Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 16(2): 99-105, Jun. 2018 **Regular paper**

k-NN Join Based on LSH in Big Data Environment

Jiaqi Ji[1,2] and Yeongjee Chung[2]*, *Member*, *KIICE*

[1]Department of Information Center, Hebei Normal University for Nationalities, Chengde 067000, China
[2]Department of Computer Engineering, Wonkwang University, Iksan 54538, Korea

Abstract

k-Nearest neighbor join (*k*-NN Join) is a computationally intensive algorithm that is designed to find *k*-nearest neighbors from a dataset *S* for every object in another dataset *R*. Most related studies on *k*-NN Join are based on single-computer operations. As the data dimensions and data volume increase, running the *k*-NN Join algorithm on a single computer cannot generate results quickly. To solve this scalability problem, we introduce the locality-sensitive hashing (LSH) *k*-NN Join algorithm implemented in Spark, an approach for high-dimensional big data. LSH is used to map similar data onto the same bucket, which can reduce the data search scope. In order to achieve parallel implementation of the algorithm on multiple computers, the Spark framework is used to accelerate the computation of distances between objects in a cluster. Results show that our proposed approach is fast and accurate for high-dimensional and big data.

Index Terms: Big data, High dimension, *k*-NN join, LSH, Spark

I. INTRODUCTION

k-Nearest neighbor join (*k*-NN Join) is widely used as a clustering or classification method in data mining or machine learning [1]. For classification, some high-dimensional data points are provided for training, and some unlabeled data are used for testing. The dataset *S* is defined as the training set, and *R* is used to denote a test set. The core idea of *k*-NN Join is as follows: we first calculate the distance between $r \in R$ and each datum in *S*, and then we can obtain *k* nearest neighbors. $|R|$ represents the size of *R*, $|S|$ represents the size of *S* and $|d|$ represents the dimension, so the time complexity of the traditional algorithm is $O(|d|\times|S|\times|R|)$. Hence, the cost of computation time is higher, and more CPU resources are required.

k-NN Join can be widely used in areas like classification problems [2], similar image matching [3], and related fields. *k*-NN Join is expensive to run, as it has to calculate the distance for each pair of data. Therefore, it cannot generate results rapidly in big data environments. To overcome this drawback, many researchers have proposed the use of the approximate *k*-NN Join algorithm [4, 5], which can index raw data with only a slight impact on accuracy (compared to the exact *k*-NN Join algorithm) in exchange for lowering the time of computation. However, when the data volume and data dimensions increase drastically, the time of computation increases concomitantly, and the results cannot be generated within a stipulated time-frame by employing a single computer. To solve this problem, two technologies have been proposed. (1) Parallel distributed computing, a popular solution, is widely used with big data in a cluster. Due to the improvement of parallel computing with the effectiveness of Hadoop MapReduce [6] computing in particular, the *k*-NN Join algorithm can be improved to work in the cluster. In recent years, significantly improved *k*-NN join algorithms [7-9] have been advanced, which could be used within the

Received 10 December 2017, **Revised** 18 January 2018, **Accepted** 19 January 2018
***Corresponding Author** Yeongjee Chung (E-mail: yeongjee@gmail.com, Tel: +82-63-850-6887)
Department of Computer Engineering, Wonkwang University, 460, Iksan-daero, Iksan 54538, Korea.

Open Access **https://doi.org/10.6109/jicce.2018.16.2.99** print ISSN: 2234-8255 online ISSN: 2234-8883

MapReduce framework. The idea that MapReduce can be applied in distributed processing of big data was initially proposed by Google. However, MapReduce failed to provide persuasive explanations as to the applications to process data in multiple procedures involving interactive queries or iterative algorithms. Conversely, Spark is a fast and general engine for large-scale data processing and can run programs up to 100× faster than Hadoop MapReduce in memory, or 10× faster on disk [10]. Therefore we used Spark instead of Hadoop to perform parallel computing in this study [11]. (2) Locality-sensitive hashing (LSH) [12], a novel approximate *k*-NN Join algorithm, is used on pre-indexed data.

