

Efficient Join Algorithms for Integrating XML Data in Grid Environment

Hongzhi Wang, Jianzhong Li, and Shuguang Xiong

Harbin Institute of Technology, Harbin, China
wangzh@hit.edu.cn, lijz@mail.banner.com.cn, n2xiong@126.com

Abstract. For its self-description feature, XML can be used to represent information in grid environment. Querying XML data distributed in grid environment brings new challenges. In this paper, we focus on join algorithms in result merge step of query processing. In order to transmit results efficiently, we present strategies of data compacting, as well as 4 join operator models. Based on the compacted data structure, we design efficient algorithms of these join operators. Extensive experimental results shows that our data compact strategy is effective; our join algorithms outperform XJoin significantly and have good scalability.

1 Introduction

Grid is a promising computing environment. One of the important application running on grid is information processing. Information processing in grid can not only gain more computation power to process very large data but also integrate information from various data sources.

XML is an important format of information integration. One of its features making XML popular is that XML document has tags representing semantic of data. In grid environment [4], with heterogeneity among nodes, representing data by XML can make information processing in grid environment flexible.

With effective query processing techniques, queries to grid can retrieve data from all data sources in data grid like from single data source. One of the important step of query processing of grid is result merge, which is to merge intermediate results returned from data sources. Among the operators in result merge, join operation is a difficult one.

Challenges brought by Join operation in grid environment include:

- The bandwidth has limit. Strategy should be presented to save the bandwidth without affecting the query evaluation efficiency.
- The data from data sources may arrive in instable style. Even though existing algorithm for data in relation form can solve part of the problem, the join operation of XML data may be much more complex. The problem will be more interesting when the join operation is performed on the XML data compacted for transmission.

Based on the feature of semi-structured data, we focus on efficient evaluation of join operations for XML data in the merge step of query processing in grid environment. The contributions of this paper includes:

- To save bandwidth, we design data transmission strategy for intermediate results of XQuery. This strategy supports efficient join evaluation in result merge step.
- Based on this transmission format, we present models for join operations in result merge step.
- For each join model, we give an efficient algorithm for implementation of the join operation.
- Our extensive experimental results demonstrate that our transmission strategy can save lots of bandwidth and accelerate the transmission. It also significantly outperforms XJoin in the evaluation of join over intermediate results.

The rest of the paper is organized as follows: Section 2 presents the strategy of compacting data for transmission of intermediate results. Section 3 discusses join models and their implementations. We present our experimental results and analyses in Section 4. Section 5 concludes the paper.

2 Prepare Data for Transmission

In this section, we discuss the strategy of compacting data for transmission. We have two strategies.

Tree Structure Keeping. This strategy is to keep the parent-child or ancestor-descent relationship of XML nodes. The benefit of this strategy is that it will avoid information redundancy.

Large Element Latency. When the query selectivity is small, lots of objects will not exist in the result of join. If the candidate of join is large element with complex structure but only a small part of them will be in the final result, since these objects should also be transmitted, the bandwidth is wasted. To address this problem, our strategy is that only the id of large element is transmitted with the elements to perform join operation on. In the final step of generation of final result, selected large elements are obtained from data source with their ids. We will also illustrate the benefit with an example.

3 Join Algorithms

In this section, we will present algorithms to process join of XML in grid environment.

3.1 Models of Join Operations

Based on different types of queries, we can define four join models. Both model can be represented as “ $L \square R$ ”, where \square is a join operator. Tuples corresponding to L are called *left candidates* and tuples corresponding to r are called *right candidates*.

Natural Join. For two sets of XML fragment, nature join is to connect the fragment together with the join attributes existing once in join results. Considering XML feature, the result of the natural join is corresponding attributes with the intersection of join sets.

Combined Join. For two sets of XML fragment, combined join is the same as natural join but the result has no the intersection of join sets. Note two tuples can be combined joined only when the intersection of join sets is not empty.

Semi-join. For two sets of XML fragment, semiJoin is to filter left set with the join set of right candidates. The tuple is filtered when the intersection of join sets is not empty. The result contains the intersection of join sets.

Half Join. For two sets of XML fragment, half join is to filter left set with the join set of right candidates. The result does not contain the intersection of join sets. The tuple is filtered when the intersection of join sets is not empty.

