compared with another array by evaluating each pair of elements from the same position of the two arrays. A characteristic of arrays comparison is that once an element is found to be different from its counterpart (i.e., element of the same position from the other array), the comparison stops and returns a negative result.

Unlike arrays comparison, sets comparison is not based on the position of each element in the collection, since the order of the elements is not significant. For example, array(2,3,1) ≠ array(3,2,1), but set(2,3,1) = set (3,2,1). In comparing two sets, it will become easier if the two sets are alphabetically/numerically sorted first. For instance, set(2,3,1) is sorted to be set(1,2,3), and so
is the second set. Comparison can then be carried out as per array comparison.

<table>
<thead>
<tr>
<th>Collections 1</th>
<th>Collections 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,1)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(4,2)</td>
<td>(6,3)</td>
</tr>
<tr>
<td>(2,1)</td>
<td>(6,2)</td>
</tr>
<tr>
<td>(1,5)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>(6,2)</td>
<td>(5,1)</td>
</tr>
<tr>
<td>(5,3)</td>
<td>(3,5)</td>
</tr>
<tr>
<td>(3,4)</td>
<td>(2,4)</td>
</tr>
</tbody>
</table>

**DATA PARTITIONING**

CASE 1:
Collections 1 and 2 are arrays, 3 processors are used, and range partitioning is used (proc 1 = elements 1-2, proc 2 = elements 3-4, proc 3 = elements 5-6).

| (2,1) | (1,2) |
| (1,5) | (1,3) |
| (1,2) | (2,4) |
| (3,1) | (3,2) |
| (4,2) | (3,5) |
| (3,4) | (2,4) |

Processor 1

Processor 2

Processor 3

**CASE 2:**
Collections 1 and 2 are sets, 3 processors are used, and range partitioning is used (proc 1 = element 1, proc 2 = element 2, proc 3 = element 3).

| (3,1) | (1,2) |
| (2,1) | (1,3) |
| (1,5) | (1,3) |
| (1,2) | (5,1) |
| (4,2) | (3,2) |
| (4,2) | (3,5) |
| (5,3) | (6,2) |
| (5,1) | (6,2) |
| (6,2) | (6,2) |
| (6,2) | (6,2) |

Processor 1

Processor 2

Processor 3

**Figure 4. Data Partitioning**

It can be summarized that array comparison much depends on the position of each element in an array. The first element will open the gate for further element comparisons, if the first pair is evaluated to be true. In contrast, set comparison depends on the smallest element in a set, which is the first element after sorting. This element acts like the first element in the array. Based on these characteristics, the first element of an array and the smallest element of a set play an important role in data partitioning.

Common horizontal data partitioning strategies, such as range or hash, can be used. Parallelism is achieved through data partitioning, and each partition is allocated a processor on which to act on. Load balancing can be maintained through bucket tuning [4] which has been widely respected by most parallel database systems. Because the partitioning attribute, also being the join attribute, is a collection, only one of the elements will be used as the partitioning value. If the collection is an array or a list, partitioning is based solely on the first element of the list/array. As stated above, the reason is that a list/array comparison operates on the original elements composition of the collection. If the partitioning attribute is a set or a bag, partitioning is based on the smallest element of the collection, because a set/bag comparison will require the collections to be sorted. Consider the two cases in Figure 4 as an example. Case 1 is where the two collections are arrays, whereas case 2 is where the collections are sets.

3.2. Joining Step

As the name states, a PARALLEL DOUBLE SORT-MERGE join is a variant of a parallel sort-merge join, and is recognized to be better than the conventional nested loop join through a divide and broadcast technique. Parallel Double Sort-Merge join algorithm makes use of the sort-merge operation twice; one to the collection attribute, the other to the objects.

**CASE 1 (Arrays):**

| (1,2) | (1,2) |
| (1,5) | (1,3) |
| (2,1) | (2,4) |

Processor 1

Results=(1,2,1,2)

Processor 2

Results=none

Processor 3

Results=(6,2,6,2)

**Figure 5. Sorting phase**

The joining step is further decomposed into the sorting and the merging phases. The sorting operation is applied twice: to the collections, and to the objects. Sorting each collection is only needed if the collection is a set or a bag, and sorting the objects is based on the first element (if it is an array or a list) or on the smallest element (if it is a set or a bag). The sorting phase is not carried out before data partitioning, as sorting done in parallel in each processor after data partitioning will minimize the time. Figure 5 shows the result of the sorting phase of the two cases presented previously.
Like the sorting phase, the merging phase consists of two operations: object merging and collection merging. Merging the objects of the two classes is based on the first element of each collection. If they are matched, subsequent elements comparison can proceed. Merging the two collections of each pair of objects is, in fact, implemented by the relational operators. Figure 6 gives the pseudo-code for the Parallel Double Sort-Merge join algorithm.

