Parallelizing Structural Joins to Process Queries over Big XML Data Using MapReduce

Huayu Wu

Institute for Infocomm Research, A*STAR, Singapore
huwu@i2r.a-star.edu.sg

Abstract. Processing XML queries over big XML data using MapReduce has been studied in recent years. However, the existing works focus on partitioning XML documents and distributing XML fragments into different compute nodes. This attempt may introduce high overhead in XML fragment transferring from one node to another during MapReduce execution. Motivated by the structural join based XML query processing approach, which uses only related inverted lists to process queries in order to reduce I/O cost, we propose a novel technique to use MapReduce to distribute labels in inverted lists in a computing cluster, so that structural joins can be parallelly performed to process queries. We also propose an optimization technique to reduce the computing space in our framework, to improve the performance of query processing. Last, we conduct experiment to validate our algorithms.

1 Introduction

The increasing amount of data generated by different applications and the increasing attention to the value of the data marks the beginning of the era of big data. It is no doubt that to effectively manage big data is the first step for any further analysis and utilization of big data. How to manage such data poses a new challenge to the database community.

Gradually, many research attempts converge to a distributed data processing framework, MapReduce [9]. This programming model simplifies parallel data processing by offering two interfaces: map and reduce. With a system-level support on computational resource management, a user only needs to implement the two functions to process underlying data, without caring about the extendability and reliability of the system. There are extensive works to implement database operators [5][12], and database systems [3][11] on top of MapReduce.

Recently, researchers started looking into the possibility of managing big XML data in a more elastic distributed environment, such as Hadoop [1], using MapReduce. Inspired by the XML-enabled relational database system, big XML data can be stored and processed by relational storage and operators. However, shredding XML data in big size into relational tables is extremely expensive. Furthermore, with relational storage, each XML query must be processed by several θ-joins among tables. The cost for joins is still the bottleneck for Hadoop-based database systems.

Most recent research attempts [8][7][4] leverage on the idea of XML partitioning and query decomposition adopted from distributed XML databases [13][10]. Similar to the join operation in relational database, an XML query may require linking two or more arbitrary elements across the whole XML document. Thus to process XML queries in a distributed system, transferring fragmented data from one node to another is unavoidable. In a static environment like a distributed XML database system, proper indexing techniques can help to optimally distribute data fragments and the workload. However, for an elastic distributed environment such as Hadoop, each copy of XML fragment will probably be transferred to undeterminable different nodes for processing. In other words, it is difficult to optimize data distribution in a MapReduce framework, thus the existing approach may suffer from high I/O and network transmission cost.

Actually different approaches for centralized XML query processing have been study for over a decade. One highlight is the popularity of the structural join based approach (e.g., [6]). Compared to other native approaches, such as navigational approach and subsequence matching approach, one main advantage of the structural join approach is the saving on I/O cost. In particular, in the structural join approach, only a few inverted lists corresponding to the query nodes are read from the disk, rather than going through all the nodes in the document. It will be beneficial if we can adapt such an approach in the MapReduce framework, so that the disk I/O and network cost can be reduced.

In this paper, we study the parallelization of the structural join based XML query processing algorithms using MapReduce. Instead of distributing a whole big XML document to a computer cluster, we distribute inverted lists for each type of document node to be queried. Since the size of the inverted lists that are used to process a query is much smaller than the size of the whole XML document, our approach potentially reduces the cost on cross-node data transfer.

2 Related Work

Recently, there are several works proposed to implement native XML query processing algorithms using MapReduce. In [8], the authors proposed a distributed algorithm for Boolean XPath query evaluation using MapReduce. By collecting the Boolean evaluation result from a computer cluster, they proposed a centralized algorithm to finally process a general XPath query. Actually, they did not use the distributed computing environment to generate the final result. It is still unclear whether the centralized step would be the bottleneck of the algorithm when the data is huge. In [7], a Hadoop-based system was designed to process XML queries using the structural join approach. There are two phases of MapReduce jobs in the system. In the first phase, an XML document was shred into blocks and scanned against input queries. Then the path solutions are sent to the second MapReduce job to merge to final answers. The first problem of this approach is the loss of the "holistic" way for generating path solutions. The second problem is that the simple path filtering in the first MapReduce phase is not suitable for processing "//"-axis queries over complex structured XML

data with recursive nodes. A recent demo [4] built an XML query/update system on top of MapReduce framework. The technical details were not thoroughly presented. From the system architecture, it shreds an XML document and asks each mapper to process queries against each XML fragment.

