2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

A Grid-based k-Nearest Neighbor Join for Large Scale Datasets on MapReduce

Miyoung Jang, Young-Sung Shin and Jae-Woo Chang*
Dept. of Computer Engineering
Chonbuk National University
Jeonju, Republic of Korea
{brilliant, twotoma, jwchang}@jbnu.ac.kr
*corresponding author: jwchang@jbnu.ac.kr

***Abstract*— Because MapReduce supports efficient parallel data processing, MapReduce-based query processing algorithms have been widely studied. Among various query types, k-nearest neighbor join, which aims to produce the k nearest neighbors of each point of a dataset from another dataset, has been considered most important in data analysis. Existing k-NN join query processing algorithms on MapReduce suffer from high index construction and computation costs which make them unsuitable for big data processing. In this paper, we propose a new grid-based k-NN join query processing algorithm on MapReduce. First, we design a dynamic grid index that represents the distribution of join datasets. Based on this index, we prune out unnecessary cells for the join with the distance-based filtering. This can reduce the data transmission and computation overheads. From performance analysis, we show that our algorithm outperforms the existing scheme up to seven times in terms of query processing time while achieving high query result accuracy.**

Keywords—cloud computing, location data protection, grid index, Bitmap encryption, density aware

I. INTRODUCTION

Recently, the amount of data is rapidly increasing with the continuous development of computation and communication capabilities. Powerful telescopes in astronomy, particle accelerators in physics, and genome sequencers in biology produce massive volumes of data to be analyzed. Hence, parallel data processing is essential to process a massive volume of data in a timely manner. MapReduce [1-3] is a framework which is introduced by Google to perform large-scale data processing in a distributed manner. MapReduce adopts a flexible computation model with a simple interface consisting of map and reduce functions which can be customized by application developers.

Among the typical analytical queries, k nearest neighbor (kNN) join is the most popular operation in analyzing large size data. The kNN join algorithm aims to combine the k nearest neighbors of each point of a dataset R from another dataset S. There has been intense research that attempt to process the kNN join query with MapReduce [4-9]. However, there are some problems to incorporate the existing kNN join query processing algorithms with MapReduce. One of the main considerations to process kNN join query on MapReduce is how to support join operations efficiently over multiple datasets. Because MapReduce is originally designed for data processing on a homogeneous datasets, it is necessary to design data partition and job assignment in a sophisticated manner to perform join operations on the heterogeneous datasets. To address this problem, W. Lu et al.[9] proposed a Voronoi diagram based kNN join algorithm using MapReduce. However, this algorithm requires high computational cost for constructing and updating the Voronoi cells. Moreover, the computational overhead of the algorithm is high because it utilizes R-tree index which is not suitable for MapReduce environment.

In this paper, we propose a new grid-based index and a k-NN join query processing algorithm on MapReduce. In the first MapReduce phase, our algorithm partitions the data in R and S into grid cells. To reduce the data transmission cost, we determine an optimized grid cell size by considering the data distribution of randomly selected samples. In second MapReduce phase, the mapper retrieves the neighboring grid cell S_j in S for each R_i in R. All objects in a set of R_i and its neighbors are assigned to the same reducer in the shuffling phase. The reducer performs kNN join for a set of R_i and its neighbors. This can reduce the data transmission and computation overhead. We show from performance analysis that our algorithm outperforms the VkNN-join up to three times, in terms of query processing time.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 introduces the basic terminologies with the formal problem definition and an overview of the proposed system. Section 4 describes our grid-based k-NN join processing algorithm. Experimental analysis of our algorithm is described in Sections 5. Finally, we make a conclusion with future work in Section 6.

II. RELATED WORK

A. *MapReduce and k-NN join*

MapReduce [1,2] is a framework for parallel processing of massive data sets. A job to be performed using the MapReduce framework is composed of two functions: the Map function and the Reduce function. Map function takes key/value pairs as input and produces intermediate results in the form of key/value pairs. The data from the map phase are shuffled to the reduce phase as input data. The Reduce function performs the user-defined function and outputs result to users. We define the *k nearest neighbor join* of two datasets R and S, where each record is interpreted as d-dimensional data in a metric space R_d.

