kNN-DP: Handling Data Skewness in kNN Joins Using MapReduce

Xujun Zhao, jifu zhang, *Member, IEEE*, and Xiao Qin, *Senior Member, IEEE*

Abstract—In this study, we discover that the data skewness problem imposes adverse impacts on MapReduce-based parallel *kNN-join* operations running clusters. We propose a data partitioning approach—called *kNN-DP*—to alleviate load imbalance incurred by data skewness. The overarching goal of *kNN-DP* is to equally divide data objects into a large number of partitions, which are processed by mappers and reducers in parallel. At the heart of *kNN-DP* is a data partitioning module, which dynamically and judiciously partitions data to optimize *kNN-join* performance by suppressing data skewness on Hadoop clusters. Data partitioning decisions largely depends on data properties (e.g., distributions), the analysis of which is highly expensive for a massive amount of data. To speed up the data-property analysis, we incorporate a sampling technique to profile the data distribution of a small sample dataset representing big datasets. After building a data-partitioning cost model for parallel *kNN-join*s, we derive the time-complexity upper and lower bounds of parallel *kNN-join* algorithms. The cost model offers us a guidance to systematically investigate *kNN-DP*'s performance. *kNN-DP* obtains global nearest neighbors using local nearest neighbors. To improve the accuracy of such an approximation solution, we augment each node's local data by a small amount of redundant data. We develop two *kNN-DP*-based schemes called *LSH+* and *z-value+*, which seamlessly integrate *kNN-DP* with the existing *LSH* and *z-value* algorithms for *kNN-join* computing. We implement and evaluate *LSH+* and *z-value+* on a 24-node Hadoop cluster driven by both synthetic and real-world high-dimensional datasets. The experimental results show that *kNN-DP* significantly improves the performance of *LSH* and *z-value* while offering high extensibility and scalability on Hadoop clusters.

Index Terms—Data skewness, *kNN-join*s, data partitioning, MapReduce

1 Introduction

The *k-nearest-neighbor join* (i.e., *kNN-join*) is a primitive operation widely adopted by a wide range of data-mining applications like *k-means* clustering and outlier detection. Combining the *kNN* query and the join operation, *kNN-join* becomes a very expensive operation. To mitigate the high overhead of *kNN-join*, a handful of prior studies have put forward a series of parallel *kNN-join* methods using MapReduce. We discover that a common limitation of the existing MapReduce-based *kNN-join* solutions lies in data skew problems, which lead to imbalanced workload among MapReduce tasks. In this study, we propose a holistic approach to tackling the data skew problem by partitioning data among nodes of a Hadoop cluster. We show that handling data skew can substantially improve the performance MapReduce-based *kNN-join* operations. In particular, we seamlessly integrate our data partitioning scheme with the *locality-sensitive-hashing*-based (i.e., *LSH* [1]) and *space-filling-curves*-based (i.e., *z-value* [2]) *kNN-join* algorithms. We refer to our two *kNN-DP*-based schemes as *LSH+* and *z-value+* throughout this paper.

- *X. Zhao and J. Zhang are with Taiyuan University of Science and Technology (TYUST), Taiyuan, Shanxi 030024, China. E-mail: zxj0226@126.com, jifuzh@sina.com.*
- *X. Qin is with Department of Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, Auburn, AL 36849-5347. E-mail: xqin@auburn.edu.*

Manuscript received 23 Jan. 2017; revised 15 June 2017; accepted 17 Oct. 2017. Date of publication 30 Oct. 2017; date of current version 9 Feb. 2018. (Corresponding author: Jifu Zhang.)
Recommended for acceptance by T. El-Ghazawi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2767596

1.1 Observations

We are primarily motivated by the following three observations to address the data skew problem in MapReduce-based *kNN-joins*.

Observation 1. A *kNN-join* operation combines each object of one dataset with its *kNN*s in another dataset, providing more meaningful query results than range joins (a.k.a., range similarity joins). *kNN-joins* are expensive operations, because both the nearest neighbor search and the join operations are time consuming. The high overhead of *kNN-join* becomes more pronounced when it comes to large datasets with multi-dimensions. In the past decade, optimization algorithms were proposed to improve *kNN-join* performance [3], [4]. Popular optimization ideas that help in reducing I/O and CPU costs include join scheduling, data sorting, as well as filtering and reduction [5]. These simple and yet efficient techniques improve efficiency of processing high-dimensional data.

Observation 2. An increasing number of parallel *kNN-join* algorithms are developed to deal with the fast growing input datasets with multi-dimensions (see, for example, [6]). A majority of traditional parallel *kNN-join* algorithms consist of three phases, namely, task creation, task assignment, and parallel task execution [7]. Preserving data locality is an efficient way of reducing both CPU and I/O cost.

1045-9219 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

Apart from conventional parallel *kNN-join*s, MapReduce -based *kNN-join*s capture much attention in the past few years [2], [8], [9], [10]. MapReduce [11] is a simple yet efficient parallel computing framework offering high scalability and fault tolerance. Prior studies confirmed that MapReduce is an important framework of processing a massive amount of data with multi-dimensions. We are motivated to address performance issues in MapReduce-based *kNN-join*.

Observation 3. We elaborate an intriguing data-skewness problem in MapReduce-based *kNN-join*s. We observe that existing *kNN-join* algorithms using MapReduce are sensitive to data characteristics and data skewness. Skewed data inevitably slow down parallel *kNN-join*s, because data skewness leads to imbalanced workload making some nodes a performance bottleneck in MapReduce clusters. The data-skewness problem motivates us to propose a data partitioning scheme to achieve balanced load in clusters running MapReduce-based *kNN-join*s.

1.2 Contributions

The above three observations motivate us to design a holistic scheme called *kNN-DP* to partition data among a Hadoop cluster's datanodes to address the data skew issue. Data skewness inevitably leads to imbalanced load across Hadoop clusters, thereby significantly downgrading *kNN-join* performance (see also Observation 3). The data skewness becomes more pronounced in *kNN-join*s when (1) input datasets R and S follow very different distributions and (2) set R is substantially larger than set S. Such data skewness is alleviated by *kNN-DP*'s data partitioning mechanism, which applies the dynamic partition-boundary adjusting method to judiciously make all the nodes equally process *kNN-join*s in parallel.

The load balancing idea of *kNN-DP* is very different from traditional load-balancing schemes that split tasks into smaller ones and reassign some of the tasks to idle processors. Rather than assigning tasks, *kNN-DP* aims to balancing load through data-placement decisions making each node evenly handle *kNN-join* operations. We conduct experiments (see Section 5) to demonstrate that by handling data skewness, *kNN-DP* improves the performance of MapReduce-based *kNN-join* operations.

Data partitioning decisions largely depend on data properties (e.g., distributions). It is highly expensive to obtain data properties of a massive amount of data. To capture data properties in a short time period, we propose a sampling technique to profile the data distribution of a sample dataset, which is a tiny portion of a big dataset.

We build a data-partitioning cost model in the context of parallel *kNN-join* algorithms (see Section 3.2). With this cost model in place, we derive the time-complexity upper and lower bounds of parallel *kNN-join*s. We make use of the cost model to study the impact of data partitions on the processing time of *kNN-join*s. Such an analysis allows us to systematically evaluate the performance of data- partitioning decisions offered by *kNN-DP*.

Like prior parallel *kNN-join*s, our *kNN-DP* obtains global nearest neighbors using local nearest neighbors. Such an approximation solution may be unable to discover all the global nearest neighbors, lowering *kNN-join* accuracy. In order to improve the *kNN-join* accuracy, we augment the local data of each node by a small amount of redundant data, which is appended to the head and tail of each data block (see Section 4.5). We also quantitatively evaluate the impact of the *kNN-DP*'s redundant-data strategy on the *kNN-join* accuracy.

One salient feature of our data partitioning scheme is that it is orthogonal to a wide range of MapReduce-baesd *kNN-join* algorithms. This feature allows us to readily and seamlessly integrate our *kNN-DP* with existing parallel *kNN-join* algorithms such as *LSH*-based [1] and *z-value*-based [2] *kNN-join*s. The new *LSH+* and *z-value+* solutions are articulated in Section 4.3.

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 describes the background knowledge on *kNN-join* operations and MapReduce programming. Section 3 analyses data skewness in parallel *kNN-join*. Section 4 presents an approach (i.e., *kNN-DP*) that partitions data to optimize MapReduce-based *kNN-join*s, and discusses an application of our data-partitioning scheme in parallel *kNN-join*s. Section 5 describes experimental setup and results. Section 6 summarizes the prior work related to *kNN-DP*. Finally, Section 7 concludes this paper.

2 Preliminaries

In this section, we first introduce *kNN-join* method, followed by an introduction of the MapReduce framework.

2.1 kNN Join

Let us consider two datasets R and S in space $\mathbb{R}^d$, where each object (e.g., $r \in R$ and $s \in S$) is represented as a d-dimensional object. We measure the similarity distance between objects $r \in R$ and $s \in S$ using their euclidean distance $d(r, s)$. Please note that the other ways of quantifying similarity distance can be found in Section 4.5. Operation $knn(r, S)$ returns a set of k nearest neighbors or *kNN* of point r from set S.

Given object $r \in R$, the *kNN-join* operation $knnJ(R, S)$ of datasets R and S returns a combination set of each object r's *kNN* set. Thus, we express $knnJ(R, S)$ using *kNN-join* operation $knn(r, S)$ as follows.

$$knnJ(R, S) = \{(r, knn(r, S))| \text{ for all } r \in R\}. \quad (1)$$

2.2 MapReduce Framework

MapReduce is a parallel programming model proposed by Google [11]. The goal of MapReduce is to simplify the processing of large datasets on inexpensive cluster computers. A MapReduce program typically consists of a pair of user-defined map and reduce functions.