3.2 Implementations of Join Operators

Join Algorithm for Half Join Model For implementation operation half join, basic data structure is a hash set to store all elements of join sets of right candidates. When a tuple t as left candidate comes, find each item in its corresponding set, if any item is found in the hash set, corresponding value of t is output and the processing of tuple t is finished. The algorithm is shown in Algorithm 1, where H is a hash set containing all elements of join sets in right candidate of join. Symbol *end* identifies the end of the candidates.

Algorithm 1. HalfJoin()

```

1: while left candidate is not end do
2:   if A tuple  $t$  of left candidate comes then
3:     for each element  $e$  in  $t.S$  do
4:       if  $e$  can be found in  $H$  then
5:         output  $t - t.S$ 
6:   if A tuple  $t$  of right candidate comes then
7:     for each element  $e$  in  $t.S$  do
8:       insert  $e$  into  $t.S$ 

```

Join Algorithm for SemiJoin Model. The implementation of SemiJoin is similar as that of half join. But For each tuple t of A, each value in $t.S$ that can be find in the hash set is outputted with $t.left$. The algorithm is shown in Algorithm 2.

Join Algorithm for Combined Join Model. The basic idea of the implementation of combined join model is to attach a signature to each tuple to represent the context of the join set, so that whether two join sets are intersected can be joined by the two signature. Since the value range of a node can not be determined, accurate signature is too long. We choose bloom filter [1] as the signature. To each tuple as left or right candidate, a bloom filter is assigned. Tuples as left or right candidates with the same bloom filter are compacted together, respectively. Two global bloom filters are maintained as the union of all bloom filters

Algorithm 2. semiJoin()

```

1: while left candidate is not end do
2:   if A tuple  $t$  of left candidate comes then
3:     for each element  $e$  in  $t.S$  do
4:       if  $e$  can not be found in  $H$  then
5:         delete  $e$  from  $t.S$ 
6:       if  $t.S$  is not empty then
7:         output  $t$ 
8:   if A tuple  $t$  of right candidate comes then
9:     for each element  $e$  in  $t.S$  do
10:      insert  $e$  into  $t.S$ 

```

of left and right candidates, respectively. Since bloom has positive false, when a new tuple t as left candidate comes, its bloom filter b_t is computed. It is intersected with the global bloom filter of right candidates. If the result is not 0, it is intersected with each bloom filter f_r of right candidates, and it will be judged whether the intersection of $t.S$ and the candidate set of each tuple t_r as right candidate with bloom filter f_r is empty. If the answer is no, then the tuple with $t - t.S$ and $t_r - t_r.S$ is outputted. In order to accelerate the judgement, we insert all elements into a hash set H and check whether $t_r.S$ has any element in H . We perform similar steps on a tuple as right candidate but it will be compared with tuples as left candidates. The algorithm is shown in Algorithm 3, where S_l and S_r are sets of bloom filters of left candidates and right candidates, respectively; $filter_l$ and $filter_r$ are global filters of left candidates and right candidates, respectively. In this algorithm, we only give the process steps when a tuple of left candidate comes. Because of symmetry, a tuple of right candidate is processed in similar method.

Algorithm 3. Combined Join()

```

1: while left candidate is not end and right candidate is not end do
2:   if A tuple  $t$  of left candidate comes then
3:      $b_t = \text{bloom\_filter}(t)$ 
4:      $b_t.\text{tuple} = t$ 
5:     add  $b_t$  to  $S_l$ 
6:      $filter_l = \text{Union}(filter_l, b_t)$ 
7:     insert all elements of  $t.S$  into  $H$ 
8:     if  $\text{intersect}(filter_r, b_t) \neq 0$  then
9:       for each element  $f_r$  in  $S_r$  do
10:        if  $\text{intersection}(f_r, b_t) \neq 0$  then
11:          for each element  $e$  in  $f_r.\text{tuple}.S$  do
12:            if  $e$  is in  $H$  then
13:              output  $(t - T.s, f_r.\text{tuple} - f_r.\text{tuple}.S)$ 

```

Join Algorithm for Natural Join Model. The implementation of natural Join model can be considered as the combination of Combined Join algorithm and Xjoin[3] algorithm. That is, when a tuple t as left candidate comes, at first its bloom filter is intersected with global filter of right candidates. If the result is

not 0, the tuple is joined with right candidates by XJoin algorithm. In order to perform XJoin, the join set should be split. Each tuple is converted to a set of tuples, each of which has one element in join set with all other attributes. The element from join set is *join attribute*. Two hash tables are associated to left and right candidates, respectively. Each converted tuple is inserted into corresponding bucket in the hash table based on the hash value of join attribute. Each converted tuple is also joined with the tuples in corresponding bucket of opposite hash table.