Definition 1. k nearest neighbors (kNN)

For a given object r and a dataset S, and a constant k, the set of k nearest neighbors of r from S, is denoted as kNN(r,S). kNN(r,S) is a set of data to satisfy the following condition.

$$\forall \mathrm{o} \in \mathrm{kNN}(\mathrm{r}, \mathrm{S}), \forall \mathrm{s} \in \mathrm{S} - \mathrm{kNN}(\mathrm{r}, \mathrm{S}), |\mathrm{o}, \mathrm{r}| \leq |\mathrm{s}, \mathrm{r}|$$

Here, |r, s| refers to the distance between objects r and s and it can be calculated by using the following equation.

$$|\mathrm{r}, \mathrm{s}| = \sqrt{\sum_{1<i\leq 2} (r[i] - s[i])^2}$$

Definition 2. kNN join

For given two datasets R and S and a constant k, the k NN join algorithm integrates each r and its kNN(r,S).

$$\mathrm{R} \bowtie \mathrm{S}\{(\mathrm{r}, \mathrm{s}) | \forall \mathrm{r} \in \mathrm{R}, \forall \mathrm{s} \in \mathrm{kNN}(\mathrm{r}, \mathrm{S})\}$$

B. k-NN join algorithms using MapReduce

A research on k-NN join algorithm has been recently studied in the MapReduce context by C. Zhang et al.[10]. The authors proposed a kNN query processing algorithm based on a block nested loop join(H-BNLJ). The baseline algorithm of H-BNLJ is to partition two datasets R and S into n equivalent size blocks, respectively. Then, every possible pair of $\{R_i, S_j\}$ ($R_i \subset R$, $S_i \subset S$) is assigned into a bucket at the end of the Map phase. Finally, each reducer reads data in a bucket and performs a kNN join between R_i and S_j in the bucket.

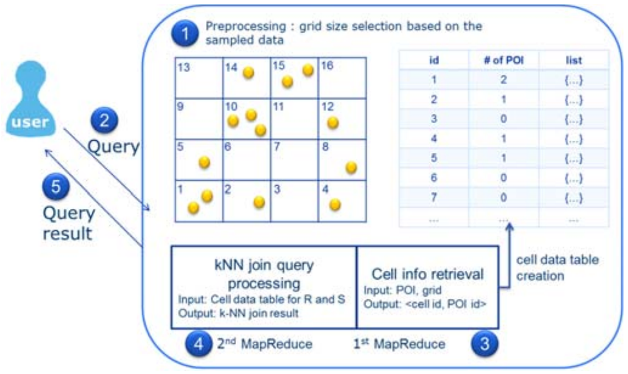
Wei Lu et al.[9] investigated a kNN join algorithm called Partitioned and Block based Join(PBJ) algorithm. For two datasets *R* and *S*, PBJ algorithm builds a Voronoi diagram-based index for each dataset by using randomly selected sample data. Then, it retrieves the neighboring partition of $R_i \in R$ from partitions in *S*. By assigning only neighboring *S* data of each R_i, PBJ algorithm can reduce the query processing time in the *Map* and *Reduce* phases. PBJ algorithm requires a preprocessing step and two rounds of MapReduce jobs to complete a join query. In the preprocessing step, it randomly selects a set of pivot objects from the input dataset *R*. The pivot objects selected is used to create a Voronoi diagram, which can partition objects in *R* and *S*. A set of objects *O* are divided into *M* disjoint partitions by assigning them to their closest pivot. When a Voronoi diagram is created, a cell-info table is generated to store <cell ID, pivot ID, pcoord(x,y), border points> for each Voronoi cell. In the first MapReduce job, a map function takes the selected pivot objects and assigns dataset *R*(or *S*) to their nearest pivot. The mappers compute statistic information about each partition R_i and store it in the HDFS. The statistic information includes the number of objects in each partition(P_i), the lower/upper bounds of data to the pivots, and a set of k nearest partitions. In the second MapReduce job, a map function finds the subset of Si from S for the partitions in R by using the statistic information. Finally, each reducer performs the kNN join between a pair of R_i and S_j, which is received from the mappers.