In fact, most existing works are based on XML document shredding. Distributing large XML fragments will lead high I/O and network transmission cost. On contrast, our approach distributes inverted lists rather than a raw XML document, so that the size of the fragmented data for I/O and network transmission can be greatly reduced.

3 Framework

3.1 Framework Overview

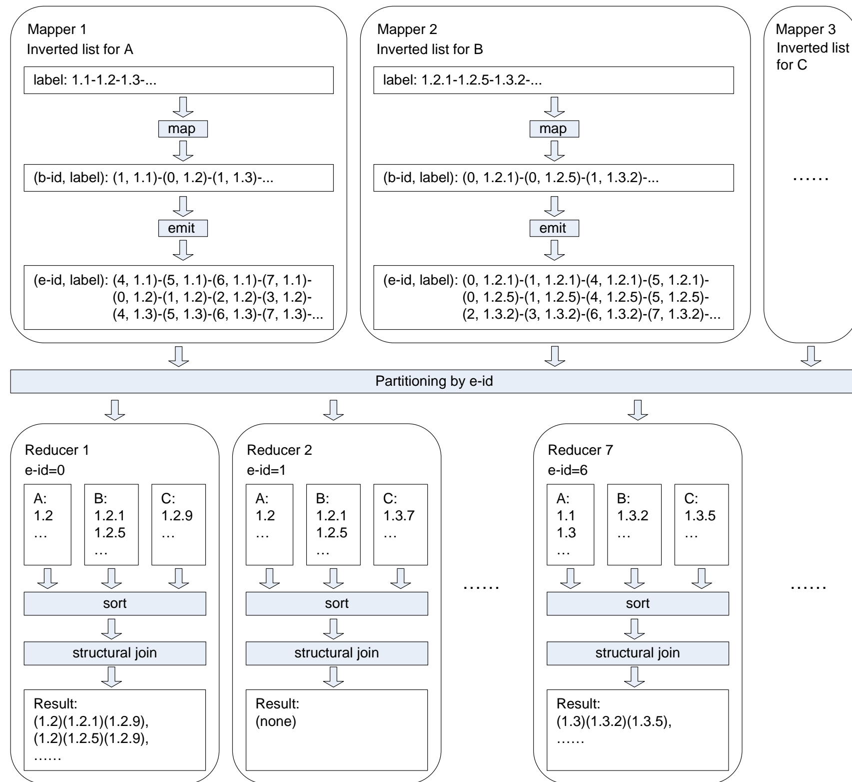


Fig. 1. Data flow of our proposed framework

In our approach, we implement the two functions, map and reduce in a MapReduce framework, and leverage on underlying system, e.g., Hadoop for

program execution. The basic idea is to equally (or nearly equally) divide the whole computing space for a set of structural joins into a number of sub-spaces, and each sub-space will be handled by one reducer to perform structural joins.

Each mapper will take a set of labels in an inverted list as input, and emit each label with the ID of the associated sub-space (called e-id, standing for emit id). The reducers will take the grouped labels for each inverted list, re-sort it, and apply holistic structural join algorithms to find answers. The whole process is shown in Fig. 1, and the details will be explained in the following sections.

3.2 Design of Mapper

The main task of a mapper is to assign a key to each incoming label, so that the overall labels from each inverted list are nearly equally distributed in a given number of sub-spaces for the reducers to process. To achieve this goal, we adopt a polynomial-based emit id assignment.

For a query using n inverted lists and each inverted list is divided into m sub-lists, the total number of sub-spaces is m^n. We construct a polynomial function f of m with the highest degree of $n-1$, to determine the emit id of each sub-space.

$$f(m) = a_{n-1}m^{n-1} + a_{n-2}m^{n-2} + ... + a_1 m + a_0$$
$$where \;\; a_i \in [0, m-1] \;\; for \;\; i \in [0, n-1] \tag{1}$$

Fig. 2 shows an example where $m = 3$ and $n = 3$. The procedure that a mapper emits an input label is shown in Algorithm 1.

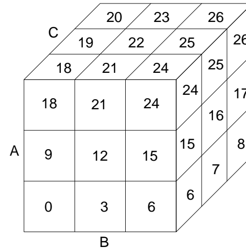


Fig. 2. Example of computing space division

The correctness and the complexity of the proposed polynomial-based emitting can be found in our technical report [14].

