IEICE TRANS. INF. & SYST., VOL.E101–D, NO.4 APRIL 2018

PAPER Special Section on Intelligent Information and Communication Technology and its Applications to Creative Activity Support

Grid-Based Parallel Algorithms of Join Queries for Analyzing Multi-Dimensional Data on MapReduce

Miyoung JANG[†], ***Nonmember and*** **Jae-Woo CHANG**[††a)], ***Member***

SUMMARY Recently, the join processing of large-scale datasets in MapReduce environments has become an important issue. However, the existing MapReduce-based join algorithms suffer from too much overhead for constructing and updating the data index. Moreover, the similarity computation cost is high because the existing algorithms partition data without considering the data distribution. In this paper, we propose two grid-based join algorithms for MapReduce. First, we propose a similarity join algorithm that evenly distributes join candidates using a dynamic grid index, which partitions data considering data density and similarity threshold. We use a bottom-up approach by merging initial grid cells into partitions and assigning them to MapReduce jobs. Second, we propose a k-NN join query processing algorithm for MapReduce. To reduce the data transmission cost, we determine an optimal grid cell size by considering the data distribution of randomly selected samples. Then, we perform kNN join by assigning the only related join data to a reducer. From performance analysis, we show that our similarity join query processing algorithm and our k-NN join algorithm outperform existing algorithms by up to 10 times, in terms of query processing time.

key words: *MapReduce based join query processing, similarity join algorithm, k-NN join algorithm, grid partitioning method*

1. Introduction

Recently, the amount of public data has been rapidly increasing due to the popularity of Social Networking Services (SNS) and the development of mobile technology. The increasing volume of data has triggered new challenges about how to efficiently analyze big data. Enterprises and governments analyze this data and leverage them to support effective decision making and marketing. Analytical join queries have become important issues due to their applicability to decision making applications [1]–[3].

For such data-intensive applications, the MapReduce framework [4], [5] has attracted much interest as a new data processing framework. The MapReduce [5] application introduced by Google is used to perform large-scale data processing in a distributed manner. Consequently, there has been substantial research on analytical join query processing in MapReduce for large datasets. However, because some applications need to handle a vast amount of data, there are three key challenges to be addressed. First, because the size of the data set is huge, the data must be partitioned and processed in a distributed manner. Hence, workload-aware data partitioning techniques are required. These ensure the balance of not only the input data but also the output of each machine. Second, a sophisticated filtering technique is required because the number of comparisons grows dramatically as dataset size and dimensions increase. Finally, a main issue in processing a join query on MapReduce is how to support join operations efficiently over multiple datasets. Thus, it is necessary to design both data partitioning and job assignment in a sophisticated manner to perform join operations on multi-dimensional datasets efficiently.

Among join query processing algorithms, similarity join and k-NN join are widely studied as primitive operations for data analysis. There has been intense research that attempts to process the similarity and k-NN join queries on MapReduce [6]–[15]. In order to reduce the size of join candidates, the existing work utilizes a data partitioning scheme for proximity search. However, the existing approach has two main problems. First, the existing data partitioning schemes may cause skewing of data in some partitions, resulting in high data duplication among clusters. Second, they require high computation cost for constructing and updating the data index. In particular, Voronoi diagram-based and tree-based data partitioning schemes are not suitable for the MapReduce environment.

In this paper, we propose novel MapReduce-based join query processing algorithms that efficiently find a set of join results in a large dataset by reducing the number of candidates prior to the join computation. For this, we make use of a grid index that divides data into partitions and filters out unnecessary join candidates. We propose two algorithms for join operations on MapReduce. First, we propose a grid-based similarity join query processing algorithm. To determine a grid partitioning threshold, we compute distances among data samples. For this, the data space is primarily divided into equal-sized regions for each dimension and the number of data is counted for the regions. We finally perform data partitioning such that the difference between the number of data in a partition and the number of data in another partition is the minimum. Moreover, to guarantee the correctness of the similarity join result, we allow overlap between partitions. Each partition stores the data in an area that can be expanded by a similarity threshold. Thus, all the data within the similarity threshold for each partition can be assigned to the same reducer. Second, we propose a grid-based k-NN join query processing algorithm on MapRe-

Manuscript received March 3, 2017.
Manuscript revised August 7, 2017.
Manuscript publicized January 19, 2018.

[†]The author is with Intelligence Information Research Division, Electronics and Telecommunications Research Institute (ETRI), Yuseong-gu, Daejeon, Republic of Korea.
[††]The author is with Dept. of Computer Eng., Chonbuk National Univ., Jeonju, 561–756, South Korea.
a) E-mail: jwchang@jbnu.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2016IIP0010

duce. 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 optimal grid cell size by considering the data distribution of randomly selected samples. In the 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 their neighboring objects in S_j are assigned to the same reducer. The reducer performs kNN joins for a set of R_i and a set of S_j, thus reducing the data transmission and computation overheads.

The rest of the paper is organized as follows. Section 2 presents related work on similarity joins and k-NN joins. In Sect. 3, we propose our grid-based join query processing algorithms. Section 4 provides the performance analysis of the proposed algorithms with experimental results. The conclusions and anticipated future work are given in Sect. 5.

2. Related Work

2.1 Similarity Join Algorithms on MapReduce

The goal of similarity join is to find all pairs of records that have scores greater than a predefined similarity threshold (θ) under a given similarity function. The similarity join [5]–[11] is an essential operation in a variety of applications, including record linkage, near duplicate detection, document clustering, marketing, analysis, and data cleaning and integration. Definition 1 shows the definitions of similarity join.

Definition 1. ***Similarity join***

Given two datasets R and S, a similarity function *sim*, and a similarity threshold (θ), the similarity join algorithm retrieves all pairs of records (r, s) where their similarities are no smaller than the given threshold.

$$\mathrm{R} \bowtie \mathrm{S} = \{(\mathrm{r}, \mathrm{s}) | \forall \mathrm{r} \in \mathrm{R},\ \forall \mathrm{s} \in \mathrm{S},\ \mathrm{sim}(\mathrm{r}, \mathrm{s}) \geq \theta\}$$

A similarity join using a MapReduce job was first studied by Okcan and Riedewald [10]. The proposed algorithm (1-Bucket-Theta) uses a matrix to map regions for the assignment of balanced reduce tasks. The algorithm follows a randomized process to assign incoming tuples from S/R to a random row/column. The approximate equi-depth histograms of the inputs are constructed using two MapReduce jobs and are exploited to identify empty regions in the matrix. Turning a matrix-to-reducer mapping into a MapReduce algorithm is conceptually straightforward. For an incoming S-tuple, the map function finds all regions intersecting the row corresponding to the tuple in the matrix. For each region, it creates an output pair consisting of the region key and the tuple.

Recently, A.D. Sarma et al. [11] proposed a ClusterJoin to compute similarity joins based on metric distance functions in MapReduce. It can perform a similarity join along with various similarity measures including Euclidean distance, cosine similarity and Hamming distance. The ClusterJoin algorithm makes two key contributions. First, it designs a general filter that can prune away candidate pairs without actually computing their similarities. Second, the ClusterJoin algorithm proposes a dynamic load balancing scheme that is adaptive to data distribution with good load balancing. However, there is a high probability that the ClusterJoin data skewness is not fully solved when the data density is high. Furthermore, a bisector-based clustering algorithm using randomly sampled data may cause high data duplication among clusters.

2.2 k-NN Join Algorithms on MapReduce

The kNN join algorithm finds k closest pairs of data based on the distances [12], [13], [13]–[15]. For example, a pair (r, s) is returned such that s is the k-closest neighbor of r. In general, Euclidean distance and vector distance are widely used for the distance computation. Definition 2 shows the concept of a k-NN join.