For computing kNN join, this algorithm uses only data in neighboring Voronoi cells from a query instead of reading the whole data. However, PBJ algorithm requires high computational cost for constructing the Voronoi diagram. Moreover, the computational overhead of the PBJ algorithm is high because the number of the candidate data in neighboring cells is increased as *k* increases.

III. K-NN JOIN QUERY PROCESING ALGORITHM USING MAPREDUCE

A. System architecture

The overall system architecture and a query processing flow are illustrated in Figure 1. First, in the preprocessing step (Fig 1.1), our algorithm selects pivots in a random manner. For this, we divide data R and S into chunks and send them to mappers to perform a pivot selection. In the reduce phase, we insert the whole data into the grid index and calculate the densities of the cells. Based on the cell density and the previous query history, we decide a proper grid size to perform a k-NN join. For finding neighbor subsets of a query, it stores the ids of neighboring cells for every grid cell. Secondly, in the first MapReduce job (Fig 1.2), it assigns the data into the grid index and generates a *SummaryTable* that indicates data partition (Fig 1.3). Thirdly, in the second MapReduce job (Fig 1.4), the algorithm searches k number of nearest neighbors from S for all data in R. Finally, the join result is sent to a query issuer (Fig 1.5).



id	# of POI	list
1	2	{...}
2	1	{...}
3	0	{...}
4	1	{...}
5	1	{...}
6	0	{...}
7	0	{...}
...	...	...

Fig. 1. System architecture and query processing flow

B. Preprocessing steip: Grid-based partitioning

In the preprocessing step, we employ an n*n gird to partition the data space. As a result, each dimension is divided into n sections and there are n^d partitions in total for <u>*d*</u> dimensional data. Then we join only the relevant partitions of R and S to reduce the data computation cost. First, it is important to find a proper partition threshold n based on both the k (for k-NN) and data popularity. For this, we generate a histogram by using a sampled dataset.

$$BS_{pi} = \begin{cases} 0, & if\ p_i is\ a\ non-empty\ cell \\ 1, & otherwise \end{cases} \qquad (1)$$

The original dataset R is divided into disjoint subsets R_1, $R_2, \ldots, R_m$ and each of them is sent to a corresponding mapper. A mapper generates a bitstring for its data by using the equation (1). A bit of a bitstring represents data existence in a corresponding grid-cell. A reducer merges the generated bit strings from mappers and calculates the number of non-empty cells. Then it determines the data partition threshold n using the number of tuples (GR), the cardinality of a grid cell (CG) and an average k from the previous queries, as shown in equation (2).

$$C_G = \frac{C_R}{N^d} = average\ k \times 2 \tag{2}$$

$$N_P = \sqrt[d]{\frac{C_R}{C_G}} \quad \begin{array}{l} C_R\text{: the cardinaltiy of all tuples in R} \\ C_G\text{: the cardinality of a grd cell} \\ average\ k \\ \text{: a statistic value from the previous queries} \end{array}$$

C. 1st MapReduce: Summary table generation

In the first MapReduce phase, a mapper reads input splits and inserts them into grid partitions. The mapper outputs each object o along with its grid partition id, data origin (*R* or *S*) and distance to the edges of the corresponding grid cell. Then, a reduce function aggregates statistic information for each grid partition. For a partition R_i of R, a data in R_i is represented as <R_i, (data ID, [x,y])>. For a data from S, it is required to store the neighboring partition of R to perform a join. Hence it is stored in the form of <R_i, (S_j, data ID, [x,y])>. In this way, objects in each partition of R and their potential k nearest neighbors will be sent to the same reducer. Because the algorithm can minimize the size of candidate set with the number of data in each partition, the performance of join process can be improved.