Based on the above technologies, an algorithm named LSHS *k*-NN Join is proposed on LSH and implemented in Spark. The main mechanism of LSHS *k*-NN is that dataset S is first hashed to a different bucket by LSH; it then calculates the distance based on Spark. The experimental results indicate that the proposed algorithm is efficient and accurate for high-dimensional big data.

The main sections of this study are organized as follows. (1) Proposing an algorithm by employing LSH on the basis of Spark in a cluster. (2) Reducing the computation time of *k*-NN Join for high-dimensional big data. (3) Improving the speed of queries when it is used in production circumstances.

The remaining part of the study is outlined as follows: in Section II, some concepts, definitions and mathematical theory are introduced; in Section III, the implementation of the Spark-based LSHS *k*-NN Join algorithm is described in detail; Section IV firstly investigates the experimental environment, and then presents the outcomes; and finally, in Section V, a summary of and conclusion to this study are provided.

II. DEFINITIONS OF TERMS

A. k-NN Join

Given two data sets R and S in the n-dimensional metric space D, any $r \in R$ and $s \in S$ are n-dimensional points formed by 0, 1. $r(i)(s(i))$ indicates the i^{th} point in $R(S)$, and $r_i(s_i)$ represents the i^{th} dimension in the $r(s)$. $d(r,s)$ shows the distance between the r and s. Jaccard distance is adopted to calculate the distance in this study.

DEFINITION 1:

$$d(r,s) = 1 - \frac{|r \cap s|}{|r \cup s|},$$

in the above equation, $|r \cap s|$ indicates the number of values as 1 in the intersection between r and s. $|r \cup s|$ shows the number of values as 1 in the union between r and s.

DEFINITION 2: *k*-nearest neighbors (*k*-NN): given an integer K, datum r, and a data set S, $knn(r, S)$ stands for K number of nearest distance $d(r, s)$.

$$knn(r,S) = \{(r, s[i])_k \mid \forall s[j] \in S, d(r,s[i]) < d(r,s[j])\}.$$

Definition 3: *k*-Nearest neighbor join (*k*-NN Join): given an integer K and two data sets R and S, K number of nearest neighbors are targeted for each $r \in R$.

$$knnJoin(R,S) = \{(r,s) \mid \forall r \in R, \forall s \in knn(r,S)\}.$$

B. Locality-Sensitive Hashing

LSH is able to map similar data onto the same bucket with higher probability than other hashing algorithms. Therefore, LSH can solve the problem of spotting the approximate nearest neighbors in the high-dimensional metric space. Afterwards, the same hash function is used to map new datum onto a certain bucket. The distance between the new mapped datum and each of all data in the bucket is calculated.

LSH can be defined as follows:

$$\begin{aligned} &if\ d(\vec{i}, \vec{j}) \le d_1\ then\ P_H(h(\vec{i}) = h(\vec{j})) \ge Pr_1 \\ &if\ d(\vec{i}, \vec{j}) > d_2\ then\ P_H(h(\vec{i}) = h(\vec{j})) \le Pr_2 \end{aligned}.$$

This is known as (d_1, d_2, Pr_1, Pr_2)-sensitive, where $\vec{i}$ and $\vec{j}$ are high-dimensional vectors composed of 0 and 1 in D; $d(\vec{i}, \vec{j})$ denotes the distance between $\vec{i}$ and $\vec{j}$ measured by the Jaccard distance; H denotes a hash map function from one vector to another, $H = \{h: s \rightarrow u\}$; Pr describes the probability of mapping data onto the same bucket.

III. LSHS *k*-NN JOIN ALGORITHM

This section introduces the approximate *k*-NN Join algorithm on the basis of LSH and conducted in Spark, named "LSHS *k*-NN Join". The major procedures are presented as follows:

(1) The high-dimensional raw data set S is normalized into the high-dimensional matrix formed by 0 and 1 represented as S_{ij}.