Definition 2. ***k-nearest neighbor(k-NN) join***

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

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

where kNN(r, S) is a set of data to satisfy the following condition. Here, |r, s| refers to the distance between objects r and s.

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

Here, we introduce the existing k-NN join algorithms processing large-scale data on MapReduce. First, C. Zhang et al. recently proposed a k-NN join algorithm using a block nested loop join (H-BNLJ) on MapReduce [15]. The algorithm partitions two datasets R and S into blocks of equivalent size of $\left\{R_i, S_j\right\}\left(R_i \subset R,\ S_j \subset S\right)$, every possible pair is assigned to a bucket at the end of the map phase. Then, each reducer reads data in a bucket and performs a kNN join between R_i and S_j. Second, C. Zhang et al. also proposed the H-zkNNJ algorithm [15] that utilizes one-dimensional mapping (i.e., z-order). The H-zkNNJ works in three MapReduce phases. In the first phase, the random copies of tuples in R and S are shifted and the partitions R_i and S_j are generated. In the second phase, a pair of R_i and S_j is assigned to a block such that the algorithm finds a candidate k nearest-neighbor set from S_j for each object $r \in R_i$. In the third phase, the real k nearest neighbors are simply derived from the candidate set.

Finally, W. Lu et al. [13] proposed a kNN join algorithm called Partitioned and Block based Join (PGBJ). The PGBJ algorithm requires a preprocessing phase and two rounds of MapReduce phases to complete a join query. In the preprocessing phase, it randomly selects a set of pivot objects from the input dataset R to create Voronoi-based data partitions. The cell information table is then generated to store Voronoi cell information, including the border and neighboring cell information. In the first MapReduce job, a map function takes the selected pivot objects and assigns dataset R (or S) to their nearest pivots. The mappers compute statistical information about each partition R_i and store it in the HDFS. In the second MapReduce job, a map func-

tion finds the subset of S_i from S for the partitions R_i in R. Each reducer performs the kNN join between a pair of R_i and S_j. For computing a kNN join, the PGBJ algorithm read only data in neighboring Voronoi cells from a query, instead of reading the whole dataset. However, VkNN-join requires high computational cost for constructing and updating the Voronoi cells. Moreover, VkNN-join suffers from too much overhead because it utilizes an R-tree index that is not suitable for the MapReduce environment.

2.3 Analysis of Join Algorithms for MapReduce

Generally, the cost of communication between map and reduce phases is the most dominant cost of a map-reduce join algorithm. Notice that in an instance of the join problem, not all the inputs will be present. That is, the relations R and S will be subsets of all the possible tuples, and the output will be those triples (a, b, c) such that both R(a, b) and S(b, c) are actually present in the input data. To retrieve genuine join results, all possible subsets of R and S should be sent to mappers/reducers. Hence, some data will be duplicated and sent to multiple mappers/reducers. Consequently, it is crucial to find the minimum subsets of R and S for a join operation to reduce the data replication [19]. The data replication indicates the average number of key-value pairs that the mappers create from each input. In this paper, we focus on sophisticated grid-based data partitioning schemes that can reduce the data replication for join candidates. In the following, analysis of the join algorithms on MapReduce is provided based on five important factors that influence the query processing performance. First, *pre-processing* can provide a fast overview of the underlying data distribution or input statistics. Pre-processing may require one or multiple MapReduce jobs, which entails extra costs. Second, *pre-filtering* is employed to early discard input records that cannot be in the final join result. Third, *partitioning* is one of the most important factors because input tuples are assigned to partitions being distributed to Reduce tasks for join processing. Fourth, the *replication* (or *duplication*), of input tuples over multiple Reduce tasks is required to produce the correct join result in a parallel fashion. Finally, the *load balancing* of Reduce tasks is an important one because the overall job completion time depends on the slowest Reduce task. The comparison of join algorithms is provided in Table 1.

Table 1 Comparison of join algorithms

	Algorithms	**Pre-processing**	**Pre-filtering**	**Partitioning**	**Replication**	**Load-balancing**
Similari-ty join	1-Bucket-theta [10]	No	No	Cover join matrix	Yes	Bounds
	ClusterJoin [11]	Sampling	2D hashing	Anchor-based	Yes	2D hashing
	Our algorithm	*No*	*Dimension-compression*	*Variable-sized grid*	*Yes*	*Density-based*
k-NN join	H-BNLJ [15]	No	No	bucket	Shifted copies of R, S	Quantile-based
	H-zkNNJ [15]	No	No	Z-value based	Shifted copies of R, S	Quantile-based
	PGBJ [13]	Indexing	No	Voronoi-diagram	Adjacent cells	No
	Our algorithm	*Sampling*	*PCA*	*Grid-based*	Adjacent data	*Density-based*

3. Grid-Based Join Algorithms for MapReduce

3.1 Motivations

The problem of choosing the optimal grid size for join processing is very important for achieving good performance in parallel grid-based algorithms. To tackle the problem of unbalanced computational and communicational workload in a MapReduce framework, dense data regions must be covered by a higher number of cells to contain fewer objects in a cell. For join processing, multi-dimensional equi-height histograms lead to the almost uniform distribution of objects for grid cells. Because the construction of equi-height histograms is, however, computationally expensive, we propose a heuristic solution for MapReduce. In Table 2, we summarize the symbols used in this paper.

3.2 Grid-Based Similarity Join Algorithm for MapReduce

3.2.1 Overall Processing Flow

Our similarity join algorithm contains a preprocessing step and a MapReduce job. First, in the preprocessing step, our algorithm generates a histogram by selecting sample data. For this, we divide data R and S into equal-sized regions for each dimension and count the number of data in each region. Then, we perform data partitioning such that the difference between the number of data in a partition and the number of data in another partition is the minimum. For this, we merge the small grid into join partitions. Moreover, to guarantee the correctness of the join result, we allow over-

Table 2 Symbols used in join algorithms

Symbols	**Meaning**
R(or S)	Database relation
d	Dimensionality of the complete dataset
n	Number of grid sections in a dimension
n^d	Number of grid cells for all dimension d
D	Set of dimensions in similarity join = dimension group
k	Number of a dimension group
ε_k	ε range in a dimension group k
dist	An L_m-*norm* based distance function
partition	A group of data located in a set of cells to be joined
PID	Partition ID
k-NN	Number of nearest neighbor points to retrieve
$R \bowtie_\varepsilon S$	A similarity join of dataset R and S with similarity threshold ε
$R \bowtie_{k-NN} S$	A KNN-join of dataset R and S

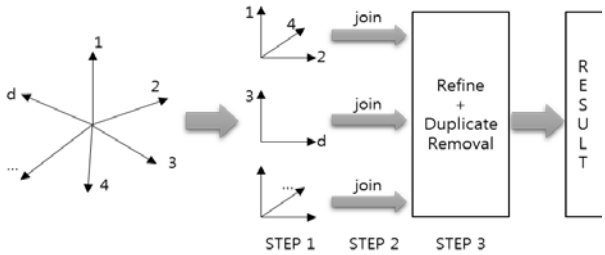


Fig. 1 Splitting of d-dimensional space into dimension groups

lap between partitions and expand a partition based on the similarity threshold. Thus, all the data within the similarity threshold can be assigned to the same reducer. Finally, mappers receive pairs of similar data partitions for joins and generate intermediate results by converting those into <key, value> pairs. In the reduce phase, a pair of similar data partitions are joined and the final result is sent to users.