D. 2nd MapReduce: k-NN join processing

Based on the statistics from the *SummaryTable*, *Mappers* of the second MapReduce job find the subset S_j for each subset R_i. Each reducer performs the kNN join between a pair of R_i and S_j. To guarantee the accuracy of a query result, we perform a cell expansion from the query cell to retrieve the genuine k-NN result in neighboring cells. In order to reduce the cost of distance computations, we create a priority queue PQ with size k and store the current k-NN result of the query in ascending order. When a new grid cell C_{new} is included in the query region, we compute Euclidean distances between all the data in C_{new} and the query. By comparing the sorted distances with *dist*(PQ.top), we can reduce the computation cost. To determine whether or not a cell is included in a query region, the *dist*(PQ.top) is set as the distance threshold. For a query tuple $r \in$ R, we compute the distances between *r* and all the edges of the cell including *r*. If the distance to the closest edge(E_i) is smaller than *dist*(PQ.top), we expand the query cell toward E_i. Then, we update the PQ with all the data in the expanded cell. This algorithm stops when the distances to the remaining edges exceed *dist*(PQ.top).

Algorithm 1 describes the details of kNN join procedure that is described in the second MapReduce job. For each object r ∈ R, the map function generates a new <key, value> pair in which the key is its grid cell id, and the value is its coordinates (line 1-3). For each object s ∈ S, the map function creates a set of <key, value> pairs, where the key is its neighboring partition R_i and the value consists of its id and coordinates. Hence, the number of <key, value> pair for an object s ∈ S is the same as the number of its neighboring cells in R. The mappers retrieve the neighboring cell information from the IVList generated from the first MapReduce (line 4). In this way, all the objects in R_i and their potential k nearest neighbors in S are assigned to the same reducer. The reducer receives the partition R_i and subsets S_j and computes the kNN objects in partition R_i (line 4-15). For this, the reducer computes the distances from a query r ∈ R to the edges of its partition R_i and it computes the distance between a query and its k-NN object s (line 5-6). To guarantee the accuracy of query results, we compare the distance threshold and determine if the cell expansion is required to retrieve the final results for r (line 7-12). While the expansion condition is true, this step is repeated with data from the expanded partitions. Finally, the k-NN result is aggregated and returned to the users (line 13-15).

Algorithm 1. 2nd MapReduce

```
<Map phase>
Input: Query R, SummaryTableS, RealData S, N, NN_Cell_Info Table
Output: <RCell_NN id, SCell id, pid, x, y>
1:   For each tuple in R and S
2:   If data = R,  insert R into Grid and return <Rcid, pid, x, y>
3:   Else if data = S, retrieve NN Grid Cell ids in NN_Cell_Info
4:   For i =1 to Number of NN Cell
         Return <RCell_id, SCell_id, pid, x, y>
<Reduce phase>
Input: cell group CGi of S, grid index of R
Output: k-NN Join result
5:   Aggregate k-NN join results for all tuples Check the number of POI(P_n)
     in its cell
6:   Retrieve k-NN POI from q and calculate dist D_k between (k-NN, r_q)
7:   Expand Check(r_q, Grid Cell info, k-NN)
8:   For each {NE, NW, SE, SW } directions
         Count bit of the enclosing cells
         Return the cell_direction with count
9:   If Dvi-q <= D_k && SumCount != NULL
         Expand Cell from the vertex direction
         Retrieve all POIs in the expanded area and add them to the list
10:  Else retrieve all POIs within the Dk and add them to the result list
11:  If P_n>k
         Expand the query cell where Dist(vi) is NN && SumCount is large
12:  repeat 9-11 for k-NN result retrieval
13:  Retrieve all POIs in the expanded area and add them to the list
14   Aggregate k-NN results for all tuples and return
15   Return result to the user
```

