IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 7, 2019, accepted July 20, 2019, date of publication July 25, 2019, date of current version August 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931005

Scalable Distributed kNN Processing on Clustered Data Streams

MIN YANG[1], YIXUAN ZUO[2], MENG CHEN[3], (Member, IEEE), AND XIAOHUI YU[4], (Member, IEEE)

[1]School of Computer Science and Technology, Shandong University, Jinan 250002, China
[2]School of Computer Science and Technology, Shandong Jianzhu University, Jinan 250002, China
[3]School of Software, Shandong University, Jinan 250002, China
[4]School of Information Technology, York University, Toronto, ON M3J 1P3, Canada

Corresponding author: Meng Chen (mchen@sdu.edu.cn)

This work was supported in part by the Fundamental Research Funds of Shandong University, in part by the Natural Science Foundation of Shandong Province of China under Grant ZR2019BF010, in part by the National Natural Science Foundation of China under Grant 61572289, and in part by the NSERC Discovery Grants.

ABSTRACT Recommender systems provide an important tool for users to find interested items from the massive amount of user-generated contents. As user interests often change over time and contents become available in a streaming fashion, it is highly desirable to support real-time recommendation that can adapt to changes in user interests and contents. If we represent both user interests and items by high-dimensional points in the same vector space, we can recommend to the user the k items that are the nearest neighbors (kNN) of the user. The problem of real-time recommendation, thus, translates to computing the kNNs based on the most recent items when the user interests change. As such, the main issue we tackle in this paper is to efficiently process high-dimensional kNN queries over a sliding window on data streams. In particular, we are interested in developing a scalable distributed solution to be able to handle the ever-increasing number of users and volume of data. We propose a new index structure called the dynamic bounded rings index (DBRI) to index the data points in data streams. The basic idea is to first find a set of pivots and assign all points to their nearest pivot to form subsets and then partition each subset into finer-grained bounded rings that can be dynamically adjusted as points change. The design of DBRI lends itself to easy adoption in a distributed setting. We further present the distributed high-dimensional kNN query algorithm (DHDKNN) based on DBRI, aiming at reducing both the communication and the computational cost of query processing. The experiments demonstrate that our algorithm scales well and significantly outperforms the existing methods.

INDEX TERMS Real-time recommendation, data streams, k nearest neighbor, distributed processing.

I. INTRODUCTION

As user-generated contents are experiencing an explosive rate of growth, recommender systems offer users a great way to access information of their particular interest out of the vast volume of data. For social applications where user interests change rapidly over time and contents become available as continuous streams, traditional recommender systems no longer suffice, because most traditional recommender systems assume that the user interests are stable and operate in an offline fashion. It is thus highly desirable to develop real-time recommendation methods that are capable of updating the recommendations online accounting for the most recent changes in user interests and data items.

To provide effective real-time recommendation, we focus on a popular approach to recommender systems – content-based filtering, which is based on a characterizing description/representation of the item and a profile of the user interests [1], [2]. In particular, items (such as tweets and images) are represented by high-dimensional points in vector space (a.k.a. feature vectors) obtained using one of the many methods available, such as tf-idf computation or item embeddings [3], [4]. Users' interests (as reflected by what the users have posted or read recently, for example) are also transformed into points in the same space. With a given distance metric, we recommend to the user those items whose

The associate editor coordinating the review of this manuscript and approving it for publication was Fatih Emre Boran.

corresponding points are the k nearest neighbors (kNN) of the point representing that user.

The main technical challenge in supporting such a real-time recommendation scheme lies in the efficient computation of the new kNNs when the user profiles are updated and/or when new data items become available. When a user's interest changes (e.g., the user posts a new tweet, or the user comments on an images), its point representation is updated correspondingly and its kNNs may have to be recomputed as well. Since items are generated rapidly in a streaming fashion (with new items becoming available and old items expire), the problem of updating the recommendation can be cast as one of efficiently processing the kNN queries over data streams.

In the presence of large volume of items and users, the computational power required often exceeds the capacity of a single server. We therefore aim to find a scalable solution in a distributed setting. We assume that the solution would run on a cluster that has a master-slave configuration consisting of a master and multiple slaves, which is a general model followed by many existing systems. To ensure fast processing, we also assume that all data of interest are stored in main memory, which is a reasonable assumption in distributed clusters.

While there exist a few notable works on distributed high-dimensional kNN processing [5]–[17], they cannot be trivially adapted to our setting. Some [6], [10] directly distribute all the points to different nodes and construct a local index at each node to accelerate the query processing. However, as the queries have to be sent to all nodes for kNN search, the communication cost is large, especially when the queries are consecutively submitted at all snapshots. Other works try to build a global index (usually a tree-based-structure [11], [12]) in the master node to avoid the kNN search on all partitions. These methods either incur an unpredictable number of iterations to find a final search region, or suffer from the bottleneck in the master node due to high maintenance cost, especially on data streams. Besides, there are some approximate high-dimensional kNN query methods such as locality sensitive hashing (LSH) [15], [17] which can be implemented in a distributed environment. However, they are mostly designed for batch processing on a disk-resident dataset, and it is non-trivial for them to deal with real-time applications. In this paper, we only focus on providing exact kNN results.

To address the challenges, we propose a new distributed index called the Dynamic Bounded Rings Index (DBRI). DBRI partitions the data points into bounded rings surrounding some selected pivots, and each data point can only fall in one bounded ring determined based on its distances to the pivots. We fix the maximum and minimum capacity of each ring, and use two dynamic boundaries (i.e., the upper bound and the lower bound) to control the number of points in each ring. DBRI suits the distributed environment well as individual rings can be allocated to different nodes, and the allocation can be adjusted over time in response to changing data distributions. As the number of points in each ring is always within a threshold, load balancing is achieved over the distributed cluster.

Furthermore, we propose the distributed high-dimensional kNN query processing algorithm (DHDKNN) based on DBRI. DHDKNN first searches for several candidate bounded rings to set the primary distance to its k-th nearest neighbor. Then, it uses this distance to determine the search region and to calculate the exact kNN results by only searching the intersected bounded rings. In this way, DHDKNN involves two iterations in the kNN processing, which leads to less communication cost and predictable performance compared with existing methods.

We implement DBRI and DHDKNN on top of Apache Storm, an open source distributed platform for stream processing. Extensive experiments are conducted to evaluate the performance of our methods. In summary, we make the following contributions.

1) We propose a framework of content-based real-time recommendation based on kNN query processing on data streams. We propose a new distributed index called the Dynamic Bounded Rings Index (DBRI) to support query processing.
2) We propose an efficient distributed kNN query algorithm (DHDKNN) based on DBRI for the processing. DHDKNN processes the kNN query in only two iterations and thus has less communication cost and predictable performance in a distributed setting.
3) We implement DBRI and DHDKNN on top of Apache Storm and conduct extensive experiments on real datasets to demonstrate the superiority of our methods over baseline methods.