Hadoop is an open source software implementing the MapReduce computing framework. Data in Hadoop are stored in a Hadoop distributed file system, which consists of multiple data nodes and a master node called namenode. The Hadoop runtime system establishes two processes—JobTracker and TaskTracker. The JobTracker splits a submitted job into map and reduce tasks, which are scheduled and assigned to all available TaskTrackers. The TaskTrackers

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

accept and process the assigned map/reduce tasks. After completing all mappers in a Hadoop program, the Hadoop runtime system groups all intermediate results and launches reducers to producing final results.

3 Data Skewness in Parallel *k*NN Joins

This section presents the data skewness problem of data partitioning strategies (see Section 3.1) in *kNN-join*s through a motivational example (see Section 3.3). We also investigate time complexity and optimization upper bound of data partitioning in Sections 3.2 and 3.4. A manual data-partitioning strategy is described in Section 3.5.

3.1 Three Data Partitioning Strategies

Given two datasets (i.e., R and S) and each object $r \in R$, *kNN-join* operation searches object r's k nearest neighbors from set S. Each object in R must scan set S once to compute the distance between each pair of objects from sets R and S. Thus, *kNN-join* operation's time complexity is $O(|R| \times |S|)$. If items in sets R and S are sorted, the time complexity is reduced to $O(|R| \times log_2|S|)$. In MapReduce-based parallel *kNN-join*s, we divide sets R and S into n groups (a.k.a., n partitions), where $R_i \subseteq R$, $S_i \subseteq S$, $1 \leq i \leq n$, $R_i \cap R_j = \varnothing$, $S_i \cap S_j = \varnothing$, $(i \neq j)$. When the ith partition (i.e., R_i and S_i) is sorted and each partition is concurrently processed by a datanode, the time complexity of parallel *kNN-join*s is the maximum time of a slow node that handles more data than the other nodes. Thus, the time complexity is $Max_{1\leq i\leq n}\{O(|R_i| \times log_2|S_i|)\}$. If sets R and S can be well divided, the running time of the worst partition will be reduced. Theoretically speaking, shortening processing time of the worst partition tends to speedup the performance of parallel *kNN-join*s.

There are three strategies to partition input sets R and S. In the first strategy is referred to as *Balance_R*, which equally divides set R into n groups. Thus, all the partitions in this strategy contain the same number of objects; S set is partitioned according to the boundaries of each partition of R. The second strategy called *Balance_S* makes a partition decision by dividing S into n groups, each of which manages the same number of objects. The set R is divided according to the boundaries of each S partition. The last strategy is coined as *Balance_R∪S*; object set $\{R \cup S\}$ is grouped into n partitions, each of which contains equal number of objects.

3.2 Time Complexity Analysis

Let us consider an ideal and perfect case, where input datasets are divided into n partitions. Each partition has a block $R_i \cup S_i$ satisfying $|R_i| = |R|/n$, $|S_i| = |S|/n$. In this ideal case, workload among datanodes of a Hadoop cluster is well balanced and; therefore, the time complexity can be expressed as $O((|R|/n) \times log_2(|S|/n))$. We refer to this ideal time complexity as the best-case time complexity of parallel *kNN-join*s or O_{best}. Thus, we have $O_{best} = O((|R|/n) \times log_2(|S|/n))$.

Now we derive the worst-case time complexity of the *Balance_R* and *Balance_S* strategies. The worst case of *Balance_R* occurs when all the objects in set S is divided into a single partition (i.e., there exists one block, say S_i, such that $|S_i| = |S|$). The time complexity of the worst case of *Balance_R* is dominated by the largest single partition. Thus, the worst-case complexity $O_{R,worst}$ of *Balance_R* can be written as

$$\begin{aligned} O_{R,worst} &= \underset{(1\leq i\leq n)}{Max}\{O(|R_i| \times log_2|S_i|)\} \\ &= O((|R|/n) \times log_2|S|). \end{aligned} \tag{2}$$

The worst case of *Balance_S* emerges when all the objects in R is grouped into a single large partition (i.e., there exists R_i such that $|R_i| = |R|$). Thus, the worst-case time complexity $O_{S,worst}$ of *Balance_S* depends on time spent in processing the largest partition. Thus, $O_{S,worst}$ is expressed as

$$\begin{aligned} O_{S,worst} &= \underset{(1\leq i\leq n)}{Max}\{O(|R_i| \times log_2|S_i|)\} \\ &= O(|R| \times log_2(|S|/n)). \end{aligned} \tag{3}$$

If one partition's time complexity is lower than O_{best}, then another partition's computing complexity will be higher than O_{best}. Thus, the partitions experience unbalanced processing times, which inevitably lead to data skewness problems. We quantify the degree of data skewness in the ith partition using a ratio between its time complexity and O_{best}. We refer to this ratio as *data-skewness-degree*, which can be expressed as follows. Please note that the *data-skewness-degree* is used in Section 4.3.

$$\delta_i = \left|\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}}\right|. \tag{4}$$

If the ith *data-skewness-degree* equals to 0 (i.e., $\delta_i = 0$), it indicates that data is balanced with respect to the ith partition. Thus, when *degree* δ_i approaches to 0, data skewness is diminishing; otherwise, when δ_i is far away from 0, the skewness becomes severe. To describe an approximate data balance in the ith partition, we introduce threshold T referred to as *data-skew-degree* threshold, which is expressed from *data-skewness-degree* as (see also (4))

$$\delta_i = \left|\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}}\right| \leq T. \tag{5}$$

We derive (6) from (5) as

$$-T \leq \frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}} \leq T, \tag{6}$$

Eq. (6) indicates that if *degree* δ_i is anywhere in a range between $-T$ and T, then the ith partition is balanced; otherwise, if δ_i is less than $-T$ (i.e., $\delta_i < -T$) or δ_i is larger than T (i.e., $\delta_i > T$), the ith partition is imbalanced.

We obtain the data-skewness level Δ of a cluster from *data-skewness-degree* defined in (4) as

$$\Delta = \underset{(1\leq i\leq n)}{Max}\{|\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}}|\}. \tag{7}$$

Eq. (7) is regarded as a data-skewness cost model, which offers a guideline to the design of MapReduce-based *kNN-join*s.

3.3 Motivational Examples

3.3.1 Steps of Illustrating an Example

We carry out the following six steps to illustrate a motivational example. First, we apply a random data generator to

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Comparison of the Theoretical Processing Time Complexities of the *Balance_R*, *Balance_S*, and *Balance_R∪S* Strategies as Well as the Improved *Balance_R+* Strategy

	Balance strategies			
Parameters	R	S	R ∪ S	R+
first partition	12,379	16,398	14,136	12,888
second partition	12,541	16,306	14,094	12,871
third partition	12,216	17,303	14,616	12,787
four partition	13,811	1,429	8,069	12,889
maximum	13,811	17,303	14,616	12,889
Best Case	12,859	12,859	12,859	12,859
Worst Case	15,359	51,438	×	×

create a small dataset of 10,000 objects with 200 attributes (a. k.a., dimensions) satisfying the normal distribution. Second, using a traditional similarity measure method (i.e., *locality-sensitive-hashing* scheme or *LSH*), we express each multi-dimensional data object as a one-dimensional value referred to as data property value or property value for short. Third, we organize all the values into two datasets, namely, set R and set S. Each of sets R and S contains 5000 property values (i.e., $|R| = 5000, |S| = 5000$). Fourth, we sort objects in sets R and S in the non-decreasing order of the property values. Fifth, we partition the sorted data in sets R and S into four groups according to the *Balance_R*, *Balance_S*, and *Balance_R∪S* strategies (see Section 3.1). Finally, we use the time complexity model (see Section 3.2) to intuitively show the time complexity of each strategy.

3.3.2 Results of the Examples

Table 1 summarized the theoretical processing time complexities of the *Balance_R*, *Balance_S*, and *Balance_R∪S* strategies along with an improved method *Balance_R+*. *Balance_R+* is a manual data partitioning strategy—an extension of *Balance_R* (see, Section 3.5).

We draw the following three intriguing observations from the processing-time comparison (see Table 1) of the four partitioning strategies.

The *Balance_R*, *Balance_S*, and *Balance_R∪S* data partitioning strategies lead to data skewness. For example, in the *Balance_R* case, the processing time complexities of the four partitions are 12,379, 12,541, 12,216, and 13,811, respectively. That is, the last partition -exhibiting the longest processing time- becomes a performance bottleneck in *kNN-join*s. Such a performance bottleneck can be alleviated by tackling the data skewness.

Table 1 shows that the processing time complexities of the *Balance_R*, *Balance_S*, and *Balance_R∪S* strategies are noticeable longer than the best-case time, which is the optimization upper bound of handling data skewness. When the processing time complexity of a data partitioning strategy is approaching to the best case, the adverse impact of data skewness is diminishing.

The worst-case processing time complexities of the *Balance_R* and *Balance_S* strategies are 15,359 and 51,438, respectively. The processing time complexity of an optimization scheme of *Balance_R* is in the range between 12,859 and 15,359. Similarly, the processing time complexity of an optimization scheme of *Balance_S* falls in the range between 12,859 and 51,438. The wide gap between the best-case and worst-case times motivates us to propose optimized data partitioning schemes.

TABLE 2
The Time Complexities of *kNN-join* on the Uniform Distribution Data Set

	Balance strategies			
Parameters	R	S	R ∪ S	R+
first partition	39,800	47,639	45,280	40,933
second partition	42,781	26,812	31,856	40,962
third partition	39,582	48,426	45,400	40,970
maximum	42,781	48,426	45,400	40,970
Best Case	40,955	40,955	40,955	40,955
Worst Case	46,238	122,877	×	×

In the second example, we test a dataset satisfying the uniform distribution. The number of partitions is set to three, the numbers of objects in $|R|$ and $|S|$ are 10,000 and 15,000, respectively. The time complexities of *kNN-join* are summarized in Table 2.

The results from Table 2 are similar to those plotted in Table 1. We conclude that there is a data skew issue in parallel *kNN-join* operations on datasets with various distributions, number of partitions, and data sizes.

