This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

Scalable Distributed Processing of K Nearest Neighbor Queries over Moving Objects

Ziqiang Yu, Xiaohui Yu, *Member, IEEE,* Ken Q. Pu, *Member, IEEE,* Yang Liu, *Member, IEEE*

Abstract—Central to many applications involving moving objects is the task of processing k-nearest neighbor (k-NN) queries. Most of the existing approaches to this problem are designed for the centralized setting where query processing takes place on a single server; it is difficult, if not impossible, for them to scale to a distributed setting to handle the vast volume of data and concurrent queries that are increasingly common in those applications.

To address this problem, we propose a suite of solutions that can support scalable distributed processing of k-NN queries. We first present a new index structure called Dynamic Strip Index (DSI), which can better adapt to different data distributions than exiting grid indexes. Moreover, it can be naturally distributed across the cluster, therefore lending itself well to distributed processing. We further propose a distributed k-NN search (DKNN) algorithm based on DSI. DKNN avoids having an uncertain number of potentially expensive iterations, and is thus more efficient and more predictable than existing approaches. DSI and DKNN are implemented on Apache S4, an open-source platform for distributed stream processing. We perform extensive experiments to study the characteristics of DSI and DKNN, and compare them with three baseline methods. Experimental results show that our proposal scales well and significantly outperforms the alternative methods.

Index Terms—*k* nearest neighbor query, distributed query processing, moving objects

1 Introduction

Processing k nearest neighbor (k-NN) queries over moving objects is a fundamental operation in many location-based applications. For example, a location-based social networking service may help a user find k other users that are closest to him/her. In location-based advertising, a store may want to broadcast promotion messages only to the potential customers that are currently closest to the store. Such needs can be formulated as k-NN queries, where each user or customer can be considered as a moving object.

Consider a set of N_p moving objects in a two dimensional region of interest. An object o can be represented by a quadruple $\{id_o, t, (o_x, o_y), (o'_x, o'_y)\}$, where id_o is the identifier of the object, and t is the current time; (o_x, o_y) and (o'_x, o'_y) represent the current and previous positions of o respectively. Without making any assumptions on the motion of objects, we adopt the snapshot semantics in this study. That is, the answer to query q at time t is only valid for the past snapshot of the objects and q at time $t - \Delta t$, where Δt is the delay due to query processing. In order to provide answers as fast as possible (i.e., reducing Δt), we focus on main-memory-based solutions.

A major challenge for processing k-NN queries lies in the sheer volume of data and concurrent queries. A recent study [1] estimates that the global pool of generated personal location data was at least 1PB in 2009 and that it is growing by about 20% a year. The abundance of such data gives rise to a variety of location-based applications and services, which must be able to effectively handle a large quantity of user-initiated concurrent queries (many of which are k-NN queries) in addition to managing the vast volume of data. For example, the number of users of WeChat, a free social networking app installed on smart phones, has exceeded 300 million as of January 2013 [2]. One of its functionalities is to allow the users to locate their nearest fellow users upon request. In the new era of big data, it is imperative to find solutions that can effectively support the processing of many concurrent k-NN queries over large volumes of moving objects data.

Most existing solutions to the problem of k-NN over moving objects [3], [4], [5] are not designed to handle the volume of data in the Internet scale, because they implicitly assume a centralized setting, where the maintenance of the object locations and query processing both take place at a central place (i.e., a single server). While this is a reasonable assumption for a small set of objects and a light update/query workload, such approaches are no longer viable when the volume of data and/or queries exceeds the capacity of a single server. It is therefore necessary to develop distributed solutions that are able to scale as

- *Ziqiang Yu and Yang Liu are with the School of Computer Science and Technology, Shandong University, Jinan, Shandong, China, 250101. E-mail: zqy800@gmail.com, yliu@sdu.edu.cn.*
- *Xiaohui Yu is with the School of Computer Science and Technology, Shandong University, Jinan, Shandong, China, 250101, and the School of Information Technology, York University, Toronto, ON, Canada. E-mail: xyu@sdu.edu.cn.*
- *Ken Q. Pu is with the Faculty of Science, University of Ontario Institute of Technology, Oshawa, ON, Canada. E-mail: ken.pu@uoit.ca.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

demands increase.