Roadmap: The remainder of the paper is organized as follows. In Section II we present the related work. Section III presents the overview, and Section IV describes the DBRI index. Section V presents the DHDKNN algorithm for kNN query processing. In Section VI we present the experimental results, and Section VII concludes the paper.

II. RELATED WORK

A. RECOMMENDER SYSTEMS

Recommender systems have been widely studied in the literature. Content-based filtering and collaborative filtering (CF) are the two leading recommendation paradigms. As social websites become popular, real-time recommender systems [18]–[20] attract much attention, with recommendations updated more frequently to match users' instant need. However, most of these methods use CF to produce recommendation which often suffer severely from the ''cold start'' and data sparsity problem. Yang *et al.* [21] propose to use continuous kNN join processing for real-time recommendation. They propose two tree-based structures to support fast updates. However, their work cannot capture changes in user interests and lacks the scalability to handle massive data. Huang *et al.* [22] build a general framework for recommender system on Storm named TencentRec for real-time recommendation over streams, but focus mainly on CF. In contrast,

we propose solutions for real-time recommendation using content-based filtering, which in general suffers less from the "cold start" problem as the user interests can be readily derived in the situations we consider.

B. HIGH-DIMENSIONAL kNN QUERY PROCESSING

Many techniques have been proposed for high-dimensional kNN query processing, such as data approximation (e.g., VA-File [23]), one-dimensional transformation (e.g., iDistance [24]), and dimensionality reduction (e.g., Δ-tree [25]). For approximate similarity queries, locality-sensitive hashing (LSH) [15] is an efficient method which can be considered as performing probabilistic dimensionality reduction. Distributed high-dimensional kNN processing has also been studied in the literature. Ali *et al.* [11] develop the PN-tree as an efficient data structure for parallel and distributed multidimensional indexing, which can be used for the computation of kNN queries for certain distance metrics. Zhang *et al.* [26] propose the Voronoi diagram-based partitioning method for efficient kNN joins using MapReduce. Choi and Lee [12] propose to build a tree on the master node to manage the VA-files of local clusters for approximate high-dimensional kNN processing. Haghani *et al.* [17] propose an method using LSH for distributed approximate kNN search. However, these works either use complex hierarchical structures or use batch-processing schemes on static datasets which is very expensive to handle in the presence of consecutive updates and queries on data streams.

KNN query processing algorithms on data streams are mostly studied in a low-dimensional setting. Bohm *et al.* [27] aim at efficient processing of exact kNN queries. They use a grid structure to index queries and use skyline techniques to monitor query results. Tao and Papadias [28] propose a repetitive query processing approach with a TPR-tree for answering kNN queries. Yu *et al.* [16] propose a dynamic strip index for efficient distributed processing of kNN on two-dimensional moving objects. However, these structures can be outperformed by a simple sequential scan when the dimensionality of data increases due to "the curse of dimensionality", and thus cannot work well in our problem.

III. OVERVIEW

Most types of data in social applications (e.g., text, image, video) can be represented by points in a vector space. For example, images can be transformed to a 128-dimensional point using the scale-invariant feature transform (SIFT) algorithm [29]. Texts like tweets can also be represented by fixed-length vectors using distributed representation methods [3], [30]. Besides, a variety of approaches using a user's historical interested items to represent his/her interest by high-dimensional points as well have been proposed in the literature [31], [32]. Hence, an effective approach to content-based filtering is to recommend to the user his/her k nearest neighbors (kNN) items, given a distance metric. Without loss of generality, we use the Euclidean distance to measure the similarity between the user interest and the item, but other distance metrics can also apply to our methods.

As a user's interest often changes over time, we use a sliding window on the user's historical interested items to monitor the interest changes. Only recent items in the window are considered available to reflect a user's current interest. When an "old item" expires from or a new item appears in the sliding window, a new point representing the user's current interest is submitted for kNN processing to acquire "fresh" recommendations. Considering that there are often millions of users online in a social application, it is a big challenge to efficiently process these kNN queries to make quick response to the update of users' interests.

Besides, considering that the user generated content are of vast volume, our real-time recommendation system shall have the scalability to handle the massive data. We therefore choose to process the data in distributed clusters. The general master-slave model which has been adopted by many existing systems (e.g., Apache Storm [33], S4 [34]) is used to represent the distributed cluster. We aim to design an efficient distributed index and kNN query algorithm on data streams to support our real-time recommendation scheme. As items are generated rapidly in a streaming fashion in social applications and users often only care about "fresh" data, we use another sliding window W on the item stream to focus only on the most recent items. Some notations used in this paper are given in Table 1.

TABLE 1. Summary of notations.

W	the sliding window
D	data set on the sliding window
P	the pivot
p, q	data point and query point
BR	the bounded ring
k	the number of nearest neighbors
lb, ub	the lower bound and upper bound
λ, Λ	the minimum and maximum capacity
Ω	the point set of a bounded ring
N	the size of a set
α, β	associated parameters of DHDKNN algorithm
C^{BR}	the candidate bounded ring set
$dist$	distance
S^q	the search region of the query q
$cdknn$	the radius of the search region
R_q	the kNN results set of the query q

IV. DYNAMIC BOUNDED RINGS INDEX

In this section, we propose a new index called the Dynamic Bounded Rings Index (DBRI), a main-memory-based index to support high-dimensional kNN processing on data streams in distributed settings.

A. INDEX STRUCTURE OF DBRI

DBRI adopts the data partitioning technique and indexes the bounded rings surrounding some selected pivots as shown in Figure 1. The construction of DBRI is in two steps: (1) First, we select a certain number of pivots which are set to

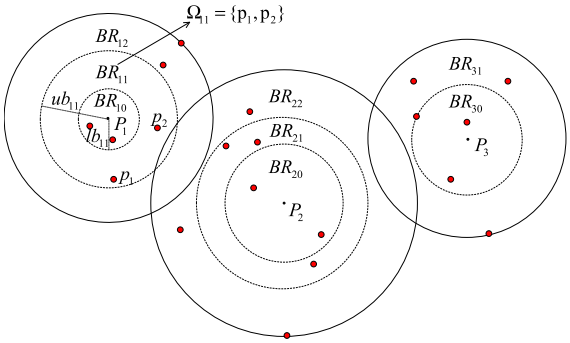

FIGURE 1. The DBRI index structure.

the clustering centers (using k-means method, for example) and assign each point in the dataset into the point set Ω_i of their nearest pivot P_i. (2) Then, we further partition each point set Ω_i into bounded rings based on the distance between the point and its nearest pivot. In this way, each point in the dataset is in a single bounded ring.