3.2.2 Similarity Join Algorithm Using Variable-Length Grid

To efficiently perform similarity joinfor mult-dimensional datasets, the d-dimensional space is projected to multiple subspaces of different size. Our grid-based similarity join algorithm (GSJ-MR) calculates the candidate sets of objects in each subspace. Duplicate pairs are removed while merging the join results from subspaces. The union of the candidate sets guarantees the correct similarity join result because our algorithm can retrieve all neighboring cells of the query cell.

Instead of performing the similarity join between datasets R and S with d-dimensional space, we split the dimensions into k dimension groups such that $2 * k < d < 3 * k$ (Definition 3, and then join objects in each dimension group. The concept of dimension group is shown in Fig. 1.

Definition 3: (Dimension groups) Let $\mathbb{R}^d$ be the domain and $D = \{D_1, \ldots, D_k\}$ a set of disjoint subsets with $\bigcup_{i=1}^{k} D_i = \{1, \ldots, d\}$. We call D_i a dimension group.

In the first step, a d-dimensional space is divided into multiple different-sized subspaces. In order to reduce the high computational complexity, we combine at most *3*-dimensions into the joining subspaces. In each dimension group, data is further partitioned into grid space of variable size by considering data distribution. The main idea of our partitioning method is that it divides the dimensional space into sub-dimensional partitions using the underlying data distribution and then computes the intersections among partitions in each dimension. Hence, the intersection points become the border points of a partition. Because the subspace has at most 3-dimensionality, our approach provides both a cost-efficient computation of border area and better load-balancing when compared to an equal-sized grid. As a result, it is possible to suppress the enlargement of border areas quickly under high-dimension data. For the given d-dimensional data, when we generate subspace by grouping *3*-dimensions, $d/3$ number of subgroups will be composed.

In order to easily decide a border area, we divide each dimensional space with a sufficiently-small bin size (e.g., 1/10 of the similarity threshold), and calculate the number of objects in each bin. According to the number of grid partitions (n) per dimension, we calculate the expected number of objects per partition and then compute the near-optimal partition size in a greedy way. That is, we start with the first bin and contain the next bins in the partition until the number of objects in the partition reaches the expected threshold. After the first partition is generated, we make the second partition by including the following bins in the partition. This procedure is terminated when the last bin is included in a partition. The partitions so generated are expected to have nearly equal numbers of objects per dimension. Because the dimensional group has at most three dimensions, it is possible to easily decide border areas by the combination of partitions contained in a dimensional group. In the second step, our similarity join algorithm calculates the candidate sets of similar pairs in each subspace. The final refinement step removes potential duplicate pairs. Our similarity join algorithm obtains the similarity join result by merging the candidate sets. All the steps are implemented as a single MapReduce job.

Splitting the d-dimensional space into k subsets leads to much smaller data replication (i.e., $k \cdot 2^{\frac{d}{k}}$, instead of 2^d), which enables the processing of high-dimensional spaces. In the following, we assume that $dist_{S \subseteq \{1,\ldots,d\}}(p, q)$ is an L_m-*norm* based distance, which is generally used in the similarity join for multi-dimensional data. For $m \geq 1$ in a subspace S, the $dist_{S \subseteq \{1,\ldots,d\}}(p, q)$ is computed as $dist_S(p, q) = \sqrt[m]{\sum_{i \in S} |p_i - q_i|^m}$.

In the Map phase, we group data objects into join partitions using variable-length grids. Thus, for an object p in a cell, all the objects in its ε-neighborhood are located either in the same cell or in one of the neighboring cells. Each reducer R_i is responsible for the home cell, c_i. All objects lying in a cell c_i are sent to both the reducer R_i and the reducers of all adjacent cells, i.e., $R_j \; for \; c_j \in NC(c_i)$. Each reducer computes not only the distances between the objects in its home cell, but also the distances between the objects in the home cell and the objects in neighboring cells. To reduce data replication, each reducer considers the neighboring cells with a smaller or equal ID in every dimension because the other cells will be computed in different reducers. Because the number of neighboring cells having smaller or equal ID in each dimension is equivalent to 2^d, our algorithm can replicate data objects 2^d times, instead of 3^d times when considering all neighboring cells.

When two objects p and q from neighboring cells are processed in two separate reducers, our approach still suffers from duplicated distance computation. To avoid this, a reducer is required to differentiate between objects from different neighboring cells. For this, we adopt bit code [16] to identify the relative position of the object's cell from the home cell. The bit code consists of d bits for a d-dimensional grid where each bit corresponds to one di-

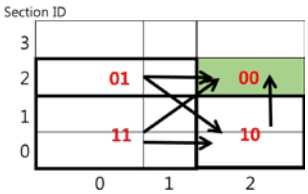


Fig. 2 Bit codes of cells when the hom cell is (2,2)

Algorithm 1. Similarity join algorithm	
<Map phase> Input : Query *R*, RealData *S*, N Output : < RCell_id, SCell_id, pid, x, y>	
1:	for each tuple in *R*(or *S*)
2:	insert *data* into Grid
3:	return <Cid, pid, x, y>
4:	Retrieve Neighboring Cell ids in a greedy way
<Reduce phase> Input : cell group CGi of *S*, grid index of *R* Output : similarity join result	
5:	For all
6:	If bitcode <= maxFlag
7:	Compute dist(p, q)
8:	If dist(p,q) <= ε
9:	Add p to the result candidate set
10:	Aggregate all result candidate sets
11:	Delete duplicate
12:	Return Final Result

mension. The home cell is represented as the bit code '0d'= 00...0 (d times). For the other cells, each bit indicates whether the position of this cell deviates from that of the home cell in the corresponding dimension. Using the bit codes, a reducer can decide which cells can be skipped. For example, when bit codes are depicted as red numbers in the cells (Fig. 2), the upper-right green cell represents the home cell with bit code 00. The thick bordered rectangles represent the sets of cells with the bit codes 01, 10 and 11. The bit codes differ from the home cells' for the dimensions where the cells lie in different partitions. The arrows between the cells indicate their distance calculation in the reducer of the home cell.

Algorithm 1 shows the pseudo code of the MapReduce phase. The map function computes a cluster for the corresponding dataset. The result of map phase is sent to the reducer in the form of <cluster id, (PID, x, y)> (line 1-4). Each datum in a cluster is compared with similar objects in a reduce function (line 5-11). The result of the similarity join is the union of the outputs of all reducers (line 12).

3.3 Grid-Based k-NN Join Algorithm for MapReduce

3.3.1 Overall Processing Flow

The main objective of our k-NN join is to process large-scale data on MapReduce. The overall architecture of our k-NN join query processing algorithm is illustrated in Fig. 3. First, in the preprocessing phase (Fig. 3 ①), our algorithm randomly selects pivots. For this, we divide data R and S into chunks and send them to mappers for selecting pivots. In the reduce step, our algorithm inserts the whole dataset into the grid index and calculates the densities of all the cells. Based on the cell density and the previous query history, we decide an optimal grid size to perform a k-NN join. To find the cells to be visited for a query, our algorithm stores the ids of neighboring cells of each grid cell. Second, in the first MapReduce phase (Fig. 3 ②), our algorithm assigns the data in the grid index and generates a summary table that represents the data partitions (Fig. 3 ③). Third, in the second MapReduce phase (Fig. 3 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. 3 ⑤).