**Procedure ParallelDoubleSortMerge:**

// step 1: partitioning step
partition the objects of both classes based on their first elements (for lists/arrays), or their minimum elements (for sets/bags).

// step 2: joining step
// sort phase
(i) sort the elements of each collection (for sets/bags only).
(ii) sort the objects based on the first element of the collection.

// merge phase
(iii) merge the objects of both classes based on their first element on the join attribute.
(iv) if matched, merge the two collection attributes based on their individual elements (starting from the second element).

End.

**Figure 6. Parallel Double Sort-Merge**

4. Relational Division

In this section, we will present how collection join queries may be processed using conventional relational algebra operators. To process collection join queries, conventional partitioned join algorithm (e.g., hybrid hash join) will have each class or table normalized prior to joining. Partitioning is then carried out based on the join attribute. For each partition, a hash join is performed. This simple join method will not produce correct results, unless a division operator is applied, because the joining operation must be on collection, not on individual elements. Therefore, collection join queries must be processed using relational algebra operators, particularly a division and an intersection operator. If the first class of the join operand is regarded as a divisor table, and each collection of the second class is regarded as a dividend table, the division between these tables will result in pairs of objects satisfying the join predicate. Figure 7 shows an example of a relational division. The divisor table is a union of all co-editors, and the dividend table is a co-chair collection of the first conference object y1. The result of this division is the combination of x2 and y1.

It is clear from the example that the division operation must be repeated for each collection of objects from the second class (it is called a loop division). The algorithm can be written as follows.

// subquery 1
for each collection c in objects of the second class
all collections of the first class divided by c giving Temp
T1 = T1 + Temp

**Figure 7. Relational Division**

Figure 8 shows the process of the first loop division (subquery 1). The results T1 are x2-y1, x1-y2, x1-y3, and x2-y3.

The division operator is a manifestation of a universal quantifier, which differs from the collection equality. The universal quantifier evaluates whether a divisor object contains all values of the dividend table. This requirement does not ensure that all values within a divisor object must contain all values in the dividend table. Therefore, the another loop division must be carried out to the two classes, but with a reverse role (e.g., the division is the second table and the dividend is each collection of the first table). The following pseudo-code is for the second loop division operation.

// subquery 2
for each collection c in objects of the first class
all collections of the second class divided by c giving Temp
T2 = T2 + Temp

**Figure 8. Loop Division**

Figure 9 shows another loop division where the divisor is now class conference and the dividend is co-editor collection.

The results from the first (T1) and the second (T2) loop division are intersected to get the final result.

Collection-Join = T1 intersect T2.
The intersection of \( T_1 \) and \( T_2 \) is given by \( x_2-y_1 \) (Figure 10).

| \( y_1 \) | \( p_2 \) |
| \( y_1 \) | \( p_3 \) |
| \( y_2 \) | \( p_1 \) |
| \( y_3 \) | \( p_2 \) |

\[ \text{divided by} \]

| \( p_1 \) | \( x_1 \) |
| \( p_2 \) | \( x_1 \) |
| \( p_2 \) | \( x_2 \) |
| \( p_3 \) | \( x_2 \) |
| \( p_3 \) | \( x_3 \) |

\[ \text{giving} \]

\[ \text{Figure 9. Reversed Loop Division} \]

5. Analytical Analysis

In this section we evaluate and compare performance of the proposed algorithm with the traditional relational division. The partitioning method used by the two methods is assumed to be the same, i.e., disjoint partitioning based on the first/smallest element in each collection. Hence, we concentrate on the local join cost only. Since the local processing put emphasis on data processing in processors, our cost models focus on the computation time only. This is certainly valid since the purpose of the analysis is to perform a relative comparison, rather than to estimate the total execution costs. The cost model is expressed in term of number of comparisons involved in the algorithm, which is very much influenced by the number of objects and the size of collections. The variables used in the cost models are summarized in Table 1.