Each bounded ring BR_{ij} takes the form $\langle i, id_j, lb_{ij}, ub_{ij}, \Omega_{ij}\rangle$, where i is the unique number identifier of the corresponding pivot and id_j is the identifier of the ring, so that (i, id_j) is the unique key of a bounded ring. lb_{ij} and ub_{ij} are the minimum and maximum distance between the boundary of the ring BR_{ij} and the pivot P_i. Ω_{ij} is an unordered list of points that fall into the ring, that is, $\forall p_x \in \Omega_{ij}, lb_{ij} \leq dist(p_x, P_i) < ub_{ij}$; meanwhile, $\forall y \neq i, dist(p_x, P_i) < dist(p_x, P_y)$. Notice that although there may be certain overlapping between a pair of rings, there are no intersections between point sets, i.e., each point in the dataset must fall into a single bounded ring. So we have $\bigcup \Omega = D$ and $\forall j \neq y, \Omega_{ij} \cap \Omega_{xy} = \phi$.

Notice that after the point set for each pivot is chosen, we do not partition the point set into rings of equal width. Rather, we determine the boundaries totally based on the data so that each ring typically has unequal width. The intuition is to make each ring contain a similar number of points. Each ring must contain at least λ and at most Λ points, that is, $\lambda \leq |\Omega| \leq \Lambda (\lambda \ll \Lambda)$. We call λ and Λ the *minimum capacity* and *maximum capacity*. The rings are split and merged to meet this condition when there are updates (insertion or expiration). Typically, we have $|\Omega_i| \gg \lambda$. In the rare case that there are less than λ points assigned to a pivot, we handle it as a special case in query processing. For simplicity of analysis, we assume without loss of generality that at any time for the data stream we have $|\Omega_i| \gg \lambda$.

We use a table to store the index. Each row of the table is a tuple $\langle i, List_i(BR)\rangle$, where $List_i(BR)$ is an array list of bounded rings for pivot P_i. Each list is sorted in an ascending order according to the distance between the boundaries of the elements and the pivot.

B. INDEX MAINTENANCE

When a new point p' is inserted into the sliding window, we first search for its nearest pivot P_i by computing the distances to all pivots. Then, we add p' to the bounded ring BR_{ij} in $List_i(BR)$ where p' falls into its boundaries. Specially, when the distance between p' and P_i is larger than the maximum upper bound of $List_i(BR)$, we add p' to the outermost bounded ring and update its upper bound to $dist(p', P_i)$. The split operation is triggered if the number of points in the bounded ring exceeds the maximum capacity after insertion. To adapt to the data distribution, we follow the strategy to split the bounded ring and generate two new bounded rings that contain a similar number of points.

When a data point expires from the sliding window, meaning that it is not "fresh" any more, we delete it from the index and make it no longer searchable. The deletion on DBRI is done after searching for the bounded ring that contains the point p'. After the deletion on bounded ring BR_{ij}, we will merge BR_{ij} with its adjacent ring in the same pivot bounded ring list $List_i(BR)$ if BR_{ij} contains less than λ points. If BR_{ij} has an adjacent ring on both sides, it will be merged to the one with less points.

C. COST ANALYSIS

Let N_D be the total number of points in the sliding window W. We use N_P and N_r to denote the number of pivots and the average number of points in each bounded ring. Clearly, we have $\frac{N_D}{\Lambda} \leq N_r \leq \frac{N_D}{\lambda}$. We have the following theorem on the time complexity of the maintenance of DBRI.

Theorem 1: Let $Cost_{insert}$, $Cost_{delete}$, $Cost_{split}$, $Cost_{merge}$ be the time cost of the insert, delete, split, and merge operations on maintaining DBRI. We have

$$Cost_{insert} = b_0 N_P + b_1 \log \frac{N_D}{N_P N_r}$$
$$Cost_{delete} = b_0 N_P + b_1 \log \frac{N_D}{N_P N_r} + b_2 N_r$$
$$Cost_{split} = b_3 \Lambda + b_4 \log \frac{N_D}{N_P N_r}$$
$$Cost_{merge} = b_5 \lambda + b_6 \log \frac{N_D}{N_P N_r}$$

for some constants $b_0, b_1, b_2, b_3, b_4, b_5, b_6$.

Proof: From the definitions of N_D, N_P, and N_r, we can evaluate the number of bounded rings for each pivot as $\frac{N_D}{N_P N_r}$. In the insert operation, we first search for the nearest pivot which costs time $b_0 N_P$. Then, we need to find a bounded ring of the pivot in its bounded ring list $List(BR)$. As $List(BR)$ is an ordered list, we can find the bounded ring and append the new point in its point list in $b_1 \log \frac{N_D}{N_P N_r}$. Thus, the time complexity of the insert operation is $b_0 N_P + b_1 \log \frac{N_D}{N_P N_r}$. Similarly, in the deletion operation we also need to locate the bounded ring that contains the point, whose time complexity is $b_0 N_P + b_1 \log \frac{N_D}{N_P N_r}$. Then, we remove the point from the unordered point list of the bounded ring. Thus, the cost of the delete operation is $b_0 N_P + b_1 \log \frac{N_D}{N_P N_r} + b_2 N_r$.

For the split operation on BR_{ij}, we need to find the median of the distances between the pivot P_i and the points in Ω_{ij} to partition the Ω_{ij} into two subsets, which can be done in time $b_3 \Lambda$. Then, the new bounded ring is inserted into the ordered bounded ring list $List_i(BR)$, which takes time $b_4 \log \frac{N_D}{N_P N_r}$.

Thus, the time cost of split operation is $b_3\Lambda + b_4 \log \frac{N_D}{N_P N_r}$. In the merge operation, all points in the "old" bounded ring (i.e., the one with less points than λ) are appended to the point list of the new bounded ring, which takes time $b_5\lambda$. We also need to remove the "old" bounded ring from the pivot's bounded ring list, which takes time $b_6 \log \frac{N_D}{N_P N_r}$. So we get $Cost_{merge} = b_5\lambda + b_6 \log \frac{N_D}{N_P N_r}$.

Compared with existing high-dimensional kNN query indexes (e.g., tree-based structures, iDistance, VA-file, etc.), DBRI can be easily deployed in a distributed setting because there is no overlapping between the data points set in each bounded ring. Individual bounded rings can be maintained at different nodes in a cluster and the kNN query processing can be performed in parallel on certain bounded rings. As the number of points in each ring is always within a threshold, load balancing is achieved over the distributed cluster. The index information of a bounded ring only consists of the identifier, the upper bound and the lower bound. The data points in each bounded ring are stored in an unordered list in different slave nodes. Thus, the overhead in storage and maintenance in main memory on different nodes is small. Furthermore, since each bounded ring has a minimum occupancy of data points, it is possible to directly determine the bounded rings that can contain at least k neighbors for a given query, without invoking excessive iterations. It will make the distributed kNN processing highly efficient, as shown in the next section.