To study the distributed processing of k-NN queries, we assume a general model of a distributed cluster that consists of a single *master* and multiple *slaves*. This model has been used in many well-known systems including recently proposed MapReduce [6] and Google File System [7]. Under this model, the following important factors must be considered when designing methods to process k-NN queries.

(1) *Communication cost*. Employing multiple servers to process k-NN queries in parallel may incur excessive data communication, which is very expensive compared with CPU cost. For example, in a typical setting of two dual-core 2.4GHz servers with Gigabit Ethernet interconnect, the time of transmitting the information of a set of objects between two nodes is two orders of magnitude greater than that of sorting the same set of objects (based on their distances to a certain point) in a single server. Thus, reducing the intra-cluster network communication becomes a critical aspect of algorithmic design.

(2) *Maintenance cost*. The movements of the large volume of objects may lead to frequent updates to the indexes used for processing k-NN queries. Therefore, the index structures must be designed in such a way that query processing can be supported efficiently and at the same time updates can be handled quickly.

(3) *Load balancing*. In reality, the moving objects and queries are usually non-uniformly distributed. It is therefore important to study how to store the objects and distribute the query load to different nodes in the cluster to achieve good load balancing.

Considering the aforementioned factors, we argue that neither tree-based indexes (such as R-tree[8], R*-tree[9], TPR*-tree[10] and Quad-tree[11]) nor grid indexes [3], [12], [4], the two main types of indexes utilized in the existing approaches to k-NN search, can be directly adopted in the distributed setting. Tree-based indexes with complex structures are expensive to maintain in the presence of frequent updates [13], [14], especially when deployed on a distributed cluster. On the other hand, grid-based approaches usually involve iteratively enlarging a search region to identify the set of cells in the grid that are guaranteed to include k-NNs. In general, the number of such iterations is uncertain. More importantly, in a distributed setting, such iterations can lead to excessive communication between the nodes in the cluster and thus degrade the performance of query processing. Therefore, there is an urgent need to develop an index structure that is suitable for being deployed in the distributed setting and performs well in terms of maintenance and communication cost.

To address this challenge, we propose a distributed strip index (DSI) that provides the basis for efficient distributed k-NN search. DSI partitions (without overlap) the region of interest along x and y dimensions respectively, resulting in a set of vertical and horizontal rectangular-shaped strips. The objects within each strip are indexed separately. Moreover, each strip of DSI has a maximum capacity and a minimum occupancy, both of which are configurable thresholds used to control the number of objects in each strip. As objects move, the strips can be split (or merged) when the number of objects in the strips goes beyond (or below) the maximum capacity (or the minimum occupancy).

Based on DSI, we further propose a distributed filter-and-refine algorithm, DKNN, that can support k-NN query processing with only two iterations. For a given query q, DKNN algorithm can use the DSI index to directly locate the candidate strips that are guaranteed to contain k neighbors of q. The algorithm then chooses a set of objects that are closest to q from each candidate strip and identify the k-th nearest neighbor in these selected objects. Using this neighbor as a reference point, it can determine a search region and compute the final k-NNs by calculating the distances between q and the objects in this region.

The DSI structure and the DKNN algorithm strike a good balance between the cost of index maintenance and that of query processing. By having a less complex index structure than the tree-based approaches, DKNN supports more efficient updates. Meanwhile, compared with the grid-based approaches that require an uncertain number of iterations in query processing, the DKNN algorithm is able to identify the final search space for each query with only two iterations, leading to great savings in communication cost as well as more predictable performance. Moreover, the DSI index is more flexible than the grid index. The strips can be dynamically merged or split as objects move to ensure that the number of objects in each strip is always within a given range. This, combined with the design of partitioning the region into strips along both vertical and horizontal directions, reduces the impact of data skewness on query processing. By keeping the number of objects in each strip within a given threshold, DSI distributes the objects more or less evenly to the strips, making it amenable to load balancing in the distributed setting.

We implement DSI and DKNN on top of Apache S4 [15], an open-source distributed stream processing platform, on which we conducted extensive experiments to evaluate the performance of our solutions.

Our main contributions can be summarized as follows.