(2) A MinHash method is adopted to map S_{ij} thus forming the signature matrix.

(3) By using the aforementioned hash function, similar vectors in the signature matrix are mapped onto the same bucket with higher probability.

(4) By adopting the hash function, each datum $r \in R$ is mapped onto a certain bucket. The distance between the new mapped datum and each of the data in the bucket is calculated. In this way, *k*-nearest neighbors are obtained.

The above process was conducted by Spark-Resilient Dis-

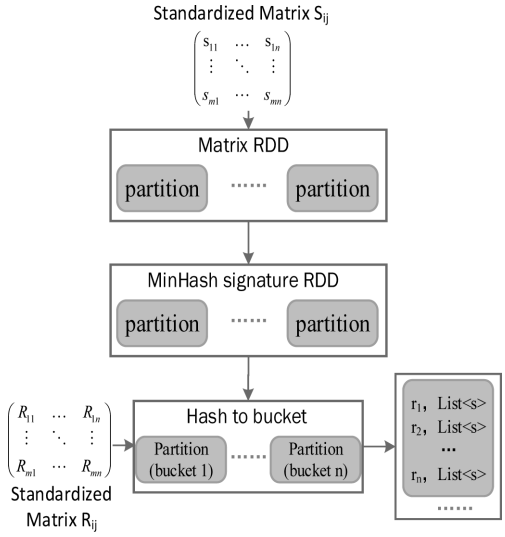

Fig. 1. LSHS *k*-NN Join algorithm.

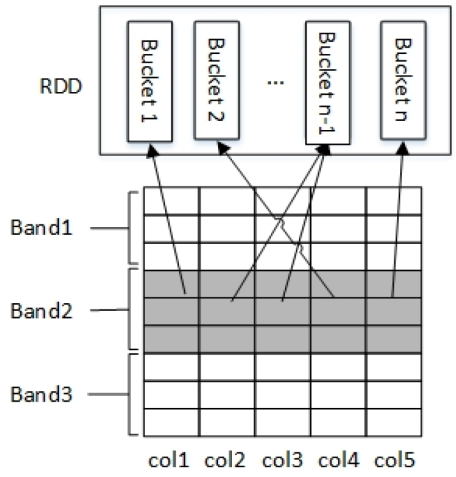

Fig. 2. Mapping the signature matrix onto the bucket.

tributed Datasets (RDD) as is shown in Fig. 1. The actual algorithm and its implementation will be discussed in the following section.

Algorithm 1: MinHash Signature

```
1: s ∈ ℝ^{n×d}  represent signature matrix
2: for i = 0:n
3:     for j = 0:d
4:         if s[i][j] ==0 continue;
5:         else
6:             for c = 0:n
7:                 sig[c][j] = min(h[i],sig[c][j])
8: Repeat step 2 to step 7 n times
```

The amount of calculation is still big because the normalized vectors are high-dimensional ones. MinHash signature can reduce the number of dimensions for raw data while maintaining their features as much as possible. This approach ensures that if the similarity in raw data is high, then the similarity of the data after signature remains high, and vice versa. A random function $h(x)=(ax+b)/d$ is adopted to implement the MinHash signature. In each iteration a and b are generated, x is the row indexed value, and d stands for data dimensions. Algorithm 1 describes the pseudo-code in the MinHash signature process.

A. Mapping of Signature Matrix onto the Bucket

Although, the signature matrix can reduce the number of dimensions of raw data, the time cost incurred in the detection of nearest neighbors in such a huge signature matrix is large. Therefore, similar data in the signature matrix is mapped onto the same bucket. Thus, we only need to search the nearest neighbors in the same bucket; searching the nearest neighbors in the whole signature matrix is not necessary. If the number of buckets is n, the calculation only considers $1/n$ when the data are distributed evenly in the bucket. The actual algorithm can be presented as follows:

(1) The signature matrix is parted into a number of bands, each of which is composed of rows. The number of rows can be determined by users.