3.4 Optimization Upper Bound

An optimization upper bound quantifies the extent to which a data-partitioning scheme may optimize the *kNN-join* performance. In other words, an optimization upper bound signifies the maximum improvement percentage achieved by a data partitioning decision. The optimization upper bound depends on the worst-case O_{worst} and best-case O_{best} time complexity of *kNN-join*s. It is worth noting that time O_{worst} is either $O_{R,worst}$ or $O_{S,worst}$, which ever is larger. In other words, O_{worst} can be written as $O_{worst} = max\{O_{R,worst}, O_{S,worst}\}$. We define the optimization upper bound as a ratio between time O_{worst} and O_{best}; thus, we express the upper bound as

$$U = (O_{worst} - O_{best})/O_{worst}. \tag{8}$$

From (2) and (8), we derive the upper bound of *Balance_R* as

$$U_R = \frac{O(\frac{|R|}{n} \times log_2|S|) - O(\frac{|R|}{n} \times log_2 \frac{|S|}{n})}{O(\frac{|R|}{n} \times log_2|S|)} = log_{|S|}n. \tag{9}$$

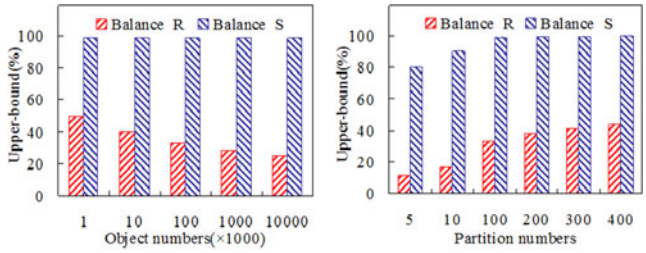
We observe from (9) that the optimization upper bound of *Balance_R* significantly decreases with an increasing value of $|S|$ when the number of partitions is fixed; the upper bound dramatically goes up with the increasing n value when $|S|$ is constant.

Similarly, we obtain the upper bound of *Balance_S* from (3) and (8) as

$$U_S = \frac{O(|R| \times log_2 \frac{|S|}{n}) - O(\frac{|R|}{n} \times log_2 \frac{|S|}{n})}{O(|R| \times log_2(|S|/n))} = \frac{n-1}{n}. \tag{10}$$

We conclude from (10) that the optimization upper bound of *Balance_S* is only dependent of the number n of partitions.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.



(a) Objects Number vs. Upper Bound. (b) Partitions Number vs. Upper Bound.

Fig. 1. The optimization upper bound in Balance_R and Balance_S schemes.

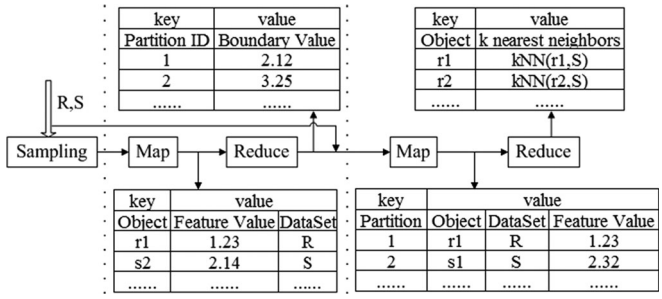


Fig. 2. An overview of the *kNN-DP* design.

The optimization upper bound depends on the numbers of data objects in dataset R and S as well as the number n of partitions. Fig. 1 exhibits a series of $optimization - upper - bound$ values of *Balance_R* and *Balance_S* schemes in various number of objects and the number n of partitions. In the example shown Fig. 1a, the numbers of objects in R and S vary from 10^3 to 10^7, and n is set to 100. We make two intriguing observations from Fig. 1a. First, in the *Balance_R* scheme, when the numbers of objects in R and S significantly increase, the optimization upper bound goes down. Second, the optimization upper bound of the *Balance_S* scheme is always maintained at a high level, because U_S is 0.99 when n is set to 100.

In Fig. 1b, we set the number n of partitions to a range between 5 to 400; we keep the numbers of objects in R and S as 10^6. Fig. 1b shows that increasing the number n of partitions leads to large upper bound U values.

The results plotted in Fig. 1 suggests that parallel *kNN-join*s running on a large-scale high-performance cluster is likely to encounter serious data skewness, which is observed in the aforementioned motivational examples illustrated in Section 3.3.2. In order to address the data-skewness issue, we design a data partitioning method called *Balance_R+* - an advanced version of the *Balance_R* scheme. The detailed description on *Balance_R+* can be found in Section 3.5.

In Table 1, when the number of partitions is four, the optimization upper bounds from the perspective of *Balance_R* and *Balance_S* are 16.3, 75.0 percent, respectively. Interestingly, when the number of partitions is increased to 400, these two optimization upper bounds significantly go up to 43.32, 97.5 percent, respectively. We conclude that optimization upper bounds are moving to the next high level when the number of partitions scales up. This trend suggest that for a large-scale Hadoop cluster processing *kNN-join* operations on big data, making good data partitioning decisions can significantly optimize *kNN-join* performance.

3.5 The *Balance_R+* Strategy

Comparing the processing time complexities of the *Balance_R*, *Balance_S*, and *Balance_R∪S* strategies (i.e., 13,811, 17,303, and 14,616) from Table 1, we conclude that *Balance_R* makes the best data partitioning decisions. Nevertheless, *Balance_R*'s time complexity is still much longer than that (i.e., 12,859) of the best case. We manually adjust the size of each partition in a way to shorten the long processing time complexity. We refer to this manual adjustment solution as the *Balance_R+* scheme.

In *Balance_R+*, we manually change the size of each partition by adjusting group boundaries. During the course of each adjustment, we focus on cutting the fat of the largest partition. Specifically, we modify the largest partition's lower and upper boundaries in order to move objects into the two small adjacent partitions. After each adjustment, we obtain the processing time complexities of the four updated partitions. Then, we compare the longest process time complexity with the best-case complexity to determine if there is a need for further adjustment. If there is a big discrepancy between the processing time complexity of the largest partition and the best-case time, we follow the above steps to repeatedly adjust the four partitions. The manual adjustment procedure is stopped when the gap between the longest processing time complexity and the best-case complexity is insignificant.

After the manual adjustment is accomplished, the processing time complexities of the four partitions are 12,888, 12,871, 12,787, and 12,889, respectively (see the last column in Table 1).

The data skewness experienced by the four nodes can be alleviated by manually adjusting the partition boundaries. Unfortunately, the *Balance_R+* strategy delineated in this motivational example suffers a few drawbacks. First of all, it is very time consuming to manually change partition boundaries. For example, we have to estimate the processing time complexity of each partition after one round of adjustment is done. Second, although it may be possible to manually adjust a small number of partitions (e.g., four partitions), configuring the boundaries of a large number of partitions becomes a daunting task. Even worse, it is an infeasible task to manual partition a massive amount of data. This challenging problem motivates us to design a data partitioning strategy, which can dynamically and judiciously partition data to minimize data skewness in parallel *kNN-join*s.

4 Data Partitioning in Parallel *k*NN Joins

In this section, we present the development of *kNN-DP*, the data partitioning scheme that optimize *kNN-join*s running in the MapReduce programming framework. After offering *kNN-DP*'s overview (see Section 4.1), we discuss the data sampling techniques implemented in the data pre-processing procedure (see Section 4.2). Next, we describe *kNN-DP*'s first MapReduce job, which divides data samples into n partitions followed by adjusting data in unbalanced partitions (see Sections 4.3 and 4.4). Finally, we provide an description on the second MapReduce job that partitions data in a way to balance load among reducers (see Section 4.5).

4.1 An Overview

Fig. 2 illustrates the working flow of *kNN-DP* consisting of a pre-processing procedure, followed by two MapReduce

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

jobs. We outline the functionalities of these three components as follows.

- *Profiling Sample Data.* The pre-processing procedure studies samples from input datasets R and S. The profiling information on the samples captures data distribution properties of the input datasets. The sampling technique implemented in the pre-processing procedure improves the accuracy of data partitioning at low cost.
- *Obtaining Data-Partition Boundaries.* The first MapReduce job determines boundaries of data partitions. This job aims to ensure that each partition's processing time complexity is approximately equal to the best-case time complexity, which implies that all the partitions are well balanced. The algorithm of the first MapReduce job is designed using the *dynamic partition-boundary adjusting* method.
- *Partitioning Data and Computing* kNN-joins. The second MapReduce job achieves two goals. First, mappers in this job partition input data across datanodes according to the boundaries determined by the first MapReduce job. Second, reducers are positioned to search k nearest neighbors in dataset S for $\forall r \in R$.

4.2 Profiling Sample Data

An ideal data partitioning strategy to optimize *kNN-join*s should cluster objects based on their similarity, aiming to create multiple partitions with equal load. When equal-sized partitions are distributed to multiple nodes, each of which handle one partition, the processing time of nodes are close to each other. In other words, making multiple partitions share with a similar processing time complexity is an efficient way of optimizing parallel *kNN-join* algorithms.

To equally partition a large input data, one has to study the data's distribution property. The overhead of profiling data property is high, because it requires sorting and scanning the massive amount of data. To reduce the expensive profiling process, we rely on a small sample dataset to resemble the large data's property.

We implement a pre-process procedure carried out in the master node of a Hadoop cluster to obtain data distribution properties of big data from small samples. Although there exist various sampling methods, none of these sampling schemes can be widely applied to treat all data types. Therefore, it is arguably true that a practical way is to employ an appropriate sampling method according to data characteristics. In order to obtain data partition boundaries at low computing cost, we design the following three data sampling schemes running on Hadoop clusters.