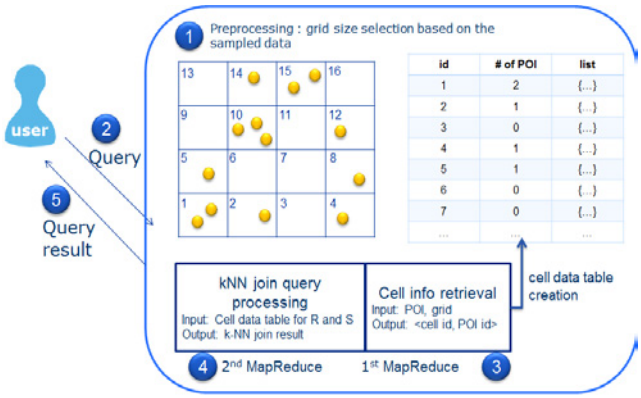


Fig. 3 Overall architecture of our k-NN join algorithm

3.3.2 Data Partitioning for Multi-Dimensional Data

In the preprocessing phase, we employ the Grid Order [17] based on a Principal Components Analysis (PCA) technique that transforms the union space of the two input datasets R and S into a single principal component space. The reason why we use the Grid Order is that it can provide high scalability for multi-dimensional data because it can support efficient k-NN join processing on MapReduce by filtering out unpromising data partitions. The Grid Order divides a grid into l^d rectangular cells, where l is the number of partitions per dimension. The transformed datasets being uniformly distributed are assigned to grids such that objects with close proximity always lie in the same grid. That is, when data are ordered based on the Grid Order, objects within the same cell are grouped together. Therefore, given a partition B containing m objects (i.e., $p_1, p_2, \ldots, p_m$), we can calculate a bounding box covering all objects in the partition by examining both the first object p_1 and the last object p_m of the ordered data. The bounding box of B can be represented as the low-left point E = $\langle e_1, e_2, \ldots, e_d \rangle$ and the high-right point T = $\langle t_1, t_2, \ldots, t_d \rangle$. Thus, we can measure the similarity of two partitions of G-ordered data as the distance between their bounding boxes. For this, we use the L_ρ distance

metric, where

$$dist(p,q) = \left(\sum_{i=1}^{d} |p.x_i - q.x_i|^{\rho} \right)^{1/\rho}, \ 1 \le \rho \le \infty.$$

Even though we use L_2 (the Euclidean distance) as an example, the proposed technique can be directly applied to L_ρ metrics, such as the Manhattan distance (L_1) and the maximum distance (L_∞). To determine the similarity between two bounding boxes BR and BS, we compute their minimum distance, i.e., MinDist(BR, BS). If the MinDist(BR, BS) is smaller than the pruning distance, BR is considered the join candidate of BS.

Definition 4. (MinDist of G-ordered Data) The minimum distance of two bounding boxes BR and BS is defined as

$$MinDist(\mathrm{BR}, \mathrm{BS}) = \sum_{k=1}^{d} d_k^2 \ \ such\ that$$
$$d_k = \max(b_k - u_k,\ 0) \ \text{where}$$
$$b_k = max(BR.e_k,\ BS.e_k),\ u_k = min(BR.t_k,\ BS.t_k).$$

Finally, the BNL (Block Nested Loops) join algorithm is executed in the second MapReduce phase, as described in the following subsection, for solving the KNN join query of G-ordered partitions in datasets R and S.

3.3.3 k-NN Join Algorithm

Our k-NN join query processing consists of two main MapReduce phases. The first MapReduce phase generates a summary table for join dataset R and S. Using this information, the second MapReduce phase performs a k-NN join while avoiding unnecessary data processing overhead.

(1) 1st MapReduce for summary table construction

In the first MapReduce phase, a mapper inserts input datasets into grid partitions. The mapper outputs each record along with its grid partition ID (G-order) and data origin (R or S). For R datasets, the neighboring cell information in G-order is stored. Next, a reduce function aggregates statistical information for each grid partition. A datum in partition R_i of R is represented as <R_i, (data ID, *coord*)> where *coord* represents the coordination of data. To perform a join between R and S, a data from a partition S_j of S is represented as <R_i, (S_j, data ID, coord)> because it is required to have its neighboring partition R_i in R. For every partition of R, a summary table (ST) maintains the partition ID, neighboring s data ID where s $\in S$, and the number of objects in the partition. Using ST, our algorithm can find the neighboring S_j for R_i, thus reducing overhead to compute a candidate set by minimizing the size of the candidate set. Therefore, the performance of our k-NN join algorithm can be improved.

Algorithm 2 shows the pseudo code of the first MapReduce phase of our algorithm. The map function inserts data into grid cells (partitions) and computes the cell id of each tuple (lines 1-5). The result of mapper is sent to the reducer,

Algorithm 2. 1st MapReduce phase	
<Map phase> Input : original data R, S, initial gird parameter (N_P) Output : <Cell_ID, POI_ID, v(data)>	
1	calculate Cell_ID of all data in sub Group(R or S)
2	**If** data in R
3	Return <R_Cell_ID, { POI_ID, v(data)}>
4	**Else** (data in S)
5	Return < S_Cell_ID, POI_ID, v(data)}>
<Shuffle phase>	
6	Aggregate map results based on the key(R_ Cell_ID)
7	Group the Cell_IDs based on the # of reducers
<Reduce phase> Input : <Cell_ID, POI_ID, v(data)> Output : SummaryTable S	
8	Generate a grid index of S by aggregating data division results

in the form of <Cell_ID, (POI_ID, v(data))> where POI_ID means the original data ID. The data value is transformed into a vector and is represented as v(data). The reduce function generates the summary table for R and S by merging the grid cell information (lines 6-8).

(2) 2nd MapReduce for k-NN join processing

Algorithm 3 shows the pseudo code of the second MapReduce phase of our algorithm. Based on statistics from the ST, the Mappers in the second MapReduce phase find the neighboring subset S_j for each subset R_i (lines 1-10). 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 a cell including a query to its neighboring cells. To reduce the cost of distance computations, we create a priority queue PQ with size k and store the current k-NN result in PQ in ascending order. When a new grid cell C_{new} is included in a query region, we compute distances between all the data in C_{new} and the query. By comparing the sorted distances with the current k-NN results stored in PQ, we can reduce the computation cost. To determine whether or not a cell is included in a query region, the maximum distance stored in PQ (max_dist) should be set as a distance threshold (lines 11-12).

For a given query q where q $\in R_i$ a reducer computes the distances between q and the edges of the cell including q. If the distance to the closest edge (CE) of the cell is smaller than max_dist, we expand the query cell toward CE (line 13). Then, we update the PQ with all the data in the expanded cell (lines 14-18). This algorithm stops when the distance to the remaining edges exceeds max_dist. Finally, the aggregated join result for all tuples in R is returned to the user (lines 19-20).