(2) The algorithm selects any band and maps the column (using MD5 or SHA) in the selected band onto a bucket. In this case, data in the same column in the band will be mapped onto the same bucket. Thus, columns where the same bands are located have a high probability of similarity.

As shown in Fig. 2, two columns in Band 2 are mapped onto the same bucket. Therefore, col 2 and col 3 have a high probability of being largely similar.

If the signature matrix is assumed to be composed of n rows and m columns, and each band include rows, the number of bands is n/r. If the similarity between col 1 and col 2 is s, the probability of col 1 and col 2 in the same bands is s^r and the probability of col 1 and col 2 in different bands is $(1 - s^r)^{n/r}$. Thus, similar bands must exist in col 1 and col 2 and the probability is $1 - (1 - s^r)^{n/r}$, for example:

(a) When $r = 5$, $n = 100$, if the similarity between col 1 and col 2 is 0.9 ($S = 0.9$), similar bands must exist in col 1 and col 2 and the probability is $1 - (1 - 0.9^5)^{100/5} = 0.999999982$, which means if raw data are similar, the probability of those data mapped onto the same bucket is 0.999999982.

(b) When $r = 5$, and $n = 100$, if the similarity between col 1 and col 2 is 0.3 ($S = 0.3$), the probability of same bands existing in col 1 and col 2 is $1 - (1 - 0.3^5)^{100/5} = 0.0474$, which also implies that if raw data are dissimilar, the probability of those data being mapped onto the same bucket is only 0.0474.

B. k-NN Join

The dataset R is read from file as Spark RDD by mapping each $r \in R$ onto a bucket and computing the distance between the data in the bucket and itself, then returning k-nearest neighbors. Algorithm 2 gives the pseudo-code.

Algorithm 2: *k*-NN Join

```
val R = readFile(R_dataset_path)
R.map{
    Using a hash function to get the No. of buckets,
    The distance is calculated with above bucket data
    return(r_id, list of k-nearest data)
}
```

IV. EXPERIMENT AND EVALUATION

A. Hardware and Software Configuration

The experiment is carried out on VMWare, which is a well-known virtual machine. After installing VMWare on a physical (real) machine, 12 virtual machines are installed using VMWare. One virtual machine is employed as the master node and others as slave nodes. The configurations of each virtual machine are the same. The detailed configuration is presented in Tables 1 and 2.

CentOS-7-x86_64 is employed as an OS. Spark 2.2, Hadoop 2.8.1, and Java1.8 64-bit make up the software environment. The number of virtual machines is n ($n \leq 12$), each of which has four cores. Therefore, the best partition number is $4n$.

B. Dataset

Three real datasets are used:

(1) CNAE-9 Data Set (http://archive.ics.uci.edu/ml/datasets/CNAE-9): The dataset contains documents of free text business descriptions of Brazilian companies, divided into 9 categories; it has 857 dimensions.

(2) Farm Ads Data Set (http://archive.ics.uci.edu/ml/datasets/Farm+Ads): The data in this dataset was collected from text ads found on 12 websites that concern various farm-animal-related topics. The binary labels reflect whether or not the content owner approves of the ad. The dataset has 54877 dimensions and 4143 data.

(3) "YouTube Multiview Video Games Dataset" Data Set (http://archive.ics.uci.edu/ml/datasets/YouTube+Multiview+Video+Games+Dataset): This dataset contains approximately 120k instances, each described by 13 feature types, with 1000k dimensions.

We adopt Java as programming language in which to generate and process random datasets for this study.