V. DHDKNN ALGORITHM FOR DISTRIBUTED kNN PROCESSING ON DBRI

In this section, we propose a distributed high-dimensional kNN algorithm (DHDKNN) on DBRI for efficient query processing.

A. DHDKNN ALGORITHM

DHDKNN follows a filter-and-refine scheme. For a given query q, we first find a candidate bounded ring set so that we can find at least k neighbors of q. Then, we search for the k nearest neighbors on the candidate set and the distance to the k-th nearest neighbor found so far is called *cdknn*. Using *cdknn* as a reference distance, we continue to identify the bounded rings that are guaranteed to contain the final kNNs points. Final kNN results are computed from these bounded rings. Now, we introduce the details of DHDKNN algorithm. Without loss of generality, we assume that the number of data points in every bounded ring is larger than λ.

1) CALCULATING CANDIDATE BOUNDED RINGS

The procedure is to calculate, in two steps, a set of candidate bounded rings that are guaranteed to contain at least k neighbors of a given query q. The pseudocode is shown in Algorithm 1.

- *Step 1:* We first determine the number of bounded rings needed to calculate a candidate set. Basically, we suppose that β ($\beta \ll \lambda$) points that are nearest to q are chosen from each selected bounded ring. We need to choose the number of bounded rings to be α such that $\alpha * \beta \geq k$. In this way, we can get at least k neighbors from these bounded rings. These k neighbors may not be the final results but we can use these neighbors to determine a search region that are guaranteed to contain the final results. Notice that it does not have to include all points in a bounded ring as we can already guarantee that we have found k neighbors of q by picking only β points from each bounded ring.
- *Step 2:* We then identify the candidate set of bounded rings C^{BR} which are the nearest bounded rings to the query q. Firstly, we compute the distance of q to all the pivots. Suppose that the radius of the pivot sphere is *MaxRadius* (i.e., the distance between P_i and the upper bound of the last bounded ring in $List_i(BR)$). If $dist(q, P_i) \leq MaxRadius_i$, indicating that q is within a bounded ring BR_{in} of the pivot, the distance between q and BR_{in} is set to 0, and we add it into C^{BR}. Then, we search for the left nearest bounded rings to q as follows. We build an upgrading ordered queue $UOR - List$ which is ranked by the distance. For the pivot P_i

Algorithm 1 CCBR Algorithm

Require: query q, pivots $P(P_1, \cdots, P_{N_P})$, DBRI index, α
Ensure: candidate bounded rings set C^{BR}

```
initial an upgrading ordered queue UOR − List with |UOR − List| = α.
for each pivot P_i do
    calculate dist(q, P_i).
    get the last bounded ring BR_iL in List_i(BR).
    if dist(q, P_i) ≥ ub_iL then
        add BR_iL into UOR − List.
    else
        find the bounded ring BR_iC that contains q from List_i(BR).
        add BR_iC into C^BR.
        get the adjacent bounded rings BR_{C−1} and BR_{C+1}
        calculate dist(q, BR_{C−1}) and dist(q, BR_{C+1})
        add BR_{C−1} and BR_{C+1} into UOR − List.
    end if
end for
while |C^BR| < α do
    add the first bounded ring BR_xF in UOR − List into C^BR and delete it from UOR − List.
    if there exist BR_{x(F−1)} and BR_{x(F+1)} in DBRI then
        if lb_xF > dist(q, P_x) then
            add BR_{x(F+1)} into UOR − List
        else if ub_xF < dist(q, P_x) then
            add BR_{x(F−1)} into UOR − List.
        end if
    end if
end while
return C^BR
```

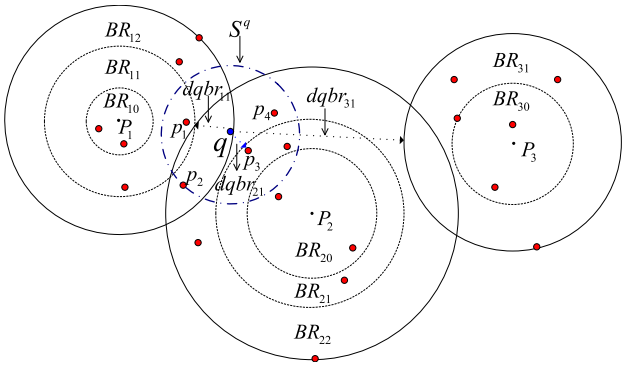

FIGURE 2. An example of searching for 3NNs of q ($\alpha = 3, \beta = 1$).

whose distance to q is larger than $MaxRadius$, we add the outermost bounded ring of P_i (i.e., the last bounded ring of list $List_i(BR)$) into $UOR-List$ with the distance being $dist(q, P_i) - MaxRadius$. For the pivot P_i which has a bounded ring BR_{in} containing q in its region, we add the adjacent bound rings of BR_{in} into $UOR - List$ with the distance being the value between q and the lower or upper boundary. Then, at every turn we pick the first one from $UOR - List$ (i.e., the nearest bounded ring to q) to C^{BR} and meanwhile add the adjacent bounded rings of the picked one into $UOR-List$ until we have α bounded rings in C^{BR}.

Take Figure 2 as an example where we assume $\alpha = 3$, $\beta = 1, k = 3$. First, BR_{12} and BR_{22} are added into C^{BR} because they contain q in their region. Then, we add their adjacent bounded rings BR_{11}, BR_{21} and the bounded ring of the other pivot BR_{31} into $UOR-List$. Obviously, the distance between q and BR_{21} ($dqbr_{21}$ in Figure 2) is the smallest. Thus, we add BR_{21} into C^{BR} and stop the procedure as $|C^{BR}| \geq \alpha = 3$. We get the final candidate bounded rings set $C^{BR} = \{BR_{12}, BR_{22}, BR_{21}\}$.