- We propose DSI, a distributed strip index, for supporting k-NN search over moving objects in a distributed setting.
- We develop DKNN, a distributed k-NN search algorithm, which utilizes DSI to perform k-NN query processing. With only two iterations, this algorithm has a superior and more predictable performance than existing grid-based approaches.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

- We implemented DSI and DKNN on top of S4, and conducted extensive experiments to evaluate the performance of DSI and DKNN, which confirm its superiority over existing approaches.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 analyses the problems of existing indexes when they are adopted in a distributed cluster. Section 4 introduces the DSI index structure. Section 5 presents the DKNN algorithm. Section 6 discusses the implementation of DSI and DKNN on Apache S4. Experimental results are presented in Section 7. Section 8 concludes this paper.

2 Related Work

The problem of k-NN query processing over moving objects has attracted considerable attentions in recent years. In this section, we present a brief overview of the literature. The problems with the existing approaches will be discussed in the next section.

2.1 Tree-based approaches

The R-tree and its variants are widely used to process spatial queries. The R-tree has been adopted extensively (e.g., [16], [17], [18]) to answer nearest neighbor queries. Ni et al. [19], Roussopoulos et al. [20], and Chaudhuri et al. [21] use the TPR-tree to index moving objects and propose filter-and-refine algorithms to find the k-NNs. Gedik et al. [22] describe a motion-adaptive indexing scheme based on the R-tree index to decrease the cost of update in processing k-NN queries. Yu et al. [23] first partition the spatial data and define a reference point in each partition, and then index the distance of each object to the reference point employing the B^+-tree structure to support k-NN queries.

2.2 Grid-based approaches

The grid index is widely used to process spatial queries. It partitions the region of interest into equal-sized cells, and indexes objects and/or queries (in the case of continuous query answering) in each cell respectively [3], [4], [5], [24], [25], [26], [27]. Zheng et al. propose a grid-partition index for NN search in a wireless broadcast environment [4]. The Broadcast Grid Index (BGI) method proposed by [5] is suitable for both snapshot and continuous queries in a wireless broadcast environment. Šidlauskas et al. [26] propose PGrid, a main-memory index consisting of a grid index and a hash table to concurrently deal with updates and range queries. Wang et al. [27] present a dual-index, which utilizes an on-disk R-tree to store the network connectivities and an in-memory grid structure to maintain moving object position updates. Most of these approaches are designed for the centralized setting, and cannot be directly deployed on a distributed cluster, which will be discussed in the Section 3.

2.3 Scalable incremental approaches

The problem of scalable incremental processing of continuous queries has been studied in the literature [24], [25], [28], [29]. Mokbel et al. [24], Wu et al. [25] and Mokbel et al. [28] almost employ a shared execution paradigm on processing spatial queries to achieve better scalability. Nehme and Rudensteiner [29] group moving objects and queries based on common spatial-temporal properties to facilitate the scalability. These proposals reduce resource usage through sharing, but do not consider how to scale to multiple servers. Moreover, they mainly focus on the incremental processing of queries, but do not consider how to accommodate newly arrived queries.

2.4 Distributed processing of spatial queries

A number of existing works [30], [31], [32], [33] have explored distributed processing of spatial queries. All these proposals focus on utilizing the computational power at the mobile devices to obtain savings in terms of server load and communication cost. In the works [30], [31], the moving objects play an important role in processing queries. Wu et al. [32] propose a distributed strategy for k-NN search, in which the server and mobile devices collaborate to maintain the k-NNs of a moving query point. Bamba [33] processes spatial alarms based on the safe region technique, which enables resource-optimal distribution of partial tasks from the server to the mobile clients. These approaches all require the moving devices to have considerable computational capabilities, which restricts their applicability. Additionally, the unreliability of moving devices could seriously affect the performance of these approaches. In contrast, our approach does not assume any computation capabilities at the mobile objects other than reporting their positions (e.g., the mobile object can be a simple GPS tracking device), and thus has wider applicability.

Some recent works [34], [14], [35] use MapReduce to process k-NN joins. Zhang et al. [34] process parallel k-NN joins with two phases of map-reduce operations based on the R-tree index. Lu et al. [14] partition the sets of objects and queries based on the Voronoi diagram in the first Map function, and then find k-NNs of each query by the second Map and Reduce operators. Eldawy et al. [35] propose a framework called SpatialHadoop to support three kinds of spatial queries including k-NN queries. However, such methods are not suitable for real-time query processing as MapReduce is a batch-oriented framework [15], unless non-trivial modifications are made.