Table 1. The configuration of the physical machine

CPU	Four processor and 4 cores per processor Intel Xeon E5-2430 2.20GHz. 4
Memory	128 GB
Disk	1 TB
Network	Gigabit Ethernet

Table 2. The configuration of virtual machine

CPU	One processor with four cores
Memory	8 GB
Disk	50 GB

C. Evaluation and Analysis

In order to verify the performance of our algorithm, two baseline methods are adopted. One is the exact k-NN Join implementation, which is selected because its accuracy is higher compared with the approximate k-NN Join implementation under the same conditions. The other is H-zkNNJ, which is proposed in literature [13]; it is an approximate solution that leverages space-filling curves and is implemented using Hadoop.

1) Precision

The number of buckets is the main factor that affects the precision of the algorithm. Precision is highest when the number of buckets is 1, as this is similar to mapping all the data onto the same bucket.

However, this case incurs maximum calculation, as it is equivalent to calculating all the data. On the contrary, the greater number of buckets is, the less time will be needed, but at the cost of reducing precision. We randomly extract 90% of the data from the dataset as training data, and the rest are used as test data. Here, p denotes accuracy, T denotes a collection of real labels, and Pr denotes a collection of prediction labels computed by the k-NN Join.

We define: $p = \frac{T \cap \Pr}{|T|} \times 100\%$.

Fig. 3(a) displays the results for the three aforementioned datasets, illustrating that the precision is the highest when the number of buckets is 1 regardless of the dataset. As the number of bucket increases, the precision declines overall, although still at a relatively high level.

It can be observed from Fig. 3(b) that the precision for an exact k-NN Join remains constant when the number of buckets is 1. This is because in this case the algorithm becomes equivalent to mapping all the data onto one bucket. As the

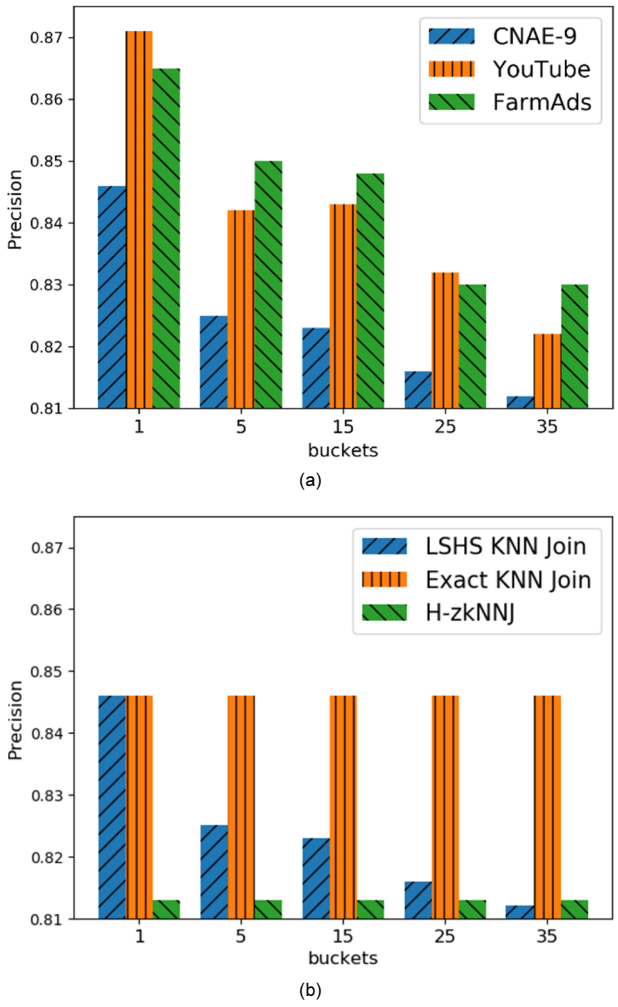

Fig. 3. Precision performance. (a) Precision for different datasets. (b) Compare with baseline.