Algorithm 2 DHDKNN Algorithm

Require: query q, pivots $P(P_1, \cdots, P_{N_P})$, DBRI, α, β
Ensure: the kNN results set of q R_q

1: $C^{BR} = CCBR(q, P, \beta)$
2: find βNNs of q on each bounded ring in C^{BR} and put them into a set R'
3: compute $cdknn$ on R'
4: determine the search region S^q
5: find the bounded ring set R^{BR} which contains the bounded rings intersecting S^q
6: find kNNs of q from each bounded ring in R^{BR} and merge the results to get the final kNNs R_q
7: return R_q

2) DETERMINING THE FINAL SEARCH REGION

After we get the set of candidate bounded rings for the query q, we calculate β nearest neighbors from each candidate bounded ring and merge the results. As we have $\alpha * \beta \geq k$, we can get at least k nearest neighbors, which may not be the final kNN results. We calculate the distance to the k-th nearest neighbors of the candidate kNN results, which is marked as $cdknn$. Then, we can determine a region (sphere) S^q with q as the center and $cdknn$ as the radius, which is guaranteed to contain the kNNs of q. The final kNN results can be found after we scan all the bounded rings that intersect with S^q. As shown in the example of Figure 2, we first compute the 1NN from each bounded ring of the set $C^{BR} = \{BR_{12}, BR_{22}, BR_{21}\}$ and get the candidate 3NNs $\{p_2, p_3, p_4\}$. Then a search region S^q is determined as shown in the figure. Thus, the final bounded ring set that should be scanned is $\{BR_{11}, BR_{12}, BR_{20}, BR_{21}, BR_{22}\}$. We search all the points in the final bounded ring set whose distance to q is less than $cdknn$. The final kNN results can be obtained by maintaining a priority queue of points based on their distances to q. In the example, we get the final 3NN results of q as $\{p_1, p_3, p_4\}$.

3) RUNNING IN A MASTER-SLAVE SETTING

Figure 3 shows the steps to run DHDKNN algorithm in a master-slave setting. DBRI is maintained in a distributed manner by multiple slaves with each slave responsible for several bounded rings. The master is the entrance for the queries and the updated points, where we maintain the index information of DBRI: the pivots set P and the bounded rings lists with each bounded ring BR_{ij} taking the form $< i, id_j, lb_{ij}, ub_{ij} >$. In the master, we determine the candidate bounded ring set and send q to the shaded slaves that hold these bounded rings. The results are sent to Slave A to calculate the $cdknn$ value and the search region S^q. This is the first iteration as shown in Figure 3. Then, the bounded rings set R^{BR} which contains the bounded rings that intersect S^q are identified in Slave A and we send S^q to the slaves that hold the bounded rings in R^{BR} (i.e., Arrow 3 in Figure 3). In each of these slaves, we compute the k nearest neighbors of q and send the results to Slave B where the final kNNs of q are obtained. In this way, we can complete the kNN processing in a master-slave model in just two iterations.

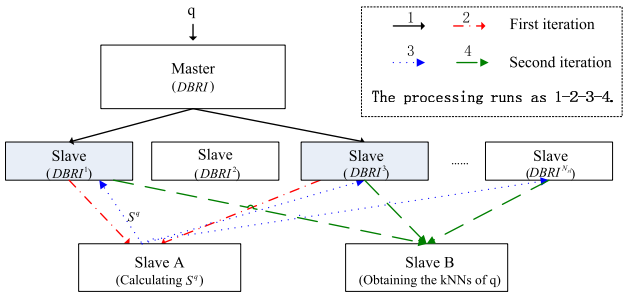

FIGURE 3. The processing in a master-slave setting.

According to the steps of running in a master-slave setting, we can deploy DHDKNN algorithm on some general-purpose distributed streaming data processing platforms, such as Apache Storm [33] and Yahoo S4 [34]. Take Apache Storm as an example. Apache Storm uses custom-created "spouts" and "bolts" to define information sources and manipulations.

A Storm application is designed as a "topology" in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. In our problem, we can define a spout as the entrance of the query source, which emits query points as a stream. DHDKNN algorithm is decomposed into several steps (i.e., arrows in Figure 3). Each step is handled by a particular type of bolts. These spout and bolt components are joint together orderly as a topology for DHDKNN algorithm.

B. ANALYSIS

Let N_D be the total number of points in the sliding window $|W|$. We use N_P and N_r to denote the number of pivots and the average number of points in each bounded ring of DBRI. Let N_{in} be the average number of bounded rings that intersect the search region for a query q. The running time of DHDKNN algorithm includes three parts: time to calculate candidate bounded rings $Cost_C$, time to calculate the search region $Cost_S$, and time to obtain kNN results $Cost_K$. Each part contains both communication and computational cost. To evaluate the communication cost, we assume g to be the ability of a network to deliver data. Let the number of slaves be N_{sl}. We can get the following theorem on the time cost of DHDKNN algorithm.

Theorem 2: The time cost of DHDKNN algorithm is $Cost_C + Cost_S + Cost_K$, while

$$Cost_C = c_0 N_P + c_1 \alpha \log \alpha + c_2 \log \frac{N_D}{N_P N_r} + g\alpha \quad (1)$$

$$Cost_S = (c_3 N_r + c_4 \beta \log N_r)\alpha / N_{sl} + c_5 k \log(\alpha\beta) + g\alpha / N_{sl} \quad (2)$$

$$Cost_K = (c_6 N_r + c_7 k \log N_r) N_{in} / N_{sl} + c_8 k \log(k N_{in}) + g(N_{in} + N_{in}/N_{sl}) \quad (3)$$

where $\{c_0, \cdots, c_8\}$ are some constants.

Proof: To calculate the candidate bounded rings set, we first compute the distance of the query to all the pivots which takes time $c_0 N_P$. Then, about α bounded rings are added into an upgrading ranking list to get the candidate bounded rings set, which costs time $c_1 \alpha \log \alpha + c_2 \alpha \log \frac{N_D}{N_P N_r}$. As there are α messages delivered to different slaves, the communication cost is $g\alpha$. So we get $Cost_C$ as shown in Equation (1).

The time to obtain β nearest points of q from each candidate bounded ring (including computing distances and selecting k smallest points by min heap) is $(c_3 N_r + c_4 \beta \log N_r)\alpha / N_{sl}$. Then, it takes time $c_5 k \log(\alpha\beta)$ to obtain the radius *cdknn* of the search region. As we have α messages from different nodes are sent to a node for the merge operation, the communication time can be evaluated as $g\alpha / N_{sl}$. Then we can get Equation (2).

We get the k nearest neighbors of q from each of these bounded rings, which takes $(c_6 N_r + c_7 k \log N_r) N_{in} / N_{sl}$. Then, it takes time $c_8 k \log(k N_{in})$ to obtain the final kNNs of q. The communication cost includes the time to send N_{in} messages to N_{sl} slaves and collect the results finally, so the communication cost is $g(N_{in} + N_{in}/N_{sl})$. We get $Cost_K$ as shown in Equation (3).

Notice that N_{in} is affected by two aspects. The first is the distribution of data points. Fortunately, the real-world data are often skewed, which can be well clustered and partitioned. Besides, N_{in} is also affected by the radius of the search region. The more bounded rings and points are considered in the step of calculating the candidate bounded rings, the higher probability to get a smaller *cdknn*. Thus, larger α and β decrease the value of N_{in}, but on the other hand, increase the computation cost in the step of calculating candidate bounded rings. The optimal value of α and β should be determined by the actual workload.