3 Problems with Existing Approaches

In this section, we analyze the existing tree-based and grid-based approaches and explain why they are not suitable for the distributed setting.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

3.1 Problems with tree-based approaches

When adopted in a *master-slaves* setting, the existing tree-based approaches suffer from high maintenance cost. The huge volumes of moving objects may incur intensive updates to the indexes (such as R-tree, k-d tree, B^+-tree, TPR-tree [10]), which are costly as they often involve frequent split and merge of nodes. This issue has been noted by a number of prior studies [13], [14]. In the distributed setting, the problem becomes even more severe as the nodes are likely to be distributed to different servers, leading to frequent interactions between servers.

3.2 Problems with MapReduce-based approaches

While the MapReduce-based approach proposed in [14] provides a scalable solution to the k-NN search problem, it cannot be directly applied to the problem of k-NN search over *moving objects*, as it suffers from large preprocessing and update costs. For one, this approach needs to select multiple pivots, and all objects and queries need to be partitioned beforehand based on the Voronoi diagram, which is an expensive operation because it requires the calculation of the pair-wise distances between each object and each pivot. For another, the partitioning has to be constantly updated as the objects move, which incurs inhibitive costs when the number of objects is large.

SpatialHadoop [35], a MapReduce framework that can support k-NN spatial queries, suffers from some critical problems when applied to the processing of continuous k-NN queries: (1) Since it is not specifically designed for moving objects, SpatialHadoop does not consider the maintenance cost explicitly, and the index may not work well in the presence of frequent position updates. For our problem, the maintenance cost is a major concern as the objects are constantly moving; (2) SpatialHadoop is not well suited to the continuous processing of queries over moving objects, because the MapReduce paradigm employed by SpatialHadoop is a batching-oriented processing paradigm and is not good at handling the incremental changes to the query results caused by numerous small updates.

3.3 Problems with grid-based approaches

Most existing grid-based approaches to k-NN search [3], [12] iteratively enlarge the search region to identify the k-NNs. For example, given a new query q, the YPK-CNN algorithm [3] initially locates a rectangle R_0 centered at the cell covering q; it then iteratively enlarges R_0 until it encloses at least k objects. Let p' be the farthest object to query q in R_0. The circle C_r centered at q with radius $\| q - p' \|$ is guaranteed to contain the k-NNs of q, where $\| \cdot \|$ is the Euclidean norm. The algorithms then computes the k-NNs using the objects in the cells intersected with C_r. The other existing grid-based approaches are based on similar ideas.

When adopted in a *master-slaves* setting, such iterative procedures may cause severe performance degradation. Consider a reasonable adaptation that lets the master maintain the partitioning information (e.g., cell boundaries) and the slaves maintain the indexes for individual cells. In this case, each iteration requires one round of communication between the master and the slaves holding the relevant cells (i.e., transmitting the distances between the query and the objects in each relevant cell), which is an expensive operation compared with other costs (e.g., the CPU cost of calculating distances). To make things worse, there is no guarantee on the number of iterations required, making the query performance unpredictable.

The main reason causing the uncertainty in the number of iterations required is that the master cannot single-handedly determine which cells should be involved in processing a given query. To solve this problem, one might attempt to augment the information kept in the master so that it can directly distribute the queries to the subset of slaves involved without iterations. The extreme solution would be to let the master store the locations of all objects; however, the master will then become a bottleneck and it defeats the whole purpose of distributed computation.

A "milder" solution would be to make the master aware of some aggregate information about each cell. But it is also problematic. To illustrate this, we use the number of objects in each cell, which is probably the most lightweight of such aggregate information, and consider the following two cases.

Case 1: Each slave keeps the master updated about the numbers of objects contained in each cell maintained by that slave. With the objects frequently moving, the update information would cause excessive communication cost.