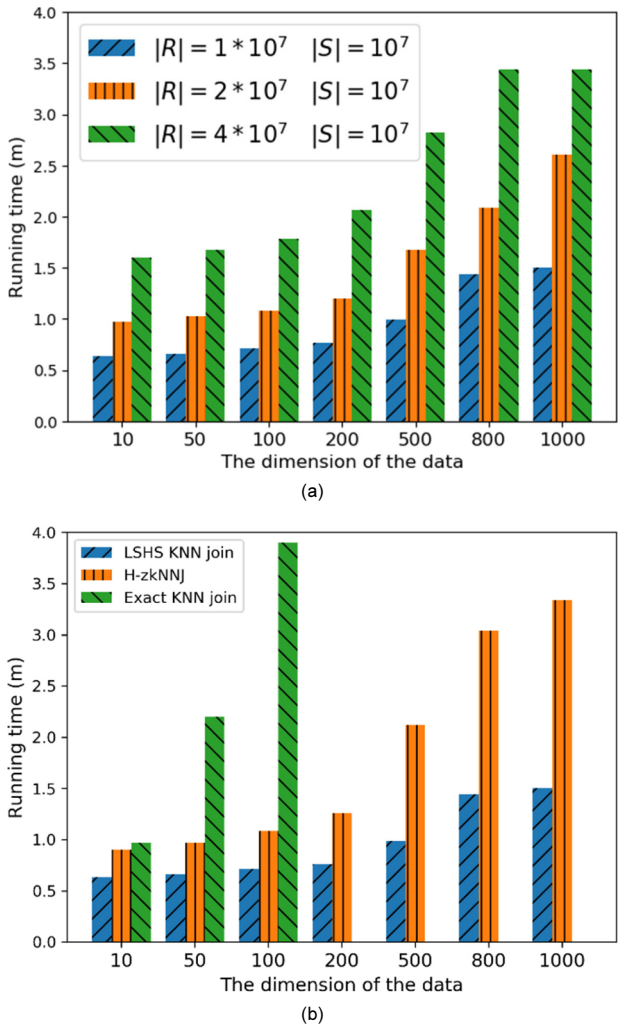

Fig. 4. Effect of data dimension. (a) Effect of data dimension. (b) Comparison with baseline.

number of buckets increase, the precision is reduced but still remains within an acceptable range. The accuracy of H-z*k*NNJ is always lower than our proposed algorithm when the number of buckets is less than 33. Therefore, by selecting a suitable number of buckets, we can obtain a precision higher than H-z*k*NNJ and very close to an exact *k*-NN Join.

2) Effect of Data Dimension

The computational complexity increases significantly with an increase in the data dimensions. However, the calculation of distance is performed by mapping the data onto the same bucket, significantly lowering the effect of data-dimension on the calculation. In the experiments in this study, we use 10 nodes, set k = 3, randomly generate dataset S and R, fix the size of S at 10^7, and vary R as 10^7, 2×10^7, and 4×10^7; the dimension of data is {10, 50, 100, 200, 500, 800, 1000}. As shown in Fig. 4(a), when the dimension increases from 10 to 1000 (i.e., by a factor of 100), the calculation time only increases by 2.512 times, 2.846 times, and 2.146 times for R = 10^7, 2×10^7, and 4×10^7, respectively. As is evident, the dimension has little effect on the computation time. As a result, the algorithm is suitable for high-dimensional data.

In order to compare our proposed approach with baseline approaches, we run these algorithms using the same set of conditions (nodes = 10, k = 3, $|S|$ = 10^7, $|R|$ = 10^7). From Fig. 4(b), we can observe that the proposed approach takes less time than the others. As the data dimensions increase, the running time of exact *k*-NN Join increases sharply (we can't draw an exact kNN join column when data dimensions are greater than 100); therefore, it is not suitable for high-dimensional data. The running time of H-z*k*NNJ increases faster than LSHS *k*-NN Join; thus, LSHS *k*-NN Join can handle high-dimensional data better than H-z*k*NNJ.