Compared to existing high-dimensional kNN processing method, the main advantage of DHDKNN algorithm is that we can complete the processing in only two iterations without storing the exact data points in the master node. It is highly beneficial for the kNN processing in a distributed environment as it helps to incur less communication cost and computation cost by searching only a small part of bounded rings in different slave nodes. In contrast, most existing algorithms do not have the property. With those algorithms, the master cannot determine the final region for kNN search without involving an uncertain number of rounds of communication between the master and slaves, incurring significant communication costs. Another advantage is that the DHDKNN algorithm is easily parallelizable and scales well with respect to the number of servers to handle the change of data size on the stream. The throughput of DHDKNN algorithm is roughly proportional to the number of servers.

VI. EXPERIMENTS

We conduct experiments on Apache Storm to validate the efficiency of the DBRI index and the DHDKNN algorithm.

A. EXPERIMENTAL SETTINGS

The experiments are conducted on a cluster of 8 Dell R210 servers with Gigabit Ethernet interconnect. The cluster runs Apache Storm platform. Each node has a 2.4GHz Intel processor and 8GB of RAM. We use a real dataset which comes from the 64-dimensional color histogram data from NUS-WIDE Image Data Set[1] [35], which contains 269,648 records from Flickr. To evaluate the performance on large datasets, we increase the size of this dataset to 1,000,000 using similar data expanding methods [36]. We assume that the size of the sliding window is approximately fixed, meaning the average number of appearances is asymptotically equivalent to the number of expirations. To mimic the real applications where the items appear in streams, we create a "streaming" version of the dataset. We feed the system a batch of 100 queries in one snapshot, and measure the time between the first query entering the system and the kNN results of all queries having been obtained. Some of the default values are shown in Table 2.

[1] http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm

TABLE 2. Default values.

parameter	default values
$\|W\|$	1,000,000
k	10
dimension	64
number of pivots	500
α	10
β	10
λ	20
Λ	150

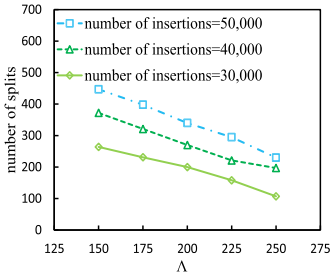

FIGURE 4. Number of splits w.r.t Λ.

B. BASELINES

As introduced in Section I, the state-of-art methods of distributed kNN processing can be categorized to two types by whether a global index is used in master node. A global index makes the query processing not necessarily happen on all nodes, but existing methods either (usually a tree-based-index [11]) require unpredictable communications across nodes in the query processing, or can only provide approximate results [12], which are difficult to be implemented on data streams to compete our methods. The other type of methods process kNN query in all nodes and finally merge the results, which are easily to be adapted to our problem. Thus, we compare our index and algorithm to the following methods on Apache Storm as the baselines:

1) *Naive solution (NS):* We simply partition the dataset to different nodes using a hashing function and search the k nearest neighbors for each query on each partition by direct distance computations, and then merge the kNN results from each partition to acquire the final kNNs.
2) *Distributed iDistance Solution (DIS):* We implement a distributed version of iDistance [24] on Apache Storm. We first find some reference points and assign all the points in the dataset to their nearest reference points. A B^+-tree is built to index the assigned points using the distance to the reference points as the key. Each sub-tree for a reference point is distributed stored in different slave nodes. When processing kNNs, we search the k nearest neighbors for each query on each sub-tree and finally merge all the kNNs to acquire the k nearest neighbors.
3) *Distributed Tree Solution (DTS) [10]:* It firstly partitions the dataset using a hashing function in the master node, and builds an M-tree index on the subset of each partition in different slave nodes. Similarly, we search the k nearest neighbors on the distributed partitions for each query and merge the kNNs to acquire the final kNN results. Without loss of generality, we can also build other existing centralized indexes (Δ-tree, VA-file, etc.) instead of M-tree to accelerate the kNN query processing.

C. PERFORMANCE OF INDEX MAINTENANCE

1) EFFECT OF Λ AND λ

From Figure 4 we can see that the split frequency is approximately reversely proportional to the value of maximum capacity Λ. As expected, a larger Λ results in a reduction in the number of splits. However, an overly large Λ will affect the average number of points in a bounded ring, which increases the time in the query processing. Figure 5 shows the influence of the minimum occupancy λ on the frequency of merge operations. A larger λ means that the underflow will occur more often and thus cause more merge operations. Furthermore, more updated points trigger higher split and merge frequency and thus cause greater maintenance cost.

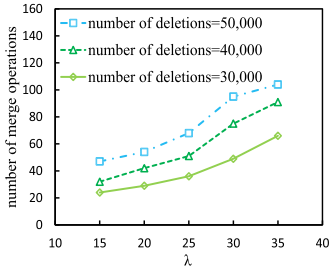

FIGURE 5. Number of merge operations w.r.t λ.

2) EFFECT OF THE NUMBER OF UPDATED POINTS

We then test the effect of the number of updated points (i.e., the time interval between two snapshots multiplying the velocity of updates) on the maintenance time and the result is shown in Figure 6. We can see that the maintenance time of all indexes increase when the number of updates becomes larger. The maintenance time is roughly linear with respect to the number of updated points since each update requires an independent maintenance operation for all indexes. Besides, we can see that the DBRI index incurs a bit more maintenance time than DIS and DTS as it involves extra split and merge operations. But on the other hand, DBRI incurs much less processing time than the other methods (Figure 9) as it can dynamically accommodate to the continuously changing data.

D. PERFORMANCE OF DHDKNN ALGORITHM

1) VARYING α AND β

Figure 7 and Figure 8 show the effect of α and β on the performance of DHDKNN algorithm. α and β are the parameters in calculating candidate bounded rings. As expected, larger α and β can help to settle a smaller *cdknn* value, i.e., larger

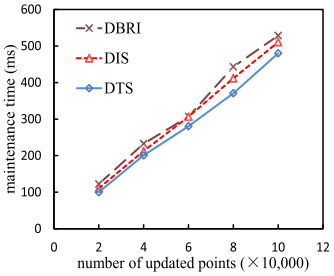

FIGURE 6. Comparison of maintenance time.

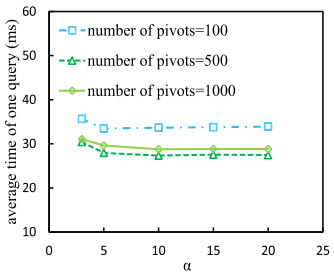

FIGURE 7. Performance w.r.t α.

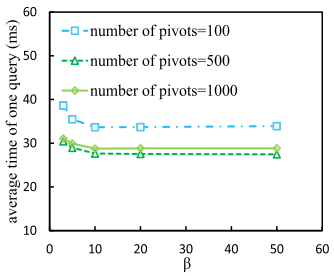

FIGURE 8. Performance w.r.t β.