- *Simple Random Sampling.* We generate a small sample set from a large input data distribution stored on *HDFS*, if datasets R and S follows a same distribution. Objects are uniformly and randomly selected with probability $p = \frac{1}{\varepsilon^2 \times N}$, $\varepsilon \in (0, 1)$, where ε controls the sample size, and N is the number of objects in dataset $R \cup S$.
- *Heterogeneous Random Sampling.* Heterogeneous data distribution refers to cases where data distributions of datasets R and S are different. We propose heterogeneous random sampling to separately perform sampling on R and S. Thus, R and S have two different small sample sets using the above simple random sampling method. Sample datasets R' and S' are uniformly and randomly chosen from R and S, respectively. Objects are sampled with a probability of $p = \frac{1}{\varepsilon^2 \times N}$, $\varepsilon \in (0, 1)$, where ε controls sample size, N is the number of objects in dataset R or S.
- *Interval Sampling.* When the distribution of datasets R and S is unknown a priori, we apply interval sampling to profile input data. Sample datasets R' and S' are extracted by separating the same number of objects (i.e., $\varepsilon^2 \times N$) from R and S, respectively. Here N is the objects number of dataset R or S; ε is in the range between 0 and 1 (i.e., $\varepsilon \in (0, 1)$).

4.3 Obtaining Data-Partition Boundaries

The first MapReduce job in *kNN-DP* attempts to divide sample dataset to n equal groups by dynamically adjusting partition-boundaries in unbalance groups. The pseudocode of this MapReduce job is detailed in Algorithm 1, which performs data partitioning coupled with adjustments using MapReduce. Algorithm 1 incorporates the *dynamic partition-boundary adjusting* method to obtain sub-optimal partition boundaries in Lines 11-14 (see also Algorithm 2).

Algorithm 1. Computing Data-Partition Boundaries

```
1: input: R', S';/* Two sampling datasets */
2: output: Boundary values;
3: function MAP(key offset, values R' ∪ S')
4:   for all (o ∈ R' ∪ S') do
5:     o.value ← charact(o);/* compute o's character value */
6:     if (o ∈ R') then/* o is an object in set R' */
7:       o.flag = Flag_R; /*o.flag indicates object o in R'*/
8:     else
9:       o.flag = Flag_S; /*o.flag indicates object o in S'*/
10:    end if
11:    object ← o;
12:    emit(object, (o.value, o.flag));
13:  end for
14: end function
15: function REDUCE(key object, values (o.value, o.flag))
16:   BoundarySet ← sort(R'); /* sort dataset R' */
17:   Boundary[n] ← get(BoundarySet); /* Obtain n boundaries from BoundarySet using Balance_R scheme */
18:   for (i=1;i < n;i++) do/*n is the number of partitions*/
19:     Calculate the sizes |R'_i| and |S'_i| of R' and S' in i-th range;
20:     if (|(O(|R_i|×log2|S_i|)−O_best)/O_best| ≤ T) then/*see Formula 5*/
21:       Boundary[i] = Optimizing_Boundary(Boundary[i]);
          /* Make boundaries approach to the best case. (see Algorithm 2) */
22:     else
23:       emit(i, Boundary[i]_S);
24:     end if
25:   end for
26: end function
```

The mapper function in the first job mainly extracts features of a dataset with multiple dimensions; the feature extraction is implemented by quickly calculating a distance between two objects. We express each multi-dimensional data object as a one-dimensional value using function

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

$charact(o)$, which may be implemented in a variety of ways. Sample implementations of function $charact(o)$ are the *locality-sensitive-hashing*-based scheme [1], and the *space-filling-curves*-based scheme [2]. These two schemes are commonly adopted in parallel *kNN-join* computing. We integrate *kNN-DP* with *LSH* and *z-value*; we refer to the two *kNN-DP*-enabled solutions as *LSH+* and *z-value+*.

- LSH+*: Integrating* kNN-DP *and LSH-based* kNN-join*s*. In the *LSH* scheme, each object in sample datasets R' and S' is represented as a hash code (i.e., one-dimensional hash value) by the hash function. Then, some objects with similar hash codes are placed into the same bucket, which represents a data partition. Each bucket contains objects whose hash codes that are in a given range adjusted by our *kNN-DP* to balance computing load among buckets.
- z-value+*: Integrating* kNN-DP *and z-value-based* kNN-join*s*. *z-curve* is one of the space filling curves, which maps an object in sample dataset R' or S' to one-dimensional z value. The z values are divided into n partitions using the *Balance_R* scheme. *kNN-DP* is applied to balance the n partitions.

Algorithm 2. Optimizing_Boundary()

1: **input:** $|R'|$, $|S'|$; Initial boundary values: $boundary_values$;
2: **output:** Optimized boundary-value array: $boundary[]$;
3: $Boundary[n] \leftarrow Boundary_values$;
4: **for** (i=1,j=0;i+j < n;i++) **do**/* n is the number of partitions. */
5: $|R'_i|$, $|S'_i| \leftarrow$count(Boundary[i-1],Boundary[i]);/* Calculate sizes $|R'_i|$ and $|S'_i|$ in a range between Boundary[i-1] and Boundary[i]. */
6: **while** ($\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}} < -T$) **do** /*The ith partition's time complexity is small than the ideal value. (see Equation (6))*/
7: Boundary[i]=Boundary[i+j];/* Merging partitions and adjusting boundary */
8: j++;
9: $|R'_i|$, $|S'_i| \leftarrow$count(Boundary[i-1],Boundary[i]);/* Recalculating $|R'_i|$ and $|S'_i|$. */
10: **end while**
11: **if** ($\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}} > T$) **then**/*see Formula (6)*/
12: Best $\leftarrow$ Binary-Search(Boundary[i-1],Boundary[i]); /* Binary search the range between $|R'_i|$ and $Best$ to ensure that $|R'_i| \times log_2|S'_i| \geq T$.*/
13: $Boundary[i] \leftarrow Best$;/* Obtain the ith optimized boundary. */
14: **end if**
15: output($Boundary[i]$);
16: **end for**

The main goal of the Reduce function is to adjust the partition boundaries originated by the above map function. The partition adjustment makes an effort to balance processing time complexity of each partition, ensuring that partitions are approximately equal in size. The Reduce function takes the three steps below to accomplish partition adjustments. First, sample dataset R' is sorted in a non-decreasing order of one-dimensional values, which are determined by the aforementioned map function. The initial partition boundaries are obtained according to the *Balance_R* strategy (see Lines 16-17). Second, the processing time of each partition is estimated using time complexity analysis (see Lines 19). Finally, given a *data-skewness-degree* threshold (see Section 3.2), we must compare the estimated processing time of all the partitions against the best-case partition (i.e., ideal case). Such comparisons are implemented in the Reduce function by comparing the *data-skewness-degree* with its threshold. Comparison results govern dynamic adjustments of data objects in an imbalance partition using the *dynamic partition-boundary adjusting scheme* (see Lines 20-24), which is described in Algorithm 2.

4.4 Dynamically Adjusting Partition Boundaries

Recall we obtain initial values of partition boundaries from Algorithm 1's Line 17, which has not yet solved the data-skewness problem. Now we propose an algorithm to determine optimal boundaries, alleviating imbalanced load among data partitions. The pseudocode is delineated in Algorithm 2, which consists of the following three steps.

Step 1. This step counts the number of objects in sample sets R' and S' in the ith partition (see Line 5), in which the numbers of objects in R' and S' are expressed by $|R'_i|$ and $|S'_i|$, respectively.

Step 2. When a partition's time complexity is smaller than that of the best case, the partition will be expanded by merging with the next partition until *data-skewness-degree* $\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}}$ becomes larger than threshold $-T$ (i.e., $\frac{O(|R_i| \times log_2|S_i|) - O_{best}}{O_{best}} \geq -T$; see Lines 6-10). Thus, a small partition is extended to a large one.

Step 3. In a big data partition, we adopt the *binary-search* method to optimize the partition's lower boundary. After Step 3 is completed, we obtain the initial upper boundary and the optimized lower boundary, which form an optimized new partition. The processing time complexity of the new partition is very close to the ideal time complexity (see Lines 11-14).

4.5 Partitioning Data and Computing *kNN-join*s

The second MapReduce job has two responsibilities. First, this job aims to partition data according to the boundaries obtained in the first MapReduce job. Second, the job executes *kNN-join* operations in parallel on a Hadoop cluster.

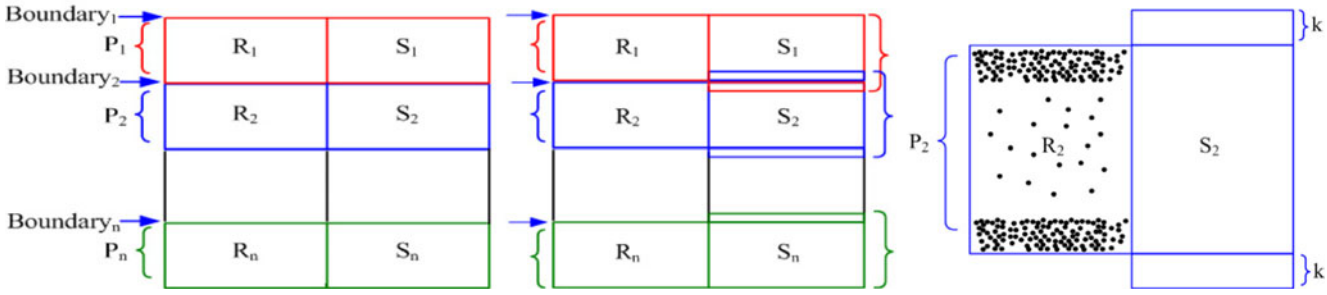
The pseudocode of the second job is delineated in Algorithm 3, which consists of the following five steps.

Step 1. This step calculates the feature value of each data object in datasets R and S. Data objects' feature values can be used to compare the similarities among these objects (see Line 5).

Step 2. Each data object is placed into a specific partition according to the partition's boundaries determined by the first MapReduce job (see the output of Algorithm 1). This step transfers partition identifiers along with a list of objects in each partition to the Reduce function (see Lines 6-13).

Step 3. Local *kNN-join* results are approximates of global *kNN-join*s. To improve the accuracy of approximate *kNN-join* results, we expand each partition of dataset S by adding a head segment and a tail segment. Specifically, the head segment of the ith partition in S contains the last k objects in the $i-1$th partition; the tail segment of the ith partition in S is comprised of the first k objects in the $i+1$th partition. Thus, the first and last partitions of dataset S have a total of

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.