3.3.4 Cost Analysis for k-NN Join

By grouping the k nearest neighbors for all objects in a partition P_i^R, we assign $\forall s \in S_j$ to P_i when $|s, P_i^R| \ge \mathrm{dist}(h)$. Intuitively, by selecting a larger number of grid-cells, the bound of the kNN distance for all objects in each partition

Algorithm 3. 2nd MapReduce phase

<Map phase>
Input: Query R, SummaryTableS, RealData S, Grid parameter Np, NN_Cell_Info Table
Output: <R_Cell_ID, S_Cell_ID, PID, v(data)>

1: **For** each tuple in R and S
2: **If** data = R,
3: Insert R into Grid
4: **Return** < R_Cell_ID, PID, v(data)>
5: **Else if** data = S,
6: Insert S into Grid and compute S_Cell_ID
7: Retrieve NN R_Cell IDs of S_Cell_ID in NN_Cell_Info
8: **For** the Number of NN R_Cells
9: **Return** < R_Cell_ID, S_Cell_ID, PID, v(data)>
10: **End for(line 1)**

<Reduce phase>
Input: Map results, NN_Cell_Info Table
Output: k-NN Join results

11: **Check** the number of POIs (P_n) in each cell
12: **Retrieve** k-NN POI p_k from q and calculate distance D_k between (p_k, q)
13: **Expand** condition check(q, NN_Cell_Info Table, p_k)
14: **If** $P_n < k$
15: **For** each NN cells in G-order of R
16: **Retrieve** all POIs in the expanded area and add them to the candidate list
17: **Else if** $P_n > k$
18: **Retrieve** all POIs within D_k and add them to the result list
19 **Aggregate** k-NN results for all tuples in R
20: **Return** result to the user

of R will become tighter because we can split the dataset into a set of grid partitions with finer granularity. By enlarging the number of grid cells, each object from R ∪ S is able to be assigned to a grid cell with a smaller distance, which reduces both $|s, P_i^R|$ and the upper bound $U(P_i^R)$ for each partition P_i^R. Hence, in order to minimize the replicas of objects in S, it is required to select a larger number of grid cells. However, in this way, it might be impractical to provide a single reducer to handle each partition P_i^R. A simple way to cope with this problem is to divide the partitions of R into disjoint groups and take each group as R_i. In this way, S_i needs to be refined accordingly. By default, let $R = \bigcup_{1 \le i \le N} G_i$, where G_i is a group consisting of a set of partitions for R and $G_i \cap G_j = \emptyset$ where $i \neq j$.

Theorem 1. Given partition P_i^R and group G_i, $\forall s \in P_j^S$, the necessary condition that s is assigned to G_i is:

$$dist(\text{kNN}, P_i^R) \ge mindist(s, P_i^R) \quad (1)$$

where $dist(k)$ is the k-NN distance between a query $\forall q \in P_i^R$ and $s \in P_j^S$.

Proof. A data s in P_j^S is assigned to G_i as long as there exists two partitions P_j^S and P_i^R such that $dist(\text{kNN}, P_i^R) \ge mindist(s, P_i^R)$. If $dist(\text{kNN}, P_i^R) \ge mindist(s, P_i^R)$ and a pair of P_j^S and P_i^R are disjoint, the data s is a k-NN candidate for a query $\forall q \in P_i^R$. Consequently, $\forall s \in P_j^S$ should be included in G_i for further join processing.

Apparently, the average number of replicas of objects in a dataset S is reduced because duplicates in S are eliminated. According to Theorem 1, we can easily derive the number of all replicas (denoted as RP(S)) as follows.

Theorem 2. The number of replicas of objects in a dataset S that are distributed to reducers is:

$$\text{RP(S)} = \sum_{\forall G_i} \sum_{\forall P_j^S} \left| \left\{ s | s \in P_j^S \wedge dist(k) \ge mindist(s, P_i^R) \right\} \right|$$

Proof. From Theorem 1, a data s is assigned to G_i as long as there exists two partitions P_i^R and P_j^S such that $dist(\text{kNN}, P_i^R) \ge mindist(s, P_i^R)$. Because each G_i for a dataset R is distributed and computed on data nodes, a datum s where $\forall s \in P_j^S$ is duplicated as many times as the number of P_i^R that satisfies the distance threshold Eq. (1).

4. Performance Analysis

In this section, we compare the performance of the proposed join algorithms with those of the state-of-the-art join algorithms in terms of query processing time. Because both the proposed algorithms and the existing algorithms are proven to provide 100% accuracy in a query result, we exclude the performance evaluation of our algorithms in term of the query result accuracy.

4.1 Dataset Description

To prove the efficiency of the proposed join algorithms, we carried out some experiments with two kinds of datasets: synthetic and real datasets. The detailed description of the datasets is as follows.

Synthetic data The synthetic dataset was generated by using Generate Spatio Temporal Data (GSTD) with three different data distribution in a square Euclidean space [20]: UNIFORM, GAUSSIAN, SKEWED dataset. The SKEWED datasets were generated by following the Zipf distribution where the skewness parameter θ was varied from 0.0 to 0.9. The size of the synthetic dataset was varied from 10 to 200 M tuples. We consider that all data sets should be distributed in an area 1×1. Figure 4 (a-c) shows the distribution of the synthetic dataset.

Real data We used two different kinds of real datasets obtained from different sources. First, we utilized the REAL dataset containing 119,898 points that have real postal ad-

(a) Uniform (b) Gaussian (c) Skewed (d) NE dataset (e) MODIS Aqua data

Fig. 4 Data distribution of the synthetic dataset

Table 3 Experimental environment

	Performance
CPU	2.9GHz Quad-Core Intel Core i5
Memory	8GB
OS	Ubuntu 12.04
Hadoop	Hadoop 2.6.0
Network	Transfer 83.8GBytes, Bandwidth 71.9 Gbits/sec

Table 4 Parameter settings

Parameters	**Settings**
Similarity threshold	0.02, 0.04, 0.06(*default*), 0.08, 0.1
#of data partitions	10, 20, 50, 100, 150(*default*), 200
# of data tuples	2.5M, 5M, 7.5M, 10M(*default*)
Data dimensionality	2, 4, 6, 8(*default*)

dresses in the northeastern America (New York, Philadelphia, and Boston, called NE dataset). Second, we obtained MODIS level 2 data from a NASA website [21] that includes daily records, such as sea surface temperature, a pair of <altitude, latitude> and chlorophyll. The size of the dataset is varied from 250MB (n = 70) to 1,000MB (n = 140) where n is the number of records. The distributions of two real datasets are plotted in Fig. 4 (d) and 4 (e).

4.2 Performance Analysis of Our Similarity Join Algorithm

In this section, we compare the performance of our similarity join algorithm with that of the existing ClusterJoin proposed by Sarma et al. [11]. For this, we measure the query processing time of two algorithms using both synthetic datasets and real datasets under different settings, such as data size, similarity threshold and the number of data partitions. For our experiment, we use a Hadoop cluster that consists of one master node and four data nodes. The experimental setup of the nodes is described in Table 3. The parameter settings of similarity algorithms are also summarized in Table 4.

4.2.1 Performance Comparison with 2-Dimensional Datasets

For similarity join processing on MapReduce, it is important to reduce the size of join candidates by grouping relevant join partitions. Figure 5 shows the computation time of each processing step for similarity join algorithms. Due to the page limitation, we include only the experimental results with default settings and the results with other settings would show patterns similar to that in Fig. 10. As shown in the figure, the most important step is the 2nd MapReduce (i.e., join phase), which consumes at least 70% of the overall processing time for all algorithms.