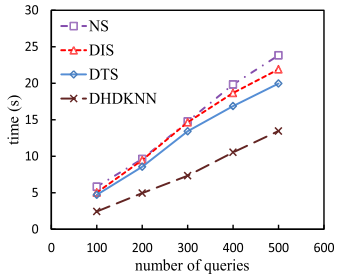

FIGURE 9. Performance w.r.t number of queries.

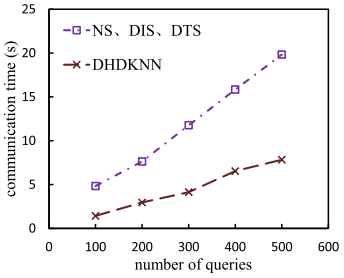

FIGURE 10. Communication time w.r.t number of queries.

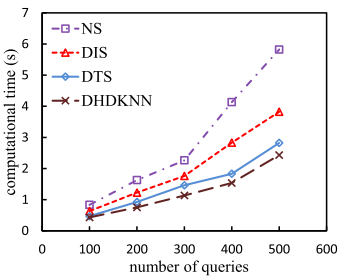

FIGURE 11. Computational time w.r.t number of queries.

α and β make *cdknn* closer to the final *dknn* value of the kNN results and thus less bounded rings intersecting with the search region are computed. Therefore, the average time of one query decreases when α and β get larger. However, when α and β is greater than 10, increasing α and β has little influence on the *cdknn* value. Thus, the curves go upward when the value is greater than 10 in both figures as the increment of computation cost in calculating candidate bounded rings becomes dominant.

Meanwhile, we can see that efficiency increases when the pivot number goes from 100 to 500, because more pivots lead to finer granularity of the data partition, which increases the pruning ability in the query processing. When the pivot number grows to more than about 500, the increment of pivot number has less and less effect, but the computation cost to the pivots incurs a little increment.

2) VARYING THE NUMBER OF QUERIES

We vary the number of queries and compare DHDKNN algorithm with baseline methods. From Figure 9 we can see that the time for all the methods increases almost linearly with the increasing number of queries as each query involves a single processing. DIS and DTS run faster than NS as the index in each partition accelerates search processing in the slave nodes. DHDKNN algorithm incurs the least processing time as we can reduce both computation cost and communication cost by pruning a large amount of bounded rings in the search procedure. Details of time cost are shown in Figure 10 and Figure 11. As can be observed from Figure 10, DHDKNN algorithm costs about 60% less communication time than baseline methods. NS, DIS, and DTS incur same communication time as each query is sent to all the slave nodes for kNN search in the three algorithms. Meanwhile, DHDKNN algorithm incurs the least CPU time as shown in Figure 11 because we only need to search for the kNN results in the small amount of bounded rings in the candidate set. Besides, experimental results show that the communication cost is almost one order of magnitude greater than the computing cost for processing one query, which verify our view that the communication cost between different nodes in a distributed cluster is very expensive compared with CPU cost.

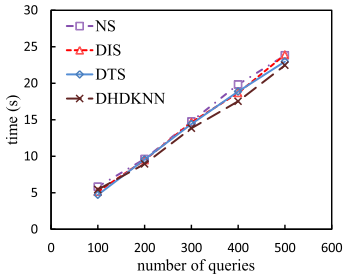

FIGURE 12. Performance on the uniformly distributed dataset.

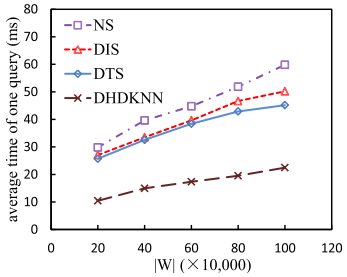

FIGURE 13. Performance w.r.t $|W|$.

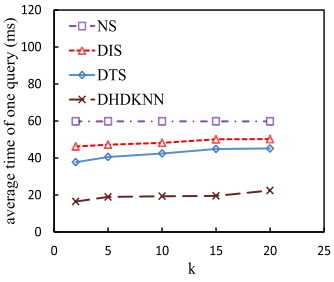

FIGURE 14. Performance w.r.t k.

We further generate a 64-dimensional synthetic dataset containing 1,000,000 points which follow a uniform distribution for comparison. From Figure 12 we can see that all the index structures have very close time cost to the NS method in a uniform dataset. The reason is that the data partitions formed by clustering technique in the index structures can be greatly overlapped with each other for the affection of ''dimensionality curse'' in high-dimensional space. Fortunately, data are often skewed in real world which can be well partitioned. Thus, DHDKNN algorithm is efficient to solve the problem in most cases.

3) VARYING $|W|$ AND k

As can be observed from Figure 13, the average time of one query for all the methods increases when the size of sliding window $|W|$ gets larger. The NS algorithm suffers the most from the increasing $|W|$ because it has to go through all the points in all slave nodes without an index. From Figure 14 we can see that the processing time of NS algorithm almost remains unchanged when k increases as it always involves distance computations to all points in the sliding window.

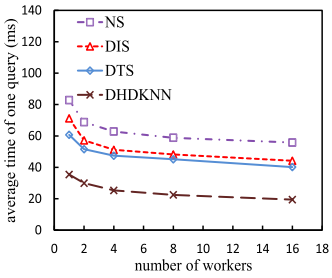

FIGURE 15. Scalability w.r.t number of workers.

DIS, DTS, and DHDKNN algorithm incurs more time as k becomes larger because more points need to be computed in the indexes with increasing k. The effect of k on DHDKNN algorithm is most obvious as both communication cost and computation cost increase when k increases.

4) VARYING THE NUMBER OF WORKERS

Finally, we vary the number of workers to measure the average response time per query and the result in shown in Figure 15. As can be observed, with more workers being employed, all algorithms enjoy a decrease in the processing time. The curves are non-linear because the communication cost increases with more workers. Clearly, DHDKNN algorithm has good scalability and performs much better than all the baseline methods.

VII. CONCLUSIONS

We present a real-time recommendation system that can make fast response to a user's current interest. If we represent both users' interests and items by high-dimensional points in vector space, a sensible and robust approach for content-based filtering is to recommend the user the k nearest neighbors (kNN) items, given a distance metric. Thus, the main issue tackled is to efficiently process kNN queries on data streams. Considering the vast volume of data, we aim to solve the problem in distributed settings to support our recommendation scheme. We propose a new distributed index called the Dynamic Bounded Rings Index (DBRI), which can be better adapted to the changing data points on data streams. DBRI can also be naturally distributed in the cluster to make the query processing more efficient. We present the distributed high-dimensional kNN query algorithm (DHDKNN) based on DBRI. DHDKNN algorithm involves only two iterations to prune a large amount of unnecessary bounded rings in distributed slave nodes, and thus incurs less communication and computational cost. Experimental results on Apache Storm demonstrate that our methods scale well and significantly outperform existing methods.