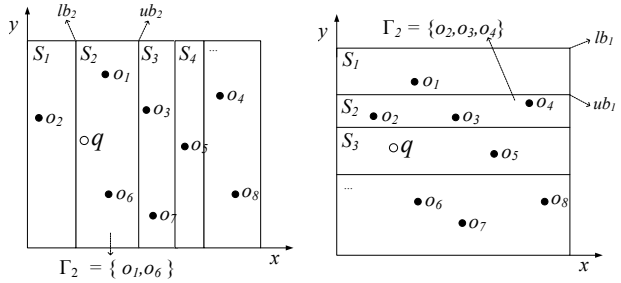
Case 2: The master maintains an index on the number of objects in each cell. However, updating this index also involves considerable cost as the frequent movement of the objects may result in constant changes in the number of objects in each cell. This task puts a heavy burden on the master, which must also take care of distributing the queries to cells, making it prone to become a performance bottleneck.

In summary, the existing grid-based approaches to k-NN query processing cannot be applied to the distributed setting without non-trivial extensions.

4 Dynamic Strip Index

We have analyzed the pitfalls of tree-based indexes and grid-based indexes when applied to a distributed setting. We argue that a distributed index for k-NN search over moving objects must have the following properties: (1) The index can be easily partitioned and distributed to different servers in a cluster; (2) It can be updated efficiently as objects move continuously; (3) It can support efficient k-NN search algorithms

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering



(a) Vertical strips (b) Horizontal strips

Fig. 1. The elements of the DSI index structure

that involve few iterations. To meet these requirements, we propose the Dynamic Strip Index (DSI), a main-memory index structure to support distributed k-NN search.

4.1 Structure of DSI

In building DSI, the region of interest R in an Euclidean space (normalized to the $[0, 1)$ square) is partitioned into non-overlapping strips. The partition is done separately along both the x and y axes (See Fig. 1). Therefore the result of the partition is two sets of strips: vertical strips and horizontal strips. In what follows, our description will be based on vertical strips; the characteristics of the horizontal strips are exactly the same except for the orientation.

A vertical strip S_i ($1 \leq i \leq N_v$, where N_v is the number of vertical strips) in the index takes the form of $\{id_i, lb_i, ub_i, \Gamma_i\}$, where id_i is the unique identifier of S_i, lb_i and ub_i are the lower and upper boundaries of the strip respectively, and Γ_i is an unordered list of objects that fall in S_i. That is, $\forall o \in \Gamma_i, lb_i \leq o_x < ub_i$. The strips are non-overlapping and every object must fall in one strip. That is, $\Gamma_1 \cup \cdots \cup \Gamma_n = \mathcal{O}$, where $\mathcal{O}$ is the set of all indexed objects, and $\Gamma_i \cap \Gamma_j = \phi$ for any pair of strips S_i and S_j ($i \neq j$). Note that DSI is not a space partitioning index but a data partitioning index, which means that the boundaries of a strip are not simply equally spaced; rather, they are determined dependent on the data.

We require every strip to contain at least ξ and at most θ objects, i.e., $\xi \leq |\Gamma_i| \leq \theta$ for all strips S_i. The strips are split or merged as needed to ensure this condition is met when object locations are updated. We call ξ and θ the *minimum occupancy* and *maximum capacity* of a strip respectively. Typically $\xi << \theta$. In the rare case where the total number of objects N_p is less than ξ, the minimum occupancy requirement cannot be satisfied. This is handled as a special case in query processing. To simplify our discussion, we assume without loss of generality that at any time the total number of objects $N_p \geq \xi$.

We use a list L_V to store information related to the vertical index, where each element corresponds to a strip. All elements in this list are sorted in ascending order according to their boundaries.

4.2 Insertion

An object is inserted into vertical strips based on its x coordinate (y coordinate for the horizontal strips), i.e., an object o is inserted into strip S_i if o falls into its boundaries ($lb_i \leq o_x < ub_i$). The insertion is done by appending its id_o into the object list Γ_i. Initially, there is only one strip covering the whole region of interest.