4 Experimental Results

In this section, we present results and analyses of part of our extensive experiments of data transmission strategy and join algorithms.

4.1 Experimental Setup

All of our experiments were performed on a PC with Pentium 1GMHz, 512M memory and 40G IDE hard disk. The OS is Windows 2000 Professional. We implemented our system using Microsoft Visual C++ 6.0. We implemented our data transmission strategy and half join, semiJoin, combined join and natural join algorithms. For comparison, we also implementation XJoin algorithm.

We choose XMark as test data[2]. XMark document. In the result we only give count the run time of join algorithms.

We designed a set of queries that has different characteristics. We consider comparison feature of left and right candidates: one-to-one(oo for brief) and one-to-multiple(om for brief). Queries are shown in Table 1.

Table 1. Query Set and Comparison of Data Transmission

Query	left candidate	right candidate	OEM	SEM	OQM	SQM
Q1	person/id	bidder//person	84758	84758	11047764	1972700
Q2	person//category	item//category	119804	76269	36380718	1815120
Q3	open_auction//person	person/id	108758	73129	22943812	2012104
Q4	open_auction/id	person//open_auction	62828	43019	22943812	2012104

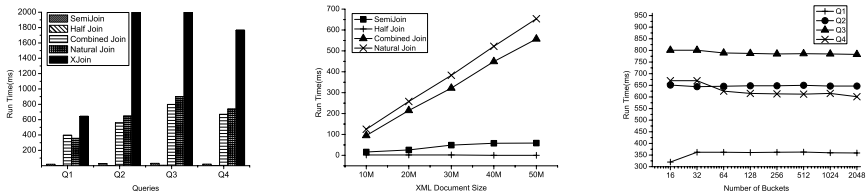
4.2 Data Transmission Strategy

In order to evaluate our data transmission strategy, we compare the size of original data to transmit with the size of data applied our strategy to transmit. Results of data transmission strategy is shown in Table . To test “Tree Structure Keeping” strategy, we compare the number of elements to be transmitted in original XML document(OEM for brief) and in our strategy (SEM for brief). To test “Large Element Latency” strategy, we compare the quantity of data (in byte) to be transmitted for SemiJoin algorithm of all these 4 queries. The quantity of original data should be transmitted is called OQM for brief and the quantity of data compacted by our strategy is called SQM for brief. The

results are shown in Table 1. From the results, in most instances, our strategy can compact data to transmit significantly. One special is Q1, OEM equals to SEM. It is because each person element only has one id and each bidder element only has one person attribute.

4.3 Results

Comparisons. In this subsection, we present comparison results with XJoin, which is a efficient join algorithm for relational model in information integration system. Since XJoin performs the 4 kinds of join operation in the same method, we compare all our join algorithms with XJoin. We fixed the number of buckets of hash 32 and the filter length 32 bits. The results of comparison are shown in Figure 1(a). From the result, our algorithms performs to XJoin significantly.



(a) Comparison with XJoin (b) Run time vs document size (c) Time VS Bucket Number

Fig. 1. Experimental Results

Scalability. We vary the size of document from 10M to 50M. The results are shown in Figure 1(b). From the result, the run time nearly linearly increases with document size. Our algorithms have good scalability.

Changing the Number of Buckets. In order to test the affect of number of buckets to the efficiency of Natural Join algorithm, we fix the length of filter 32 bits and vary the number of buckets from 16 to 2048. Results are shown in Figure 1(c). From the result, the number of buckets does not affect the efficiently remarkably. It is because that the number the selectivity of the join operation is not very large and the bloom filter filtered most of useless results.

5 Conclusions

In this paper, we focus on efficient join algorithms for information integration in grid environment based on XML. We address this problem in two aspects. One is to save bandwidth by compacting data for transmission. The other is to design efficient algorithm based on compacted structure adaptive to transmission. Our extensive experiments shows that data compacting strategies are effective and our join algorithms are efficient with good scalability. Further work includes estimating of intermediate results in grid environment and designing query optimization strategies for join operations in grid environment.

References

1. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.
2. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 974–985, 2002.
3. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
4. H. Zhuge. China's e-science knowledge grid environment. *IEEE Intelligent Systems*, 19(1):13–17, 2004.