3.3 Design of Reducer

After each reducer collecting all labels from all inverted lists that have the same e-id (key), it can start processing the XML queries over this sub-space. Since the map function only splits the computing space into small sub-spaces without other

Algorithm 1 Map Function

Input: an empty key, a label l from an inverted list I as the value; variables m for the number of partitions in each inverted list, n for the number of total inverted lists, r a random integer between 0 and $n-1$

1: identify the coefficient corresponding to I, i.e., a_i where i is the index
2: initiate an empty list L
3: toEmit(L, i) /*m, n, r, l are globally viewed by Function 1 and 2*/

Function 1 toEmit(List L, index i)

```
1: if L.length == number of inverted lists then
2:    Emit(l)
3: else
4:    if L.length == i then
5:       toEmit(L.append(r), i)
6:    else
7:       for all j ∈ [0, n − 1] do
8:          toEmit(L.append(j), i)
```

Function 2 Emit(List L)

1: initiate the polynomial function *f(m)* as defined in (1)
2: set the coefficients of f as the integers in L, by order
3: calculate the value of f given the value of m, as the emit key e-id
4: emit(e-id, l)

operations on data, in the reduce function, any structural join based algorithm can be implemented to process queries.

In the example in Fig. 1, for each reducer, after getting a subset of labels in each inverted list, the reducer will sort each list and then perform holistic structural join on them, to find answers.

4 Optimization

Let us start with a motivating example. Suppose an algorithm tries to process a query //A[//B]//C. In the sorted inverted list for A, the first label is 1.3, while in the sorted inverted list for B the first label is 1.1.2. Obviously, performing a structural join A//B between 1.3 and 1.1.2 will not return an answer.

Recall that in Algorithm 1 when a map function emits a label, it randomizes a local partition and considers all possible partitions for other inverted lists for the label to emit. In our optimization, to emit a label from an inverted list I, we (1) set the local partition in I for the label according to an index pre-defined based on node statistics, i.e., the cut-off index, and (2) selectively choose the coefficients (i.e., the partitions) for all the ancestor inverted lists of I, such that the current label can join with the labels in those partitions and return answers. The *toEmit* function for an optimized mapper is presented in Function 3.

Function 3 toEmit^O(List L, index i)

Input: the partition cut-off index cutoff[x][y] for inverted list I_x and partition y; the current inverted list I_u with numerical id u; other variables are inherited from Algorithm 1

```
1:  if L.length == number of inverted lists then
2:    Emit(l)
3:  else
4:    if L.length == i then
5:      initiate v = 0
6:      while cutoff[x][y].precede(l) && v<m do
7:        v++
8:      toEmit(L.append(v), i)
9:    else
10:     if the query node for I_{L.length} is an ancestor of the query node for I_u then
11:       initiate v = 0
12:       while cutoff[x][y].precede(l) && v<m do
13:         v++
14:       for all k ∈ [0, v − 1] do
15:         toEmit(L.append(k), i)
16:     else
17:       for all j ∈ [0, n − 1] do
18:         toEmit(L.append(j), i)
```

The intuition of the optimization is to prune a label emitting to certain reducers in which it will not produce structural join result. The details of defining the cut-off index and some running examples can be found in [14].

5 Experiment

5.1 Settings

We use a small Hadoop cluster with 5 slave nodes to run experiments. Each slave node has a dual core 2.93GHz CPU and a 12G shared memory. We keep all default parameters of Hadoop in execution.

We generated a synthetic XML dataset with the size of 10GB, based on the XMark [2] schema. The document is labeled with the containment labeling scheme so that the size of each label is fixed. We randomly compose 10 twig pattern queries with the number of query nodes varying from 2 to 5, for evaluation. The result presented in this section is based on the average running statistics.

5.2 Result

Since the main query processing algorithm is executed in the Reduce function, we first vary the number of reducers to check the impact on the performance for both the original MapReduce algorithm and the optimized MapReduce algorithm. We set the number of mappers to be 10, and try different numbers of reducers.