3) Effect of Node Number and Data Size

With an increase in the data quantity, the calculation will

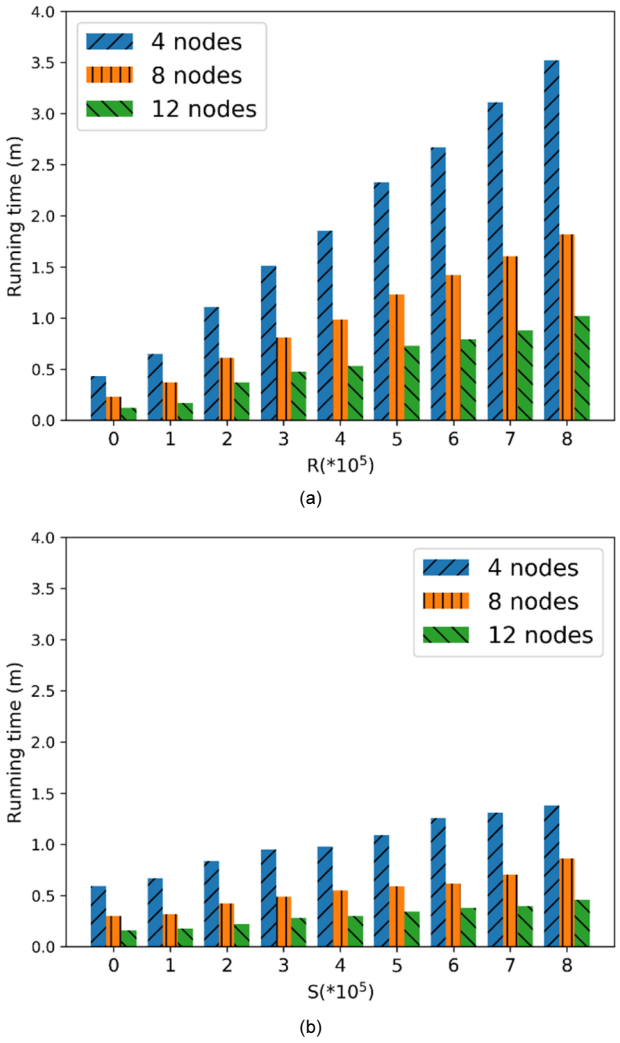

Fig. 5. Effect of node number and data size. (a) The size of *S* is fixed, and the size of *R* is changed. (b) The size of R is fixed, and the size of *S* is changed.

take longer; on the contrary, a greater number of nodes results in a lower calculation time.

For the purpose of studying the impact of data size and number of nodes on the time of computation, the following experiments are conducted: one dataset is kept unchanged and the size of another dataset is increased from 0.5×10^5 to 8×10^5 with a data dimension of 200 and $k = 3$. The number of nodes increases from 2 to 8.

Fig. 5(a) gives the results: (1) as the size of dataset *R* is increased by a factor of *m*, the computation time is also increased by a factor of *m*. (2) The time of calculation decreases with the increase in the number of nodes. If the number of nodes increases by a factor of 2, computation time is always reduced by a factor less than 2 since communication and scheduling between nodes costs a finite minimum time.

In Fig. 5(b), if the size of dataset S increases by a factor of *m*, the calculation time increases by a factor much less than *m*. The reason is why we have indexed the dataset *S*, and the data is mapped onto different buckets. Therefore, the time of computation in actual operation is greatly reduced; this is the reason our algorithm can handle large data with notably high performance.

V. CONCLUSION

In this paper, we introduce the LSHS *k*-NN Join algorithm for handling high-dimensional big data. In order to address standard issues in distributed systems, such as scalability and fault tolerance, we implement our algorithm on Spark. The key idea of the algorithm is to use LSH to map the data. The theoretical analysis and results show that the algorithm proposed in this study for processing high-dimensional big data is fast and very effective.

ACKNOWLEDGMENTS

This paper was supported by Wonkwang University in 2017.

REFERENCES

[1] Y. Hu, C. Yang, C. Ji, Y. Xu, and X. Li, "Efficient snapshot *k*NN join processing for large data using MapReduce," in *Proceedings of 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, Wuhan, China pp. 713-720, 2016. DOI: 10.1109/ICPADS.2016.0098.