IV. Performance Evaluation

In this section, we analyze the performance of our query processing algorithm with PBJ algorithm proposed by W. Lu et al. [9]. For this, we measure the query time of two algorithms. The H-zkNNJ algorithm is excluded because it only supports approximate k-NN join in MapReduce. We analyze the performance of kNN join on the real dataset, MODIS level 2 data [16]. It includes the daily records of sea surface temperature, a pair of <altitude, latitude> and Chlorophyll. The size of data is varied from 250M(n=70) to 1,000M(n=140). By considering the characteristics of the MODIS level 2 data, we use a distance measure as shown in Equation (3). Here, the distance dist(r_i, s_j) between two data ri and s_j is calculated by using both Chlorophyll(Chlor) and sea surface temperature (SST). The sum of α and β is 1 where *α* is a weight of Chlor and *β* is a weight of SST. The Hadoop cluster consists of 1 master node and 5 data nodes. The experimental setup for real dataset includes AMD Opeteron™ processor 4180 CPU with 32GB memory on Linux 3.11.0-15.

$$dist(r_i, s_i) = |\alpha \times (r_i.chlor - s_i.chlor)| + |\beta \times (r_i.sst - s_i.sst)| \quad (3)$$

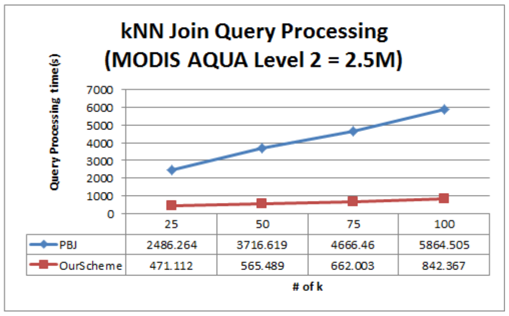
A. *Index generation time*

The pre-processing time consists of both sample selection time and index generation time. With 250M tuples, PBJ required 24.84 seconds whereas our algorithm spent 12.57 seconds on pre-processing phase. The PBJ algorithm requires almost twice more time than ours because it builds a data group using Voronoi diagram and stores the partitions into R-tree.

B. *K-NN join query processing time*

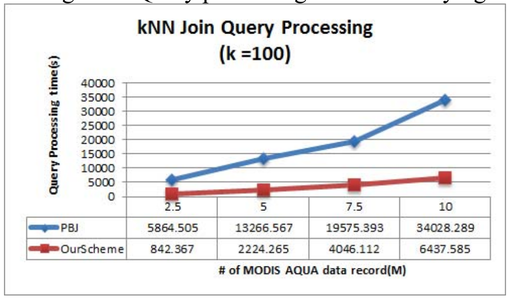
Figure 2 describes the query processing time of algorithms by using 2.5 million data of MODIS AQUA Level 2. When k is 100, the query processing time of our algorithm is 778 seconds whereas the existing one requires about 4,059 seconds. From the result, it is shown that our algorithm achieves up to 5 times better performance than PBJ by reducing the query range expansion cost. On the other hand, in case of PBJ, the number of candidate Voronoi cells is greatly increased as *k* increases.

Figure 3 describes the query processing time of k-NN join algorithms by using MODIS AQUA Level 2 whose size is ranged from 2.5 to 10 million. When the number of data is 7.5 million, the query processing time of our algorithm is 4,046 seconds whereas the existing one requires 19,575 seconds. From the result, it is shown that our algorithm achieves up to 7 times better performance than PBJ. This is because our algorithm greatly reduces the number of candidate data transmitted to reducers by pruning out unnecessary candidate cells.