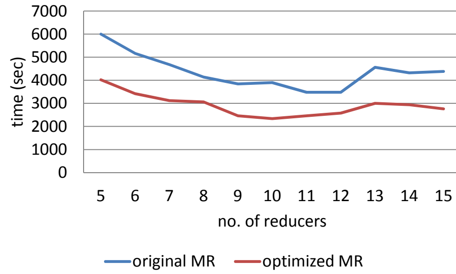
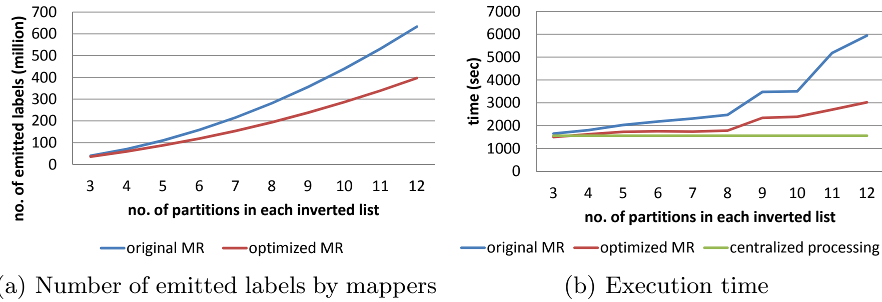


Fig. 3. Performance under different number of reducers



(a) Number of emitted labels by mappers

(b) Execution time

Fig. 4. Performance comparison between the original and the optimized algorithms

From the result in Fig. 3 we can see that the performance for both the original algorithm and the optimized algorithm are improved as the number of reducers increases, until it reaches 10 or 11. After that the performance is not stable. The result accords with the hardware settings. There are 5 dual core processors, which can support 10 tasks to run parallelly. When the processors are fully utilized, the performance may be significantly affected by other issues, such as network transmission overhead.

Also, the optimized algorithm is always better than the original algorithm without optimization. We further show this point in the second experiment.

We keep the number of mappers and the number of reducers to be 10. We vary the number of partitions in each inverted list, which determines the number of reduce jobs. Fig. 4(a) shows the number of labels emitted by the mappers for the two algorithms under different partition numbers for inverted lists. It clearly tells that the optimized algorithm prunes more labels as the sub-spaces are more fine-grained. As a consequence, the performance of the optimized algorithm is better than the original algorithm, as shown in Fig. 4(b).

We can also see from Fig. 4(b) that as the number of partitions in each inverted list increases, the overall performance will drop. We also managed to run the structural join algorithm in a computer with a 3.2GHz CPU and a 8GB memory, and show the average execution time for queries in Fig. 4(b) as well. This comparison shows that the overhead on network data transmission for a

Hadoop cluster is quite large, and can make the performance worse than a single machine (suppose a single machine is capable to handle the program). Thus, we should limit the number of reduce jobs so that each reducer will take over a computing space as large as possible, to fully utilize its resource.

6 Conclusion

In this paper, we proposed a novel algorithm based on the MapReduce framework to process XML queries over big XML data. Different from the exiting approaches that shred and distribute XML document into different nodes in a computer cluster, our approach performs data distribution and processing on the inverted list level. We further propose a pruning-based optimization algorithm to improve the performance of our approach. We conduct experiments to show that our algorithm and optimization are effective.

References

1. http://hadoop.apache.org
2. http://www.xml-benchmark.org
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In: VLDB, pp. 922-933 (2009)
4. Bidoit, N., Colazzo, D., Malla, N., Ulliana, F., Nole, M., Sartiani, C.: Processing XML queries and updates on map/reduce clusters. In: EDBT, pp. 745-748 (2013)
5. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in MapReduce. In: SIGMOD, pp. 975-986 (2010)
6. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD, pp. 310-321 (2002)
7. Choi, H., Lee, K., Kim, S., Lee, Y., Moon, B.: HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In: CIKM, pp. 2737-2739 (2012)
8. Cong, G., Fan, W., Kementsietsidis, A., Li, J., Liu, X.: Partial evaluation for distributed XPath query processing and beyond. ACM Trans. Database Syst. 37(4), 32 (2012)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: USENIX Symp. on Operating System Design and Implementation, pp. 137-150 (2004)
10. Kling, P., Ozsu, M.T., Daudjee, K.: Generating efficient excution plans for vertically partitioned XML databases. PVLDB 4(1), 1-11 (2010)
11. Lin, Y., Agrawa., D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar starage for scalable join processing in the MapReduce framework. In: SIGMOD, pp. 961-972 (2011)
12. Okcan, A. Riedewald, M.: Pricessing theta-joins using MapReduce. In: SIGMOD, pp. 949-960 (2011)
13. Suciu, D.: Distributed query evaluation on semistricutred data. ACM Trans. Database Syst. 27(1), 1-62 (2002)
14. Wu, H.: Parallelizing Structural Joins to Process Queries over Big XML Data Using MapReduce. Tech Report: http://www1.i2r.a-star.edu.sg/∼huwu/paraSJ.pdf