Figure 6 shows the query processing time of the similarity join algorithms with varying data size. When the number of tuple was 10M, the query processing time of our algorithm was 2,181s, whereas ClusterJoin required 2,913s. From the result, it is seen that our algorithm achieved up

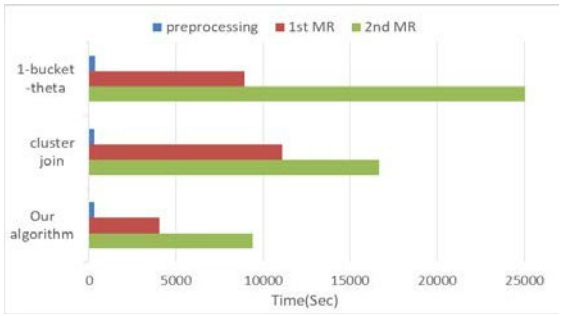


Fig. 5 Computation time for each processing step

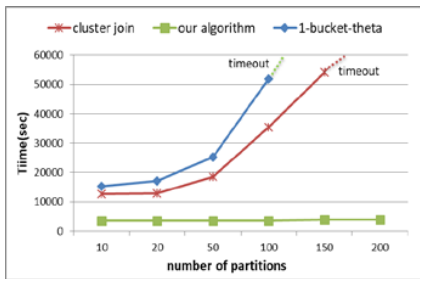


Fig. 6 Query processing time with varying data size

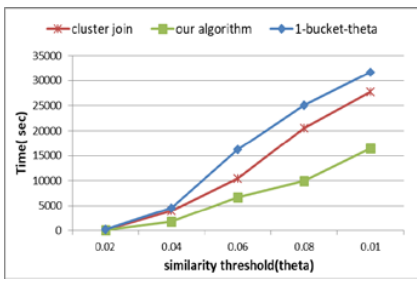


Fig. 7 Query processing time with varying similarity threshold

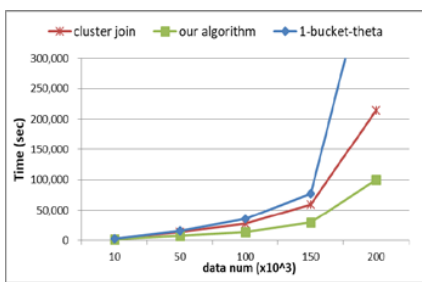


Fig. 8 Query processing time with varying number of data partitions

to 2-times better performance than ClusterJoin because our sophisticated algorithm performs data partition based on the data distribution. In the case of ClusterJoin, the number of candidate clusters is greatly increased as the number of data increases. This is because hash-based clustering does not guarantee optimal data distribution among the clusters.

Figure 7 shows the query processing time of the similarity join algorithms with varying similarity threshold.

When the similarity threshold was 0.08, the query processing time of our algorithm was 9,953s whereas ClusterJoin required 20,535s. From the result, it is seen that our algorithm achieved up to 2.5-times better performance than with ClusterJoin. As the similarity threshold increases, the number of duplicated data among clusters is greatly increased. Thus, ClusterJoin suffers from radical performance deterioration because hash-based partitioning shows worse performance with densely populated data. On the other hand, our grid-based partitioning algorithm generates clusters based on the data distribution of the sample dataset.

Figure 8 shows the query processing time of the similarity join algorithms with varying number of data partitions. The number of partition indicates the number of clusters used to assign data into the MapReduce. Hence, it is important to find an appropriate number of partitions to process a query in a distributed manner. Our algorithm shows the best performance when the number of partitions is less than 100, whereas ClusterJoin shows the best performance when the number of partitions is 10. This means that ClusterJoin suffers from data skewness and high duplication among partitions when the number of partitions increases. On the other hand, our algorithm can perform better data partitioning by considering data distribution.

To provide a distributed computation efficiently in MapReduce, it is important to partition data evenly. Hence, a workload-aware data partitioning technique is vital because it can ensure the balance of both input data and output data for each machine. The processing time of MapReduce is generally dominated by the node that finishes last. Therefore, we measured the balance ratio by dividing the average number of data by the desired number of data in a partition.

$$\text{balance\_ratio} = \frac{\textit{AVG num of data in real partitions}}{\textit{Desired num of data in a partition}}$$

Figure 9 and Fig. 10 plot the balance ratio of our algorithm and ClusterJoin for the syntactic data with uniform distribution and the real NE data, respectively. Here the value close to '1' means the even distribution of data among partitions. In both syntactic and real datasets, our algorithm showed a smaller balance ratio than that of ClusterJoin when the number of partitions varied from 10 to 200. It is seen that our algorithm provides more even data distribution among partitions than the existing ClusterJoin. From the result, we selected 150 partitions as the default, which shows tolerable performance yet support efficient parallel computation for all algorithms.

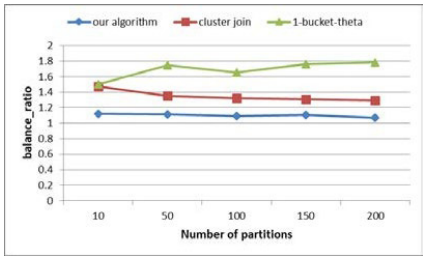


Fig. 9 Data partitioning for synthetic dataset

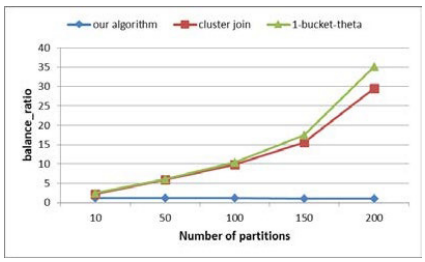


Fig. 10 Data partitioning for real NE dataset

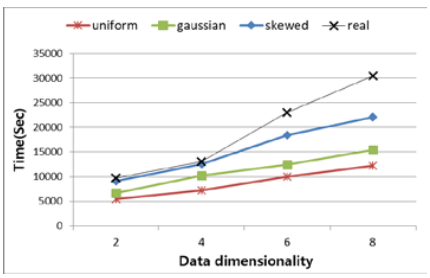


Fig. 11 Query processing time with varying data dimensions

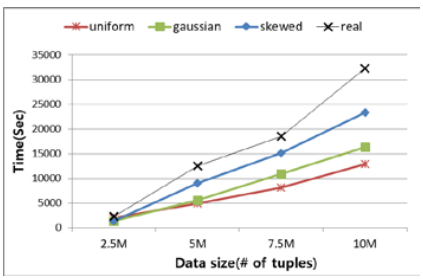


Fig. 12 Query processing time with varying data sizes

4.2.2 Performance Analysis with Multi-Dimensional Datasets

To evaluate the effect of data dimensionality on the join performance, we use both a syntactic dataset and a real MODIS dataset with multi-dimensional data. Because we have already shown in the previous section that our similarity join algorithm is much better that the existing ClusterJoin, we exclude ClusterJoin from a performance analysis with multi-dimensional data. Moreover, as the number of data dimensions increases, ClusterJoin is worse than our similarity join algorithm because our algorithm can provide more uniform data distribution among partitions than the ClusterJoin.

First, we did the performance analysis of our similarity join algorithm by varying the number of dimensions from 4 to 8, as shown in Fig. 11. We can see that the efficiency of our algorithm deteriorated with increased dimensionality. In the case of the real MODIS dataset, our algorithm required 13,045s with 4-dimensional data, whereas it required

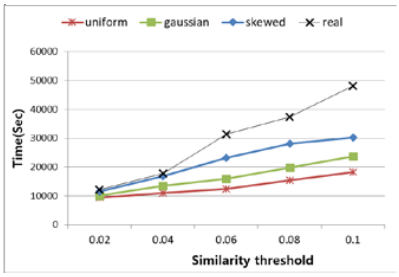


Fig. 13 Query processing time with varying similarity threshold

30,508s to process 8-dimensional data. The similarity computation cost increases linearly with the data dimensionality. However, because we transform high-dimensional data into lower dimensionality using subspace, performance degradation due to increased dimensionality is not much greater.

Second, we present the effect of data size by varying the number of tuples in datasets from 2.5 to 10 M (Fig. 12). In the case of the real MODIS dataset, our algorithm required 2,357s with 2.5M data, whereas it required 32,284s to process 10M data. Because our algorithm employs a variable-sized grid partitioning technique, it can greatly reduce the distance computation overhead for multi-dimensional data. Therefore, our algorithm provides scalable performance even for large datasets.