(a) Data partitions without redundant data in boundary regions.

(b) Data partitions with redundant data in boundary regions.

(c) Boundary regions that benefits from redundant data.

Fig. 3. An example of data partitions without (see subfigure a) and with (see subfigure b) redundant data. Each partition's (e.g., S_2) head and tail segments include $2k$ redundant objects, which improve *kNN-join* accuracy. Subfigure c shows that if a lot of objects of a partition are sitting in the two boundary regions, the objects' k nearest neighbors may store in an adjacent partition rather than its own partition. This problem can be solved by augmenting redundant objects in the partition's head and tail segments. Subfigure c is a case that takes full benefits from redundant data.

k redundant objects; the other partitions contain $2k$ redundant objects (see Lines 15-21).

Algorithm 3. Data Partitioning and *kNN-join* Computing

```
1: input: R, S, and Boundary[n];
2: output: kNNSet;
3: functionMAP(key offset, values R' ∪ S')
4:   for all (o ∈ R ∪ S) do
5:     o.value ← charact(o);/* Compute character value of o */
6:     for (i=1; i < n; i++) do/* n is the number of partitions. */
7:       if (Boundary[i − 1] ≤ o.value ≤ Boundary[i]) then
8:         emit(i, (o.value, o.flag));/* o is placed in the ith
           partition */;
9:         if (o.flag=Flag_S) then
10:          Array_S[i] ← o.value;
11:        end if
12:      end if
13:    end for
14:  end for
15:  for (i=1;i < n-1;i++) do
16:    sort(Array_S[i]);
17:    RedundantMin ← k minimum value in Array_S[i];
18:    emit(i − 1, (RedundantMin.value, RedundantMin.flag));
       /* Add k redundant objects in set S_{i−1}. */;
19:    RedundantMax ← k maximum value in Array_S[i];
20:    emit(i + 1, (RedundantMax.value, RedundantMax.flag));
       /* Add k redundant objects in set S_{i+1}. */;
21:  end for
22: end function
23: function REDUCE (key parationID, values object)
24:  parse R_i and S_i(S_{i1}, S_{i2}, . . . , S_{im}) from (parationID, object)
25:  for all (o ∈ R_i) do
26:    for (j=1;j < m;j++) do /* m is the number of objects in
       S_i. */
27:      Dis[m] ← distance(o.value, S_ij);/* calculate distance
         from object o to object S_ij*/
28:    end for
29:    kNN(o, S) ← get(Dis[m]);/* Get k minimum value */
30:    emit(o, kNN(o, S));
31:  end for
32: end function
```

Fig. 3 plots an example of head and tail segments, which are augmented to improve *kNN-join* accuracy in MapReduce computing. Figs. 3a and 3b exhibit data partitions without and with redundant data, respectively. Except the first and last partitions, a partition's (e.g., S_2) head and tail segments include $2k$ redundant objects in dataset S (i.e., each segment contains k objects). These redundant objects help in improving the accuracy of *kNN-join*s with respect to some objects in dataset R located at a partition's top and bottom boundary regions (see, for example, Fig. 3c). If a large number of objects in a partition (e.g., R_2) are sitting in the partition's two boundary regions, these objects' k nearest neighbors may store in an adjacent partition instead of its own local partition. This problem can be solved by augmenting redundant objects in the partition's head and tail segments, which improve *kNN-join* accuracy. The accuracy improvement is closely related to the distribution characteristics of data sets. We introduce two extreme cases to shed light on the impact of data redundancy on result accuracy. In the first case, if all objects in a partition are away from the two partition boundaries (i.e., the neighbours of these objects aren't residing in the redundant data), the redundant data has no impact on the accuracy of *kNN-join*. In the second case, when most of the objects in a partition fall into the partition's two boundary regions (i.e., the k neighbours of these objects are likely to be found in the redundant data), the accuracy of *kNN-join* can be substantially increased. Fig. 3c shows a sample case that takes full benefits from our data redundancy approach. In this case, the data distribution in R_2 is imbalanced in the way that most objects are located in the boundary regions. Any data-distribution case, where very few objects are found in boundary regions, may see little data-redundancy advantage.

The redundant-data strategy's overhead depends on k, which is set k to $\sqrt{|S|}$ (see [12] for a similar k value). We quantify the redundant-data strategy's overhead by comparing the non-redundant-data and redundant-data strategies, the time complexities of which are $TC_w/o = O(|R_i| \times log_2|S_i|)$ and $TC_with = O(|R_i| \times log_2(|S_i| + 2 \times k))$, respectively. Let's note the ith partition, where $|R_i| = 10^6$, $|S_i| = 10^6$, and $k = \sqrt{|S_i|} = 10^3$. Time complexity TC_w/o and TC_with become 19,931,600 and 19,934,500, meaning that the redundant-data strategy merely introduces a negligible overhead of 0.0145 percent.

Step 4. Function $distance(o.value, S_{ij})$ calculates distances between object $o \in R$ and all the objects in dataset S (see Lines 25-28); the distances are saved and managed in array $Dis[m]$. We measure the distance between objects $r \in R$ and $s \in S$ using the discrepancy between the one-dimensional

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
The Uniform Datasets and Normal Datasets are Synthetic Data Sharing the Same Size

Dataset size	8 GB	16 GB	24 GB	32 GB
Number of objects in R set ($\times 10^7$)	1.28	2.56	3.84	5.12
Number of objects in S set ($\times 10^7$)	1.28	2.56	3.84	5.12
Number of attributes	200	200	200	200

The data size is set to 8 GB, 16 GB, 24 GB, 32 GB, *respectively.*

feature values of r and s. Euclidean distances and other types of distances may also be used to quantify the similarity between a pair of two data objects. Choosing a distance measurement method largely depends on application domains. For example, if we measure the similarity between two sets, then we apply the Jaccard distance to implement $distance(o.value, S_{ij})$. If the similarity between two sequences must be computed, then we adopt the Hamming distance. Euclidean distances are a good candidate for measuring the similarity between two vectors.

Step 5. The last step selects the smallest k values from distance array $Dis[m]$ as approximate k nearest neighbors (see Line 29).

5 Experimental Results

We implement *kNN-DP* and evaluate its performance on a real-world 24-node Hadoop cluster. Each computing node in the cluster contains an Intel E5-1620 v2 series 3.7G quad core processor, 16 GB memory, and runs on the Centos 6.4 operating system, on which Java JDK 1.6.0_24 and Hadoop 1.1.2 are installed. We apply the default Hadoop parameter configurations to set up the replication factor (i.e., three) and the numbers of Map tasks.

We run *kNN-DP* on the cluster using both synthetic and real-world datasets to conduct performance evaluation.

Synthetic Datasets. We generate the synthetic datasets in the following two steps. First, we run a random data generator to create an array of small datasets. Second, we merge the small datasets generated in the first step to enlarge the dataset size. In the two types synthetic datasets (i.e., uniform datasets and normal datasets), each of which includes 200 attributes; data size is varied from 8 GB to 32 GB with an increment of 8 GB. The first dataset type satisfies the uniform distribution, and the second one follows the normal distribution. Each dataset is divided into R set and S set, which contain a same number of objects. Table 3 summarizes the characteristics of these datasets.

Celestial Spectral Datasets. The real-world datasets tested in our experiments are the celestial spectrum data released by the China National Astronomical Observatory [13]. The celestial spectral datasets have 90 attributes. We examine three data sizes: 8 GB, 16 GB, and 24 GB.

In our experiments, we incorporate the *kNN-DP* partitioning method to existing parallel *kNN-join* algorithms. The traditional parallel *kNN-join*s algorithms considered in this study include (1) the *locality-sensitive-hashing*-based scheme or *LSH* [1], and (2) *z-curves*-based scheme or *z-value* [2]. We compare the performance of these two existing algorithms with our *kNN-DP*-based approaches. We refer to our *kNN-DP*-based schemes are *LSH+* and *z-value+*. we apply *Simple Random Sampling* and *Interval Sampling* schemes in synthetic datasets and celestial spectral datasets, respectively.

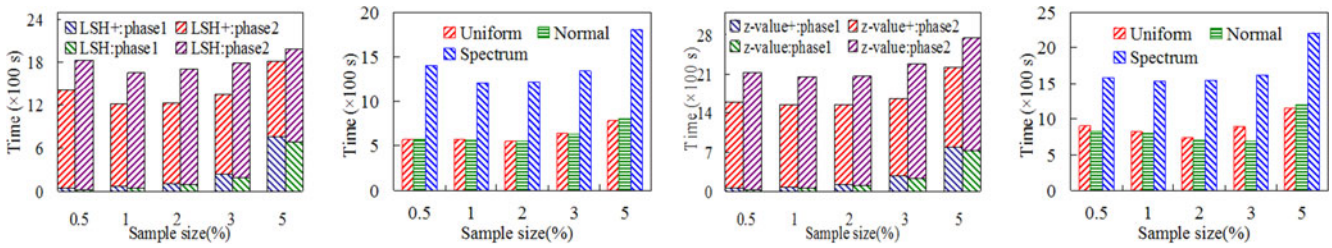
5.1 Tuning Performance

In this section, we evaluate the impacts of sample size, *data-skewness-degree* threshold (i.e., T), and parameter k.

5.1.1 Sample Size

During data pre-processing, we collect a group of sample data to capture the distribution of a given dataset. In this group of experiments, we evaluate the sampling technique by varying the sample size; we quantify the impact of sample size on *kNN-DP*'s efficiency (see Fig. 4). It is worth noting that we represent sample size as a ratio between a sample dataset size and its original dataset size.