<table>
<thead>
<tr>
<th>Var.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>Number of objects in the first class</td>
</tr>
<tr>
<td>( s )</td>
<td>Number of objects in the second class</td>
</tr>
<tr>
<td>( e_1 )</td>
<td>Average collection size in each object of the first class</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>Average collection size in each object of the second class</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>Selectivity factor (in fraction)</td>
</tr>
</tbody>
</table>

\[ \text{Table 1. Variables used in the cost models} \]

The merging operation is known to be linear, i.e. \( O(N) \) [5]. Object merging involves all objects from the two classes. When they are matched, another merging process is needed to merge the collection. The second merging process depends on the join selectivity factor. The merging cost is estimated as follows:

\[ (r + s) + \sigma(r,e_1 + s,e_2) \]

The first term is the object merging cost, and the second term is the collection merging cost. The total cost for the Parallel Double Sort-Merge join algorithm can be written as follows:

\[ \text{ParDoubleSortMerge} = r.e_1 \log e_1 + s.e_2 \log e_2 + (r + s) + \sigma(r.e_1 + s.e_2) \]

(1)

If it is assumed that \( n \) processors are used, and the workload is divided quite equally, the parallelism cost is obtained by dividing equation (1) by \( n \). Since the relational division uses the same partitioning strategy to achieve the same parallelism degree, the parallelism factor can be eliminated, as we are only concentrating on a relative comparison.

The division operator is often explained as a double join operation [8]. The first subquery containing \( r \) divided by \( e_2 \) giving \( T_1 \) can be extracted to \( r \) join \( e_2 \) giving \( T_11 \), and then \( r \) join \( T_11 \) giving \( T_1 \). As this is repeated as many times as the number of objects in the first class, the cost model for the first subquery is:

\[ \text{Subquery}_1 = [(r.e_1 \log r.e_1 + e.2 \log e_2 + r.e_1 + e_2) + (r.e_1 + T_11)].s \]

(2)

where the first two terms are the costs for sorting all objects of the first and the second classes, and the next two terms are for merging. These terms are part of the first join between \( r \) and \( e_2 \). The last two terms are the merging cost between \( r \) and \( T_11 \). No sorting is necessary for the second join, as both join operand have been sorted by the first join. The two joins are repeated as many times as the number of objects in the second class.

Without obtaining the full cost for the conventional approach, it can be shown that the Parallel Double Sort-Merge join outperforms the double loop division. By comparing equations (1) and (2), the proposed algorithm is demonstrated to be better than the conventional approach by at least one subquery and an intersection cost of the conventional join.

The division operator by nature is already expensive. It will be even more expensive as the division operator is executed repeatedly. Furthermore, it is necessary to consolidate the temporary results by means of an intersection. On the other hand, the Parallel Double Sort-Merge join is not only simple but also efficient.
6. Simulation Results

In order to validate the analytical performance comparison, simulations have been carried out. In the simulations, data are generated by running a random number generator. A transputer-based simulator Transim [2] was used, and the program was written in an Occam-like language supported by Transim. In the experimentations, a few variables was varied, particularly the sizes of the join operand and the join selectivity factor. The results are shown in Figure 11 and Figure 12.

![Figure 11. Varying the Class Sizes](image)

![Figure 12. Varying the Selectivity Factor](image)

Figure 11 shows that performance of the proposed algorithm is always better than the conventional algorithm using a loop division (by around 20%), even though the difference is not as drastic as predicted by the analytical analysis. This is due to some repetitive tasks of the loop division, such as repetitive sorting of the two classes, have been avoided. The costs for both algorithm increase as the size of the operand grows. For the proposed method, the increase is influenced by the cost for sorting the collection in each object. This is repeated as many objects in a class, and hence, the cost increases. For the relational division, the cost increases sharply especially for large operand. This is due to the expensive loop division cost.

Figure 12 shows that the join selectivity factor does not affect the degradation of performance significantly. For the proposed method, it appears that the merging cost for the matched collections is only a small component of the overall cost. And for the relational division method, intersection cost component seems to be small, compared to the loop division. Hence, the join selectivity factor does not play a significant role in the overall performance.

7. Conclusions and Future Work

The need for join algorithms especially designed for collection join queries is clear, as the conventional parallel join algorithms were not designed for collection types. In this paper, we have presented a join algorithm called "Parallel Double Sort-Merge". Performance of this algorithm is promising compared to the traditional and complicated relational division operator, since the division operator must be applied repetitively (i.e., loop division) and an intersection must also be used to obtain the final results. Our future plan includes investigating the possibility to use a hash method, instead of sort-merge method, in the joining step of the algorithm.

References