[2] J. Maillo, J. Luengo, S. Garcia, F. Herrera, and I. Triguero, "Exact fuzzy k-nearest neighbor classification for big datasets," in *Proceedings of 2017 IEEE International Conference on Fuzzy Systems*, Naples, Italy, pp. 1-6, 2017. DOI: 10.1109/FUZZ-IEEE.2017.8015686.

[3] T. Wen, Z. Zhang, M. Qiu, M. Zeng, and W. Luo, "A two-dimensional matrix image based feature extraction method for classification of sEMG: a comparative analysis based on SVM, kNN and RBF-NN," *Journal of X-ray Science and Technology*, vol. 25, no. 2, pp. 287-300, 2017. DOI: 10.3233/XST-17260.

[4] M. Antol and V. Dohnal, "Popularity-based ranking for fast approximate kNN search," *Informatica*, vol. 28, no. 1, pp. 1-21, 2017. DOI: 10.15388/informatica.2017.118.

[5] T. Emrich, H. P. Kriegel, P. Kroger, J. Niedermayer, M. Renz, and A. Zufle, "On reverse-k-nearest-neighbor joins," *GeoInformatica*, vol. 19, no. 2, pp. 299-330, 2015. DOI: 10.1007/s10707-014-0215-5.

[6] M. Afzali, N. Singh, and S. Kumar, "Hadoop-MapReduce: a platform for mining large datasets," in *Proceedings of 2016 3rd International Conference on Computing for Sustainable Global Development*, New Delhi, India, pp. 1856-1860, 2016.

[7] H. V. L. Cao, T. N. Phan, M. Q. Tran, T. L. Hong, and M. N. Q. Truong, "Processing all k-nearest neighbor query on large multidimensional data," in *Proceedings of 2016 International Conference on Advanced Computing and Applications*, Can Tho,

Vietnam, pp. 11-17, 2016. DOI: 10.1109/ACOMP.2016.012.

[8] G. Song, J. Rochas, L. El Beze, F. Huet, and F. Magoules, "k-Nearest neighbour joins for big data on map reduce: a theoretical and experimental analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2376-2392, 2016. DOI: 10.1109/TKDE.2016.2562627.

[9] J. D. Kim, "A method for continuous k-nearest neighbor search with partial order," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 15, no. 1, pp. 126-132, 2011. DOI: 10.6109/jkiice.2011.15.1.126.

[10] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, et al., "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56-65, 2016. DOI: 10.1145/2934664.

[11] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, et al., "Mllib: machine learning in Apache Spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235-1241, 2016.

[12] Y. Zhong and X. Peng, "SIFT-based low-quality fingerprint LSH retrieval and recognition method," *International Journal of Signal Processing, Image Processing and Pattern Recognition*, vol. 8, no. 8, pp. 263-272, 2015. DOI: 10.14257/IJSIP.2015.8.8.28.

[13] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *Proceedings of the 15th International Conference on Extending Database Technology*, Berlin, Germany, pp. 38-49, 2012. DOI: 10.1145/2247596.2247602.

Jiaqi Ji

received his B.S. and M.S. degrees from the Department of Computer Science, Nanchang University, Nanchang, China, in 2007 and 2010, respectively. From 2010 to 2012, he worked as a software engineer in Beijing, China. From 2012 to 2016, he worked as a lecturer at the Department of Information Center, Hebei Normal University for Nationalities, Chengde, China. He is currently a Ph.D. student at Wonkwang University, Iksan, Korea. His current research interests include big data processing.

Yeongjee Chung

received his Ph.D. degree from the Yonsei University, Seoul, Korea in 1993. He has been a Professor in the Department of Computer Engineering at Wonkwang University, Iksan, Korea from 1995. His primary research interests include the Internet of Things, big data processing, and mobile computing platforms. He is working on creating new knowledge via various applications including ubiquitous healthcare and mobile computing with big data processing.