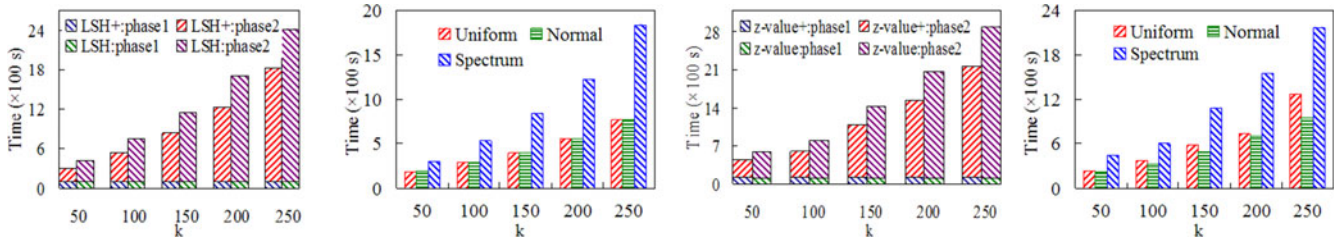
Figs. 4a and 4b show the execution time for two phases in MapReduce-based *kNN-join* algorithms, where the first phase (i.e., phase1) is to obtain data-partition boundaries in a small sample dataset and the second phase (i.e., phase2) is partitioning data in conjunction with computing *kNN-joins*. Figs. 4a and 4c reveal that the running times of our *LSH+* and *z-value+* in the first phase are longer than those of *LSH* and *z-value* due to dynamically adjusting partition boundaries. Nevertheless, the overall performances of our *kNN-DP*-based *LSH+* and *z-value+* are superior to *LSH* and *z-value*. These results indicate that our proposed *kNN-DP* can effectively alleviate data skewness to improve *kNN-join* performance. The second observation drawn from Figs. 4a and 4c is that when sample size is in a range between 1 and 2 percent, the execution times are minimized. We observe that both small and large sample sizes tend to deteriorate performance, because (1) small sample sizes are unable to accurately estimate dataset distribution, and (2) large sample sizes lead to high sampling overhead.



(a) Execution time of *LSH+* and *LSH* in celestial spectral dataset.
(b) Execution time of *LSH+* in various dataset.
(c) Execution time of *z-value+* and *z-value* in celestial spectral dataset.
(d) Execution time of *z-value+* in various dataset.

Fig. 4. Impacts of sample size on the performance of the *kNN-DP*-based *LSH+* and *z-value+*. Sample size is represented as a ratio between sample dataset size and original dataset size. The *Uniform*, *Normal*, and *Spectrum* datasets are 24 GB in size.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.



(a) Execution time of *LSH+* and *LSH* in celestial spectral dataset. (b) Execution time of *LSH+* in various dataset. (c) Execution time of *z-value+* and *z-value* in celestial spectral dataset. (d) Execution time of *z-value+* in various dataset.

Fig. 5. Impacts of *k* on the performance of the *kNN-DP*-based schemes - *LSH+* and *z-value+*. Dataset size is 24 GB, sample size is 2 percent, the number of datanodes is 8, and *T* is 2 percent.

The experimental results in Figs. 4b and 4d indicate that 1 percent is an optimal sample size for the celestial spectral dataset, and 2 percent is an ideal sample-size value for the uniform and normal datasets. The findings reveal that there is a strong correlation between sample size and dataset characteristics. Our experimental analysis suggests that sample size should be set in a range between 1 and 2 percent.

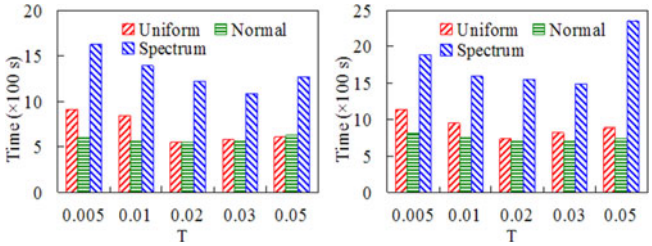
5.1.2 *Effect of* k

Next we analyze the effect of k on the performance of our *kNN-DP*-based *LSH+* and *z-value+*. Fig. 5 presents execution times of the baseline schemes (i.e., *LSH* and *z-value*) and our *kNN-DP* techniques (i.e., *LSH+* and *z-value+*) as functions of k ranging from 50 to 250. In this group experiments, dataset size is 24 GB and sample size is 2 percent. The execution times of the first phase in the *kNN-join* algorithms remains unchanged when parameter k varies from 50 to 250. The experimental results show that regardless of k values, *LSH+* and *z-value+* consistently outperform *LSH* and *z-value*. The performance improvements of *LSH+* and *z-value+* over their counterparts lie in our proposed *kNN-DP* technique, which solves the data skewness problem by balancing load through data partitioning.

The experimental results plotted in Fig. 5 indicate that running time of *kNN-join* quickly rise with the increasing value of k. Thus, in the case of a large k value, it makes more sense to incorporate *kNN-DP* to optimize the *kNN-join* computing performance.

5.1.3 *Data-Skewness-Degree Threshold*

Recall that we introduce the *data-skewness-degree* and its threshold to measure the data skewness level of each partition. In this group of experiments, we observe the sensitivity of *kNN-DP* (i.e., *LSH+* and *z-value+*) to the *data-skewness-degree* threshold T.



(a) Execution time in *LSH+*. (b) Execution time in *z_value+*.

Fig. 6. Impacts of *data-skewness-degree* threshold on the performance of *kNN-DP*-based *LSH+* and *z-value+*. The sample-size is 2 percent, dataset size is $24GB$, and the number of datanodes is 8.

Fig. 6 shows *LSH+* and *z-value+*'s efficiency in *kNN-join* computing. More specifically, we observe that execution times of *LSH+* and *z-value+* are significantly dropping when T is increased from 0.005 to 0.2. We conclude that a large *data-skewness-degree* threshold shortens running time by quickly obtaining data-partition boundaries. A second observation is that the execution times slightly go up, when T varies from 0.03 to 0.05. Such a performance degradation is attributed to the large T value that setup ill-boundaries among data partitions, which lead to imbalanced load across computing nodes.

Fig. 6a shows that when *data-skewness-degree* threshold is set 0.02 (i.e., $T = 0.02$), the uniform dataset and normal dataset's execution time is the smallest; when T is 0.03, the execution time of processing the celestial spectral dataset becomes the smallest. This performance trend indicates that the *data-skewness-degree* threshold should be tuned according to data distributions.

5.2 Extensibility

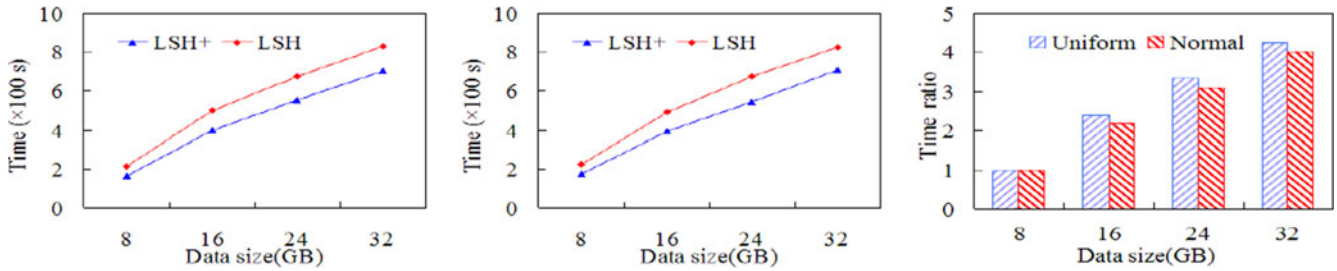
We evaluate the extensibility of *kNN-DP* by increasing the dataset size. We aim to assess the performance of *kNN-DP* when it comes processing a massive amount of data for big-data applications.

Fig. 7 exhibits the running times of *LSH+* and *LSH* as a function of dataset size. Figs. 7a and 7b demonstrate that the running times of *kNN-join* continuously increase when we enlarge the dataset size. This trend is caused by the fact that the number of objects in the R and S sets (a.k.a., object-pairs) is surging with the increasing dataset size.

Importantly, *LSH+* delivers much better performance than *LSH*, because the large number of object-pairs leads to long *kNN-join* computing time due to its high complexity (i.e., $O(|R| \times log_2|S|)$ (see also Sections 3.1). Such an improvement becomes more pronounced in cases where a datanode has an excessive number of object pairs compared with the number of object pairs in its peer nodes. *LSH+* leverages *kNN-DP* to ensure that processing times of all the nodes are well balanced, thereby effectively reducing the running time of the datanode bearing heavy load.

We also present time ratio to normalize the results plotted in Figs. 7c. The normalization allows us to compare *LSH+* overhead of processing the uniform and normal datasets. We choose 8 GB dataset as baseline readings; the time ratio of the 8 GB data is 1. The time ratios of a dataset are computed as ratios between processing times of this dataset and the baseline readings. The experimental results illustrate that the uniform data and the normal

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.



(a) Execution time of the uniform datasets. (b) Execution time of the normal datasets. (c) *LSH+*'s time ratio.

Fig. 7. Impacts of data size on *kNN-DP*-based *LSH+* and its counterpart *LSH*. The data size varies from 8 GB to 32 GB. The number of datanodes is 8, sample size is 2 percent, and T is 0.02.

data share a similar time ratio, indicating that *LSH+* has light load on both datasets.

Keeping the same parameters used in the above experiments, we compare the running time of *z-value+* and *z-value* to further analyze the extensibility of *kNN-DP* in terms of data size. Fig. 8 shows a similar trend as that plotted in Fig. 7. This experiment confirms that our *kNN-DP* optimizes the performance of both *LSH* and *z-value*.

5.3 Scalability

To evaluate the scalability performance of *kNN-DP*, we compare *kNN-DP*-based schemes to the baseline parallel *kNN-join* solutions. Figs. 9 and 10 plot the results when the number of computing nodes is varied from 4 to 24.

Figs. 9a, 9b, and 9c illustrate that the optimized speed between *LSH+* and *LSH* tends to be increased in various datasets when the number of computing nodes scales up. Due to the increasing number of computing nodes, the number of objects assigned to each node is plunging. Thus, the ideal time complexity of *kNN-join* becomes low (see O_{best} in Section 3.2). However, the worst-case complexity remains unchanged (see also O_{worst} in Section 3.4). Employing *kNN-DP*, the time complexity of *LSH+* is approaching to O_{best}. According to this trend, it is not a surprise that *LSH+* always outperforms *LSH* with the increasing number of datanodes.