REFERENCES

[1] P. Brusilovsky, A. Kobsa, and W. Nejdl, *The Adaptive Web* (Lecture Notes in Computer Science), vol. 4321. Berlin, Germany: Springer, 2007, p. 325.
[2] M. Chen, X. Jia, E. Gorbonos, C. T. Hong, X. Yu, and Y. Liu, ''Eating healthier: Exploring nutrition information for healthier recipe recommendation,'' *Inf. Process. Manage.*, to be published.

[3] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. ICML*, 2014, pp. 1188–1196.
[4] M. Chen, X. Yu, and Y. Liu, "MPE: A mobility pattern embedding model for predicting next locations," in *World Wide Web*. Berlin, Germany: Springer, 2018, pp. 1–20.
[5] Y. Djenouri, A. Belhadi, J. C.-W. Lin, and A. Cano, "Adapted k-nearest neighbors for detecting anomalies on spatio–temporal traffic flow," *IEEE Access*, vol. 7, pp. 10015–10027, 2019.
[6] I. Kamel and C. Faloutsos, "Parallel R-trees," *ACM SIGMOD Rec.*, vol. 21, no. 2, pp. 195–204, 1992.
[7] W. Wu, J. Liu, H. Rong, H. Wang, and M. Xian, "Efficient k-nearest neighbor classification over semantically secure hybrid encrypted cloud database," *IEEE Access*, vol. 6, pp. 41771–41784, 2018.
[8] G. Wu, Z. Zhao, G. Fu, H. Wang, Y. Wang, Z. Wang, J. Hou, and L. Huang, "A fast kNN-based approach for time sensitive anomaly detection over data streams," in *Proc. Int. Conf. Comput. Sci.* Faro, Portugal: Springer, 2019, pp. 59–74.
[9] M. Yang, K. Ma, and X. Yu, "An efficient index structure for distributed k-nearest neighbours query processing," in *Soft Computing*. Berlin, Germany: Springer, 2018, pp. 1–12.
[10] B. Bryan, A. W. Moore, A. Snyder, and J. Schneider, *Using Distributed M-Trees for Answering K-Nearest Neighbor Queries*. Pittsburgh, PA, USA: Carnegie Mellon Univ., 2007.
[11] M. H. Ali, A. A. Saad, and M. A. Ismail, "The PN-tree: A parallel and distributed multidimensional index," *Distrib. Parallel Databases*, vol. 17, no. 2, pp. 111–133, 2005.
[12] H.-H. Choi and K.-C. Lee, "Performance enhancement of a DVA-tree by the independent vector approximation," *KIPS Trans., D*, vol. 19, no. 2, pp. 151–160, 2012.
[13] Y. Hu, C. Yang, C. Ji, Y. Xu, and X. Li, "Efficient snapshot KNN join processing for large data using mapreduce," in *Proc. ICPADS*, Dec. 2016, pp. 713–720.
[14] X. Ding, Y. Zhang, L. Chen, Y. Gao, and B. Zheng, "Distributed k-nearest neighbor queries in metric spaces," in *Proc. APWEB*. Macau, China: Springer, 2018, pp. 236–252.
[15] K. Chakrabarti and S. Mehrotra, "Local dimensionality reduction: A new approach to indexing high dimensional spaces," in *Proc. VLDB*, 2000, pp. 89–100.
[16] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu, "Scalable distributed processing of K nearest neighbor queries over moving objects," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 5, pp. 1383–1396, May 2015.
[17] P. Haghani, S. Michel, P. Cudré-Mauroux, and K. Aberer, "LSH at large-distributed knn search in high dimensions," in *Proc. WebDB*, 2008, pp. 1–6.
[18] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl, "Real-time top-n recommendation in social streams," in *Proc. RecSys*, 2012, pp. 59–66.
[19] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel, "StreamRec: A real-time recommender system," in *Proc. SIGMOD*, 2011, pp. 1243–1246.
[20] Z. Sun, N. Luo, and W. Kuang, "One real-time personalized recommendation systems based on slope one algorithm," in *Proc. FSKD*, 2011, pp. 1826–1830.
[21] C. Yang, X. Yu, and Y. Liu, "Continuous KNN join processing for real-time recommendation," in *Proc. ICDM*, 2014, pp. 640–649.
[22] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu, "Tencentrec: Real-time stream recommendation in practice," in *Proc. SIGMOD*, 2015, pp. 227–238.
[23] R. Weber, H. J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. VLDB*, vol. 98, 1998, pp. 194–205.
[24] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.
[25] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Contorting high dimensional data for efficient main memory KNN processing," in *Proc. SIGMOD*, 2003, pp. 479–490.
[26] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *Proc. EDBT*, 2012, pp. 38–49.
[27] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-NN queries on data streams," in *Proc. ICDE*, 2007, pp. 156–165.
[28] Y. Tao and D. Papadias, "Time-parameterized queries in spatio-temporal databases," in *Proc. SIGMOD*, 2002, pp. 334–345.
[29] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2. Sep. 1999, pp. 1150–1157.
[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. NIPS*, 2013, pp. 3111–3119.
[31] C. Musto, "Enhanced vector space models for content-based recommender systems," in *Proc. RecSys*, 2010, pp. 361–364.
[32] M. Ovsjanikov and Y. Chen, "Topic modeling for personalized recommendation of volatile items," in *Proc. PKDD*, 2010, pp. 483–498.
[33] *Twitter Storm*. Accessed: Sep. 24, 2013. [Online]. Available: https://github.com/nathanmarz/storm/
[34] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. ICDMW*, 2010, pp. 170–177.
[35] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y. Zheng, "NUS-WIDE: A real-world Web image database from National University of Singapore," in *Proc. ACM Int. Conf. Image Video Retr.*, 2009, p. 48.
[36] 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.

MIN YANG received the B.S. degree from the School of Software, from Shandong University, China, in 2011, where she is currently pursuing the Ph.D. degree in computer science and technology. Her research interest includes the area of data management.

YIXUAN ZUO received the M.S. degree in visual design from Chung-Ang University, South Korea, in 2015. She is currently a Lecturer with the School of Computer Science and Technology, Shandong Jianzhu University, China. Her research interest includes the area of multimedia data mining.

MENG CHEN received the Ph.D. degree in computer science and technology from Shandong University, China, in 2016. He was a Postdoctoral Fellow with the School of Information Technology, York University, Canada, from 2016 to 2018. He is currently an Assistant Professor with the School of Software, Shandong University. His research interests include the areas of data management and data mining.

XIAOHUI YU received the Ph.D. degree in computer science from the University of Toronto, Canada, in 2006. He is currently an Associate Professor with the School of Information Technology, York University, Toronto. His research interests include the areas of database systems and data mining.