Finally, we investigate the influence of the similarity threshold on the query processing time as shown in Fig. 13. In case of the real MODIS data, the query processing time was increased by almost 3-times as the similarity threshold increased from 0.02% to 0.1%. From the results we see that the elapsed time of our algorithm increased moderately with increase of the similarity threshold. This is because our similarity join algorithm employs a variable-sized grid partitioning technique that can evenly distribute data into partitions. It is very important to minimize the job completion time in MapReduce by balancing the overall workload.

4.3 Performance Analysis of Our k-NN Join Algorithm

We compare the performance of our k-NN query processing algorithm with that of the PGBJ algorithm proposed by W. Lu et al. [13]. The H-zkNNJ algorithm was excluded from our experiments because it supports only an approximate k-NN join on MapReduce. We did two experiments as follows. First, we evaluated the performance of k-NN joins (k = 100) for the synthetic dataset using a Hadoop cluster consisting of one master node and seven data nodes. The size of the synthetic dataset is varied from 2.6 to 250 M. The experimental setup for the synthetic dataset is described in Table 5. Second, we evaluated the performance of k-NN joins for the real MODIS level 2 dataset [21] using a Hadoop cluster consisting of one master node and five data nodes. The experimental setup for the real dataset is described in Table 5. The parameter settings of k-NN join algorithms are also summarized in Table 6.

First of all, we measured the pre-processing time of our algorithm and the existing PGBJ. The pre-processing time consists of both sample data extraction time and index generation time. For 250M tuples, the pre-processing time of PGBJ was 24.84s whereas our algorithm required 12.57s. The PGBJ algorithm required almost twice as much time than our algorithm because it performs data partitions using Voronoi diagrams and stores the partitions in *R*-tree.

Table 5 Experimental environments

	Syntactic dataset	MODIS dataset
CPU	2.9GHz Quad-Core Intel Core i5	AMD Opteron Processor 4180
Memory	8GB	32GB
OS	Ubuntu 12.04	
Hadoop	Hadoop 2.6.0	
Network	Transfer 83.8GBytes, Bandwidth 71.9 Gbits/sec	

Table 6 Parameter settings

Parameters	Settings
Number of nearest neighbors(k)	20(*default*), 40, 60, 80, 100
Dataset size (# of tuples)	2.5M, 5M, 7.5M, 10M(*default*)
# of data partitions n (n*n)	10, 20, 50, 70(*default for* $k \leq 60$), 100 (*default for* $k > 60$)
Data dimensionality	2, 4, 6, 8(*default*)

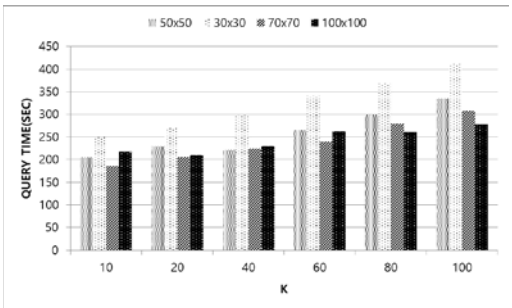


Fig. 14 Query processing time with varying k and grid sizes

The proper number of grids n is highly dependent on the actual dataset. If data points in a dataset are highly populated, we choose a small value of n. On the other hand, if the data points are sparse, we set a large value of n to partition data. In Fig. 14, we measured the query processing time with varying number of grid partitions. When k is less than or equal to 60, the medium-sized partitions with 70*70 grid cells show the best performance. When k was greater than 60, the small-sized partitions having 100*100 grid cells show the best performance. According to the results, we set the number of grid cells to show the best performance.

Figure 15 shows the computation time of each processing step for k-NN join algorithms. Due to the page limitation, we include only the experimental results with default settings using the real datasets. The 2nd MapReduce phase of our algorithm consumed 72% of overall processing time whereas that of PGBJ required 87% of the overall query processing time. This indicates the importance of filtering effi-

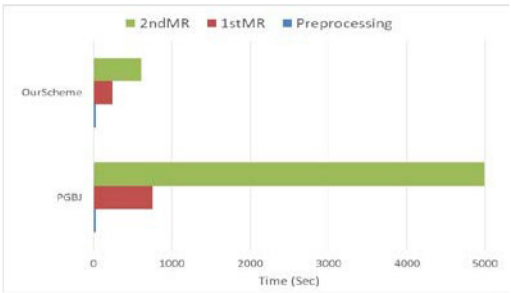
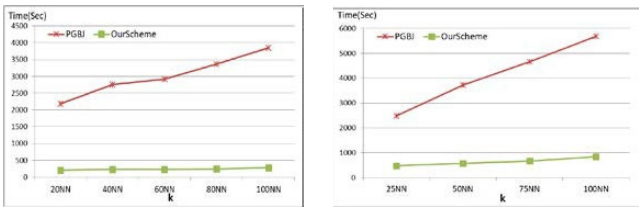
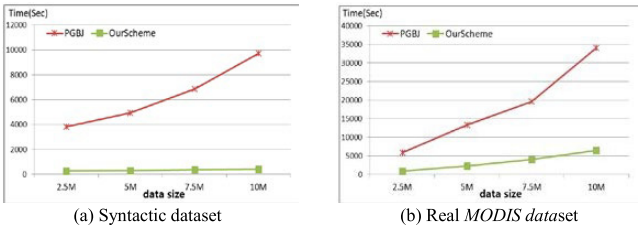


Fig. 15 Computation time for each processing step



(a) Syntactic dataset (b) Real *MODIS data*set

Fig. 16 Query processing time with varying k



(a) Syntactic dataset (b) Real *MODIS data*set

Fig. 17 Query processing time with varying data size

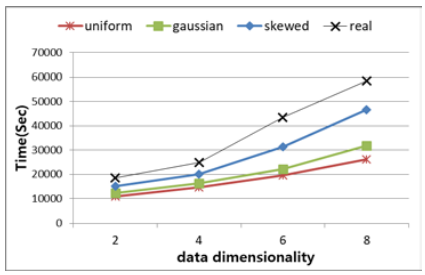


Fig. 18 Query processing time with varying data dimensions

ciency when processing a join operation on MapReduce.

4.3.1 Performance Comparison with 2-Dimensional Datasets

Figure 16 (a) shows the query processing time of the k-NN join algorithms using one million data for the syntactic dataset with uniform distribution. When k was 20, the query processing time of our algorithm was 207s whereas PGBJ required 2,189s. Figure 16 (b) shows the query processing time of two algorithms by using 2.5 million data of MODIS AQUA Level 2. When k was 100, the query processing time of our algorithm was 778s whereas the existing PGBJ required 4,059s. From the result, it is shown that our algorithm achieved up to 5-times better performance than the existing PGBJ, because our algorithm can reduce the cost of expanding a query range. In the case of PGBJ, the number of candidate Voronoi cells is greatly increased as k increases.

Figure 17 (a) shows the query processing time of the algorithms with varying data size, in the case of the uniform syntactic dataset. When the number of data was two million, the query processing time of our algorithm was 382s whereas the existing PGBJ required 6,865s. Figure 17 (b) shows the query processing times of two algorithms with the real dataset, of which the size ranged from 2.5 to 10 million tuples. When the number of data was 7.5M, the query processing time of our algorithm was 4,046s whereas the existing PGBJ required 19,575s. From the result, it is shown that our algorithm achieved up to 7-times better performance than PGBJ. This is because our algorithm dramatically reduces the number of candidate data transmitted to, by greatly pruning out unnecessary candidate cells.