	25	50	75	100
PBJ	2486.264	3716.619	4666.46	5864.505
OurScheme	471.112	565.489	662.003	842.367

Figure 2. Query processing time with varying k



	2.5	5	7.5	10
PBJ	5864.505	13266.567	19575.393	34028.289
OurScheme	842.367	2224.265	4046.112	6437.585

Figure 3. Query processing time with varying data size

V. Conclusion

In this paper, we propose a new grid-based k-NN join query processing algorithm. Our algorithm can reduce the index construction cost by considering data distribution. To efficiently perform a k-NN join query in MapReduce, we devise a candidate cell retrieval scheme based on grid-cell information. Our algorithm retrieves only neighboring data from a query cell thus can improve the data transmission and computation overhead. We show from performance analysis that our algorithm outperforms the existing scheme up to seven times in terms of query processing time, while our algorithm achieves high query result accuracy. As a future work, we plan to expand our scheme to support various query types; skyline, top-k, reverse kNN query.

Acknowledgement

This work was supported by the Brain Korea 21 PLUS Project, NRF. This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(grant number 2013R1A1A4A01010099)

References

[1] Apache Software Foundation, “Hadoop MapRedce” [online]. Available: http://hadoop.apache.org/mapreduce

[2] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters." Communications of the ACM, vol. 51, no. 1 pp. 107-113, 2008.

[3] D. Jiang, B. C. Ooi, L. Shi, S. Wu, “The performance of MapReduce: An in-depth study,” Proceedings of the VLDB Endowment, vol. 3, no.1-2 pp.472-483, 2010.

[4] C. H. Yang, A. Dasdan, R. L. Hsiao, D. S. Parker, “Mapreduce-merge: simplified relational data processing on large clusters,” In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1029–1040, 2007.

[5] D. Jiang, A. K. H. Tung, G. Chen, “MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters,” In IEEE Trans. Knowl. Data Eng. (TKDE), vol.23, no.9, pp.1299–1311, 2011.

[6] F. N. Afrati, J. D. Ullman, “Optimizing multiway joins in a Map-Reduce environment,” In IEEE Trans. Knowl. Data Eng. (TKDE), vol.23, no.9, pp.1282–1298, 2011.

[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, Y. Tian, “A comparison of join algorithms for log processing in MapReduce,” In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 975–986, 2010.

[8] C. Zhang, F Li, J. Jestes, “Efficient Parallel kNN joins for Large Data in MapReduce”, In Proc. of the EDBT, 15th International Conference on Extending Database Technology, 2012.

[9] W. Lu, S. Shen, B. Chen, O. Chin, "Efficient processing of k nearest neighbor joins using mapreduce,” In Proceedings of the VLDB Endowment, vol.5, no.10, pp.1016-1027, 2012.

[10] M. M. Breuning, H. -P. Kriegel, R. T. NG, J. Sander, “Lof: Identifying density-based local outliers,” In ACM sigmod record, vol. 29, no. 2, pp. 93-104. ACM, 2000.

[11] C. Bohm and H.-P. Kriegel. “A cost model and index architecture for the similarity join,” In Proceedings of 17th International Conference on Data Engineering, pp.411-420. IEEE, 2001.

[12] C. Xia, H. Lu, B. C. Ooi, J. Hu. Gorder, “Gorder:an efficient method for knn join processing,” In Proceedings of the 30th international conference on Very large data bases, vol.30, pp.756-767. VLDB Endowment, 2004.

[13] C. Yu, B. Cui, S. Wang, and J. Su, “Efficient index-based knn join processing for high-dimensional data,” Information and Software Technology, vol.49, no.4, pp.332-344, 2007.

[14] B. Yao, F. Li, P. Kumar, “K nearest neighbor queries and knn-joins in large relational databases (almost) for free”, In Proceedings of 26th international conference on Data engineering (ICDE), 2010.

[15] C. Yu, B. C. Ooi, K. -L. Tan, H. V. Jagadish, “Indexing the distance: An effieicnt method to kNN processing”, In VLDB, vol.1, pp.421-430, 2001.

[16] MODIS Aqua data, Available: http://modis.gsfc.nasa.gov/data/