Figs. 9d reveals the speedups of *LSH+* processing various datasets. For three different datasets (i.e., uniform, normal, and celestial spectral datasets), *LSH+* achieves a linear speedup thanks to low I/O overhead under heavy workload. This performance trend shows evidence that *LSH+* is a parallel *kNN-join* solution exhibiting high scalability, which is contributed by the following two factors. First, *LSH+* applies *kNN-DP* to improve the parallel computing performance of *LSH*. Second, all the nodes independently compute *kNN-join* from local data objects in parallel. The number of data objects assigned to each node is inversely proportional to the number of nodes. Scaling up the cluster decreases the number of objects distributed to each node, thereby shortening *LSH+*'s overall running time and speeding up the *kNN-join* performance. If we increase the number of partitions along with the number of computing nodes , the scalability performance will be further improved.

Keeping the parameters unchanged, we compare the scalability performance of *z-value+* and *z-value*. The scalability trends plotted in Fig. 10 are consistent with those in Fig. 9. The experimental results confirm that like *LSH+*, *z-value+* has high scalability performance.

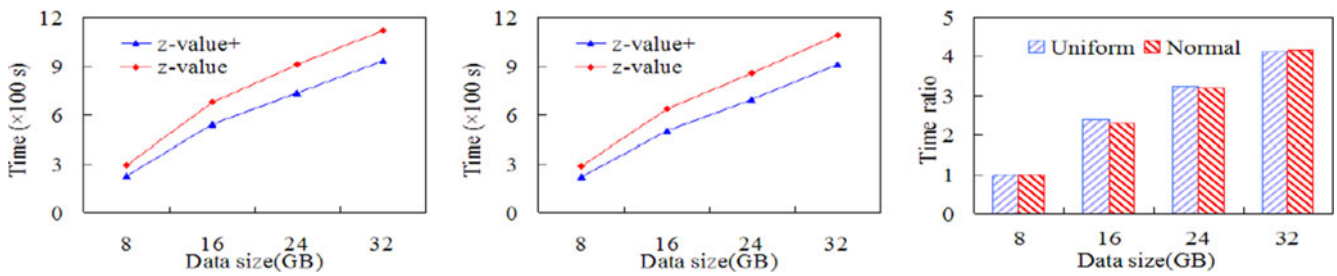
6 Related Work

In this section, we review existing techniques in the areas of *kNN* queries and data skewness.

6.1 *kNN* and Parallel *kNN* Queries

Finding nearest neighbors on high-dimensional data is still an important endeavor. Hinneburg et al. [14] argued that nearest neighbor queries are well defined using commonly metrics such as Manhattan and Euclidean distances. There has been a lot of work in nearest neighbor queries for high-dimensional data. A basic algorithm for nearest-neighbor queries is a branch-and-bound method using R*-tree [15], which is traversed based on the distance between a query point and each bounding region.

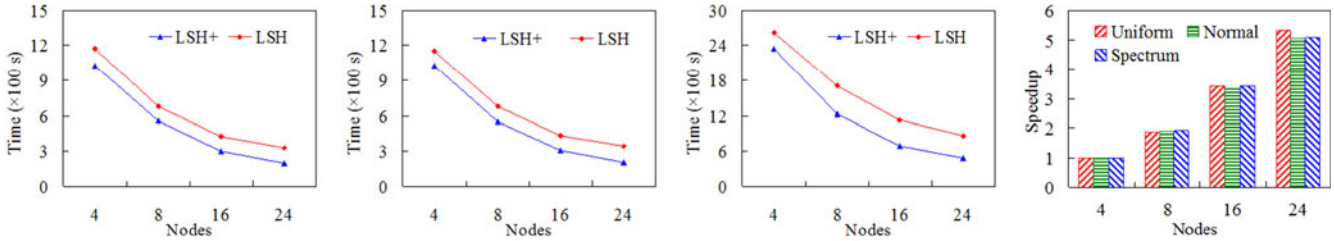
A handful of approaches were taken to find approximate solutions to nearest neighbor queries. Tao et al. [16] developed a new access method called spatial-inverted index, which extends the conventional inverted index to multidimensional data. Tao's algorithms can answer nearest neighbor queries with keywords in a real-time manner. Agarwal



(a) Execution time of the uniform datasets. (b) Execution time of the normal datasets. (c) *z-value+*'s time ratio.

Fig. 8. Impacts of data size on *kNN-DP*-based *z-value+* and its counterpart *z-value*. The data size varies from 8 GB to 32 GB. The number of datanodes is 8, sample size is 2 percent, and T is 0.02.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.



(a) Execution time of *LSH+* and *LSH* in uniform dataset. (b) Execution time of *LSH+* and *LSH* in normal dataset. (c) Execution time of *LSH+* and *LSH* in celestial spectral dataset. (d) *LSH+*'s speedup.

Fig. 9. Scalability comparisons between *kNN-DP*-based *LSH+* and its counterpart *LSH*. Scalability performance is measured by varying the number of datanodes in the Hadoop cluster. Data size is 24 GB, sample size is 2 percent, and T is 0.02.

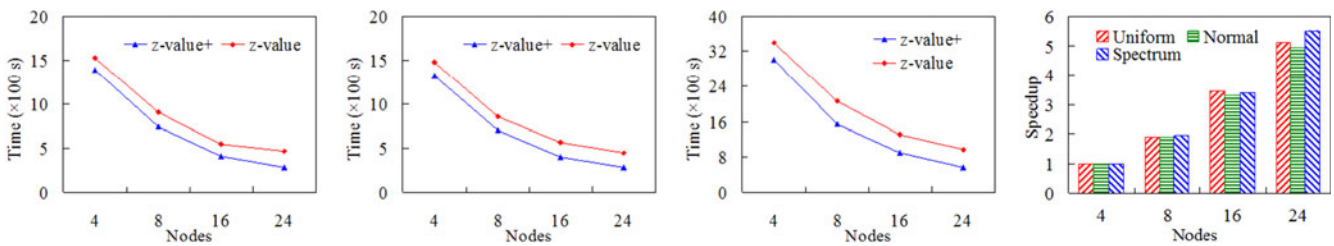
et al. [17] studied nearest neighbor searching under uncertainty, where the location of each input point and/or query point is specified as a probability density function. In *voronoi*-based nearest neighbor searching, Zhang et al. [18] extended the notion of *voronoi* cells to support uncertain objects. They further developed the *PV-index* method to evaluate probabilistic nearest neighbor queries over uncertain data [18]. Nolen et al. [19] introduced a *R-Forest* method to process efficient queries on high-dimensional data. *R-Forest*, being able to handle different data distributions, takes an advantage of data distribution without any additional parameter tuning.

Both the *kNN-search* and *kNN-join* operations are expensive; this statement is especially true on large data and high-dimensional datasets. As such, processing large data in parallel and distributed fashions is becoming a popular practice. An early work conducted by Kitsuregawa and Ogawa was focused on parallel join algorithms for relational join operators in a shared-nothing multiprocessor environment [20]. *kNN* queries were implemented in MapReduce by leveraging the *voronoi* diagram and the *locality-sensitive-hashing* (*LSH*) method, respectively. Akdogan et al. [21] adopted a *Voronoi-diagram-partitioning*-based approach using MapReduce to answer range search and *kNN* search queries. In their approach, each object of the dataset is treated as a pivot, which governs partitioning of a data space. Obviously, Akdogan's method incurs high maintenance cost and computation cost when the number of dimensions increases, thereby be limited to handle 2-dimensional datasets. Stupar et al. [1] proposed the RankReduce approach using *locality-sensitive-hashing* (*LSH*). The *LSH* algorithm implemented in MapReduce assigns similar objects to one fragment in the distributed file system enabling an effective selection of potential candidate neighbors reduced to a set of k-nearest neighbors. Lu et al. [10] presented an implementation of *kNN-join* operators using MapReduce. Lu's *kNN-join* algorithms, applying an effective mapping mechanism that exploits pruning rules for distance filtering, minimizes the number of replicas and shuffling cost. Zhang et al. designed a MapReduce-based algorithm named *H-zkNNJ* [2] to perform efficient parallel *kNN-join*s on large data by the virtue of block-nested-loop joins and R-tree indexing.

6.2 Data Skewness

The data-skewness problem is a common and critical issue in parallel distributed systems. Data skewness has been studied in the realm of parallel database; in the past few years, the data-skewness issue has been addressed in MapReduce programming. For example, Dean and Ghemawat first discovered the straggler problem and proposed its solution in [22]. Straggler is one of the common causes that enlarge the total execution time of MapReduce programs. That is, a machine takes an unusually long time to complete one of the last few map or reduce in tasks. A solution to this problem is to make use of speculative execution to back up the last few running tasks, thereby decreasing job execution time by 44 percent. Zaharia et al. [23] extended the approach to heterogeneous clusters; their algorithm speculatively executes a replica task for the ones with a slow progress rate. However, executing a redundant copy for a data-skewed task may result in wasted resources, since the duplicate tasks with data skewness still have the same amount of data. Later, Ganeshi et al. proposed a new method called Mantri [24], which uses a tasks process bandwidth to calculate the tasks remaining execution time. Mantri may not be able to alleviate data-skewness problems due to inadequate available resources during task execution.

Recently, data partitioning to solve the data-skewness problems in MapReduce has been extensively investigated [25], [26], [27]. For instance, Kwon et al. [25]



(a) Execution time of *z-value+* and *z-value* in uniform dataset. (b) Execution time of *z-value+* and *z-value* in normal dataset. (c) Execution time of *z-value+* and *z-value* in celestial spectral dataset. (d) *z-value+*'s speedup.

Fig. 10. Scalability comparisons between *kNN-DP*-based *z-value* and its counterpart *z-value*. Scalability performance is measured by varying the number of datanodes in the Hadoop cluster. Data size is 24 GB, sample size is 2 percent, and T is 0.02.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

implemented a system called SkewReduce, which optimizes data partitions for spatial-feature-extraction applications by operating pre-process extractions and sampling procedures. Kwon's approach can mitigate potential data skewness, but this scheme—being heavily rely on users' domain knowledge—is limited to specific types of applications.