4.3.2 Performance Analysis with Multi-Dimensional Datasets

In this experiment, it is certain that PGBJ shows the worse performance than our algorithm for high dimensional data, due to the rapid increase in the number of candidate Voronoi cells; hence, we excluded PGBJ. First, we evaluate the effect of data dimensionality on the join performance by varying the number of dimensions (4-8). In Fig. 18, we can see that the efficiency of our algorithm deteriorated with increase of dimensionality. In the case of the real MODIS dataset, the query processing time with 4-dimensional data was 24,848s, whereas the query processing time with 8-dimensional data required 58,537s. This is mainly because the cost of similarity computation increases linearly with the data dimensionality. However, performance degradation due to the increase of dimensionality is not as high as before because we use the G-order to transform high-dimensional data to lower dimensionality. Second, we presented the effect of data size by varying the number of tuples in datasets from 2.5 to 10 M, as shown in Fig. 19. For this, we performed the 20-NN join by using the real 8-dimensional MODIS datasets. When the data size was 2.5M, the elapsed time of our k-NN join algorithms was 3,437s, whereas it was 22,594s with 10M data. Because our algorithm employs a distance computation reduction technique that alleviates the distance computation cost for high dimensional data, it provides scalable performance for large datasets. Finally, we evaluate the effect of the number of nearest neighbors (i.e., k) by varying k from 20 to 100, as shown in Fig. 20. In the case of the real MODIS data, the query processing time increased by 10%, as k increased from 20 to 100. We can see from the results that the elapsed time of our algorithm increased moderately with the increase of k. This is because our algorithm can re-

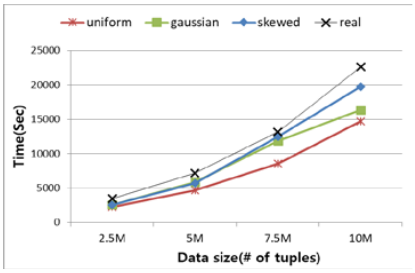


Fig. 19 Query processing time with varying data sizes

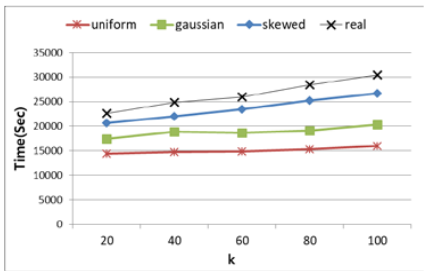


Fig. 20 Query processing time with varying k

duce distance computation using a subset of dimensions for similarity computation.

5. Conclusions

In this paper, we proposed two grid-based join algorithms for processing big data on MapReduce. First, we proposed a grid-based similarity join algorithm to evenly distribute data into partitions and to perform a join in a parallel way. To this end, we designed a dynamic grid index considering data distribution. Our algorithm reduced data computation and communication costs by sending only relevant data to the same reducer when performing a join operation. Second, we proposed a grid-based k-NN join query processing algorithm. We devised a candidate cell searching technique based on grid-cell information. Thus, our k-NN join algorithm can gain access only to the neighboring cells from a query cell and sends them as the input of a MapReduce job. This can reduce the data transmission and computation costs. We show from our performance analysis that our algorithm outperforms the existing algorithm up to seven times in terms of query processing time, while our algorithm achieves 100% query result accuracy.

In future work, we plan to extend our algorithms to support various types of join queries, such as skyline and reverse skyline queries.

Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0113-17-0005, Development of a Unified Data Engineering Technology for Largescale Transaction Processing and Realtime Complex Analytics) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant number 2016R1D1A3B03935298). This research was also supported by "Research Base Construction Fund Support Program" funded by Chonbuk National University in 2016.

References

[1] T. Lappas and D. Gunopulos, "Efficient confident search in large review corpora," Machine Learning and Knowledge Discovery in Databases, Springer Berlin Heidelberg, vol.6322, pp.195–210, 2010.
[2] J.J. Levandoski, M.F. Mokbel, and M.E. Khalefa, "Preference query evaluation over expensive attributes," Proceedings of the 19th ACM international conference on Information and knowledge management, ACM, pp.319–328, 2010.
[3] J. Lee, S.-W. Hwang, Z. Nie, and J.-R. Wen, "Navigation system for product search," Data Engineering (ICDE), 2010 IEEE 26th International Conference on, pp.1113–1116, 2010.
[4] "Hadoop-Apache Foundation," http://hadoop.apache.org/
[5] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Communications of the ACM, vol.51, no.1, pp.107–113, 2008.
[6] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," Proc. SIGIR, pp.284–291, 2006.
[7] A.Z. Broder, S.C. Glassman, M.S. Manasse, and G. Zweig, "Syntactic clustering of the web," Computer Networks, vol.29, no.8-13, pp.1157–1166, 1997.
[8] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on IEEE, 2006.
[9] Y.N. Silva and J.M. Reed, "Exploiting MapReduce-based similarity joins," Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, pp.693–696, 2012.
[10] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," Proceedings of the 2011 ACM International Conference on Management of Data, ACM, pp.949–960, 2011.
[11] A.D. Sarma, Y. He, and S. Chaudhuri, "Clusterjoin: a similarity joins framework using map-reduce," Proceedings of the VLDB Endowment, vol.7, no.12, pp.1059–1070, 2014.
[12] M.M. Breunig, H.-P. Kriegel, R.T. Ng, and J. Sander, "LOF: identifying density-based local outliers," ACM SIGMOD record, vol.29, no.2, pp.93–104, 2000.
[13] W. Lu, Y. Shen, S. Chen, and B.C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," Proceedings of the VLDB Endowment, vol.5, no.10, pp.1016–1027, 2012.
[14] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using MapReduce," IEEE 28th International Conference on Data Engineering (ICDE), IEEE, 2012.
[15] C. Zhang, F. Li, and J. Jestes, "Efficient Parallel kNN joins for Large Data in MapReduce," Proc. EDBT: 15th International Conference on Extending Database Technology, 2012.
[16] T. Seidl, S. Fries, and B. Boden, "MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce," BTW, vol.214, pp.37–56, 2013.
[17] Xia, Chenyi, H. Lu, B.C. Ooi, and J. Hu, "Gorder: an efficient method for KNN join processing," Proceedings of the 30th international conference on VLDB, vol.30, pp.756–767, VLDB Endowment, 2004.
[18] M. Jang, Y.-S. Shin, and J.-W. Chang, "A Grid-Based k-Nearest Neighbor Join for Large Scale Datasets on MapReduce," High Performance Computing and Communications (HPCC 2015), pp.888–891, IEEE, 2015.
[19] A.D. Sarma, F.N. Afrati, S. Salihoglu, and J.D. Ullman, "Upper and

lower bounds on the cost of a map-reduce computation," Proceedings of the VLDB Endowment, vol.6, no.4, pp.277–288, VLDB Endowment, 2013.

[20] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento, "On the Generation of Spatiotemporal Datasets," Proc. SSTD, vol.1651, pp.147–164, 1999.

[21] http://modis.gsfc.nasa.gov/

Miyoung Jang is a postdoc in Electronics and Telecommunications Research Institute, Korea. She received PhD in Chonbuk National University in 2016. She received the BS and MS degrees at Chonbuk National University in 2009 and 2011, respectively. Her research interests include security and privacy of databases in cloud computing.

Jae-Woo Chang is a professor in the Department of Information and Technology, Chonbuk National University, Korea since 1991. He received the BS degrees in Computer Engineering from Seoul National Univ. in 1984. He received the MS and PhD degrees from KAIST in 1986 and 1991, respectively. During 1996-1997, he was at the University of Minnesota as a visiting scholar. During 2003-2004, he worked for Penn State University (PSU) as a visiting professor. His research interests include spatial network database, context awareness and storage system.