Ibrahim et al. [26] have studied data skewness occurred in the reduce phase by collecting key frequency information on each node. The downside of Ibrahim's solution is that it does not split large blocks containing all data sharing the same key at the same reducer; as a result, such non-partitioned large blocks lead to unbalanced loads in reducers. Gufler et al. [28] proposed an approach to partitioning intermediate data into more portions than the number of reducers, followed by applying a greedy bin-packing method to allocate the partitions to a set of reducers upon completing all map tasks. The scheduling policy of Gufler's scheme does not preserve the order of original reduce outputs. In a follow-up work, Gulfler et al. proposed a sampling method called TopCluster [29] to estimate an intermediate data distribution before executing actual jobs. After the estimation is done, data are partitioned to balance computing load across all reducers. It is worth noting that the sampling phase can be time-consuming.

The SkewTune system, tackling the data-skewness problem from an intriguing angle [30], does not aim to evenly partition intermediate data at the beginning. Rather, SkewTune dynamically adjusts data partitions. Specifically, after detecting a straggler task, SkewTune repartitions unprocessed data assigned to new tasks running on other nodes. SkewTune imposes extra overhead caused by repartitioning data and concatenating final outputs. Similarly, Kolb et al. [31] proposed two approaches, BlockSplit and PairRange, to handle data skewness for entities resolution using MapReduce. A block distribution matrix needs to be distributed using the distributed Cache technique in MapReduce so that the entities can be evenly distributed to all reducers. PairRange outperforms BlockSplit, because PairRange guarantees that all reducers receive the same number of entities. These two approaches have to run an additional MapReduce job to generate the block distribution matrix or BDM. Chen et al. proposed a new sampling method—*LIBRA* [32]—for general user-defined MapReduce programs. LIBRA addresses the data-skewness problem among the reducers of MapReduce applications through sampling intermediate data in a heterogeneous environment. MapReduce program using LIBRA still have to wait for the completion of all map tasks.

7 Conclusion and Future Work

In this study, we developed a data partitioning approach called *kNN-DP* for *kNN-join*. *kNN-DP* alleviates load imbalance incurred by the data skewness problem. *kNN-DP* achieves the equitable data partitioning by optimizing the partition boundaries. Specifically, *kNN-DP* has three salient and advanced features. First, the sampling technique is utilized to quickly capture the data distribution of a big dataset through profiling a small sample set. Second, *kNN-DP* dynamically adjusts partition boundaries by assessing time complexity of each partition in a sample dataset. Optimized partition boundaries offer smart data-partitioning guidelines. Third, to improve the accuracy of parallel *kNN-join* using MapReduce, *kNN-DP* employs a redundant-data strategy, which augments each node's local data by a small amount of redundant data.

We designed two *kNN-DP*-based schemes called *LSH+* and *z-value+*, which seamlessly integrate *kNN-DP* with the *LSH* and *z-value* algorithms for *kNN-join* computing. We implemented *LSH+* and *z-value+* on a 24-node Hadoop cluster driven by both synthetic and real-world high-dimensional datasets. The extensive experiments show that our *kNN-DP* significantly improves the overall performance of *LSH* and *z-value*. Furthermore, *kNN-DP* is capable of enhancing the extensibility and scalability of the *LSH* and *z-value* solutions.

As a future research direction, we plan to integrate *kNN-DP* with *Voronoi-diagram-based kNN-join*. We will divide objects in sample datasets R' and S' into voronoi cells (a.k.a., partitions) based on a *voronoi* diagram. There is a pivot object in each *voronoi* cell. An object belongs to a *voronoi* cell, if the object's distance to the cell's pivot is the shortest compared to the object's distances to all the other pivots. *kNN-DP* will be extended to elect the most appropriate pivots to balance *kNN-join* operations in *voronoi* cells.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of P. R. China (61572343), and the Fund for Shanxi "1331 Project" Key Innovative Research Team. Xiao Qin's work is supported by the U.S. National Science Foundation under Grant CCF–0845257(CAREER).

References

[1] A. Stupar, S. Michel, and R. Schenkel, "Rankreduce-processing k-nearest neighbor queries on top of MapReduce," in *Proc. 8th Workshop Large-Scale Distrib. Syst. Inform. Retrieval*, 2010, pp. 13–18.

[2] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in MapReduce," in *Proc. ACM 15th Int. Conf. Extending Database Technol.*, 2012, pp. 38–49.

[3] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *Geoinformatica*, vol. 14, no. 1, pp. 55–82, 2010.

[4] C. Yu, B. Cui, S. Wang, and J. Su, "Efficient index-based knn join processing for high-dimensional data," *Inform. Softw. Technol.*, vol. 49, no. 4, pp. 332–344, 2007.

[5] C. Xia, H. Lu, B. C. Ooi, and J. Hu, "Gorder: An efficient method for knn join processing," in *Proc. 3th Int. Conf. Very Large Data Bases-Vol. 30*, 2004, pp. 756–767.

[6] M. Batko, C. Gennaro, and P. Zezula, "A scalable nearest neighbor search in p2p systems," in *Proc. Int. Workshop Databases, Inform. Syst. Peer-to-Peer Comput.*, 2004, pp. 79–92.

[7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Parallel processing of spatial joins using r-trees," in *Proc. IEEE 12th Int. Conf. Data Eng.*, 1996, pp. 258–265.

[8] M. Jang, Y.-S. Shin, and J.-W. Chang, "A grid-based k-nearest neighbor join for large scale datasets on MapReduce," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, 2015, pp. 888–891.

[9] G. Song, J. Rochas, F. Huet, and F. Magoules, "Solutions for processing k nearest neighbor joins for massive data on MapReduce," in *Proc. 23rd Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2015, pp. 279–287.

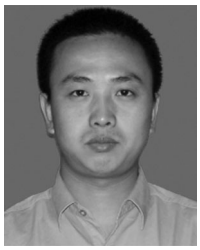
[10] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," *Proc. VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[11] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[12] M. Bouguessa and S. Wang, "Mining projected clusters in high-dimensional spaces," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 4, pp. 507–522, 2009.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.

[13] J. Zhang, S. Zhang, K. H. Chang, and X. Qin, "An outlier mining algorithm based on constrained concept lattice," *Int. J. Syst. Sci.*, vol. 45, no. 5, pp. 1170–1179, 2014.
[14] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, "What is the nearest neighbor in high dimensional spaces?" in *Proc. 26th Internat. Conf. Very Large Databases*, 2000, pp. 506–515.
[15] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, 1999.
[16] Y. Tao and C. Sheng, "Fast nearest neighbor search with keywords," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 4, pp. 878–888, Apr. 2014.
[17] P. K. Agarwal, B. Aronov, S. Har-Peled, J. M. Phillips, K. Yi, and W. Zhang, "Nearest neighbor searching under uncertainty II," in *Proc. 32nd ACM SIGMOD-SIGACT-SIGAI Symp. Principles Database Syst.*, 2013, pp. 115–126.
[18] P. Zhang, et al., "Voronoi-based nearest neighbor search for multi-dimensional uncertain databases," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 158–169.
[19] M. Nolen and K.-I. Lin, "Approximate high-dimensional nearest neighbor queries using r-forests," in *Proc. 17th Int. Database Eng. Appl. Symp.*, 2013, pp. 48–57.
[20] M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC)," in *Proc. Int. Conf. Very Large Data Bases*, 1990, pp. 210–221.
[21] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based geospatial query processing with MapReduce," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 9–16.
[22] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapReduce performance in heterogeneous environments," in *Proc. Usenix Symp. Operating Syst. Des. Implementation*, 2008, pp. 29–42.
[24] G. Ananthanarayanan, et al., "Reining in the outliers in map-reduce clusters using mantri," in *Proc. Usenix Symp. Operating Syst. Des. Implementation*, 2010, pp. 265–278.
[25] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 75–86.
[26] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for MapReduce in the cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.
[27] Y. Xun, J. Zhang, X. Qin, and X. Zhao, "FiDoop-DP: Data partitioning in frequent itemset mining on hadoop clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 101–114, Jan. 2017.
[28] G. Benjamin, A. Nikolaus, R. Angelika, and K. Alfons, "Handling data skew in MapReduce," in *Proc. Int. Conf. Cloud Comput. Serv. Sci.*, 2011, pp. 574–583.
[29] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in MapReduce based on scalable cardinality estimates," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 522–533.
[30] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in MapReduce applications," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2012, pp. 25–36.
[31] L. Kolb, A. Thor, and E. Rahm, "Load balancing for MapReduce-based entity resolution," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 618–629.
[32] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: Lightweight data skew mitigation in MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2520–2533, Sep. 2015.



Xujun Zhao received the MS degree in computer science and technology from the Taiyuan University of Technology (TYUT), China. He is currently working toward the PhD degree with the Taiyuan University of Science and Technology (TYUST). His research interests include data mining and parallel computing.



Jifu Zhang received the BS and the MS degrees in computer science and technology from the Hefei University of Tchnology, China, and the the PhD degree in pattern recognition and intelligence systems from the Beijing Institute of Technology, in 1983, 1989, 2005, respectively. He is currently a professor in the School of Computer Science and Technology, Taiyuan University of Science and Technology. His research interests include data mining and parallel computing. He is a member of the IEEE.



Xiao Qin received the BS and the MS degrees in computer science from the Huazhong University of Science and Technology, China, and the PhD degree in computer science from the University of Nebraska-Lincoln, Lincoln, in 1992, 1999, and 2004, respectively. Currently, he is a professor in the Department of Computer Science and Software Engineering, Auburn University. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He received the U.S. NSF Computing Processes and Artifacts Award and the NSF Computer System Research Award in 2007 and the NSF CAREER Award in 2009. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:34:56 UTC from IEEE Xplore. Restrictions apply.