When an object o is inserted into a strip S_i, S_i will be split if the number of objects in it exceeds the maximum capacity, i.e., $|\Gamma_i| > \theta$. A split method that can adapt to the data distribution is to split S_i and generate two new ones that hold approximately the same number of objects. In this method, we first find an object o such that o_x is the median of the x coordinates of all objects in strip S_i, which implies that there are approximately $|\Gamma_i|/2$ objects whose x coordinates are less than or equal to o_x. This can be accomplished in $O(|\Gamma_i|)$ time using the binning algorithm. Next, we set the line $x = o_x$ as the split-line, according to which S_i can be split into two new strips S_{i1} (to the left) and S_{i2} (to the right).

Once S_i is split, the attributes of new strips need to be determined. As shown in Fig. 2, the lower boundary of S_{i1} is the same as that of S_i, and its upper boundary is the split-line. For S_{i2}, it uses the split-line as the lower boundary, and the upper boundary of S_i as its upper boundary. The id of S_i is inherited by S_{i1}, and a new id is assigned to S_{i2}.

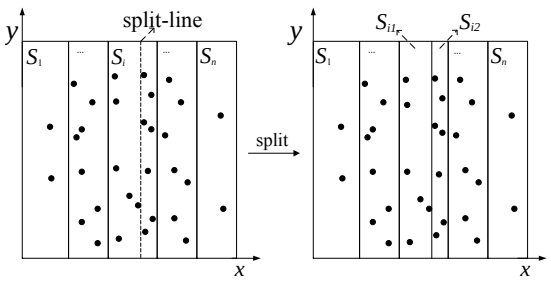


Fig. 2. The split of S_i

4.3 Deletion

When an object disappears or moves out of a strip, it has to be deleted from the strip that currently holds it. We assume that an object o about to disappear will notify this event to the server by presenting itself as $\{id, t, (-1, -1), (o'_x, o'_y)\}$, where the coordinates $(-1, -1)$ signify that the object will disappear from the region. To delete an object o, we need to determine which strip current holds it, which can be done using its previous position (o'_x, o'_y).

If after deleting an object, the strip S_i has less than ξ objects (i.e., S_i has an underflow), it will be merged with an adjacent strip. If S_i has a strip to its left and another to its right, then S_i is to be merged with the one with fewer objects. Ties are broken at random.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

Let this adjacent strip be S_j. S_i will be deleted from the index, and the merged strip will inherit the id of the S_j, and its lower and upper boundaries are set to be the lower and upper boundaries of S_i and S_j, respectively. The object lists Γ_i and Γ_j are merged. In case $|\Gamma_j| + |\Gamma_i| \geq \theta$, the number of objects in the resulting strip exceeds the threshold θ, triggering another split. However, since in general $\xi << \theta$, such situations rarely happen and their impact on the overall performance is minimal.

4.4 Analysis of the DSI structure

4.4.1 Time cost of maintaining DSI

Theorem 1: Let N_P and N_s be the number of objects and strips respectively, and assume that the objects are uniformly distributed. Then, $T_{insert} \approx a_0 \log N_s$, $T_{delete} \approx a_1 \log N_s + a_2 \frac{N_p}{N_s}$, $T_{split} \approx a_3\theta + a_4 \log N_s$ and $T_{merge} \approx a_5\xi + a_6 \log N_s$, where T_{insert}, T_{delete}, T_{split}, and T_{merge} are the time costs of the insert, delete, split, and merge operations respectively, and $a_i (i = 1, \ldots, 6)$ are constants.

Proof: From the definitions of θ and ξ, we have $\frac{N_p}{\theta} \leq N_s \leq \frac{N_p}{\xi}$. Inserting an object o involves finding the right strip and appending it to the end of its object list; therefore, $T_{insert} \approx a_0 \log N_s$. To delete an object from a strip, we need to first identify the strip and then remove the object from its object list. The costs of these two operations are $a_1 \log N_s$ and $a_2 \frac{N_p}{N_s}$ respectively. For a split operation, we need to first identify the object whose position is the median of all the object positions in the strip, which takes linear time with respect to the number of objects: $a_3\theta$. On top of this, we need to insert this new strip into the index, which takes time $a_4 \log N_s$. Thus, $T_{split} \approx a_3\theta + a_4 \log N_s$. For a merge operation, we need to append the object list of the old strip to that of the new strip, which takes time proportional to the number of objects in the old strip, i.e., $a_5\xi$, and we also need to remove it from L_V (or L_H), which takes time $a_6 \log N_s$. □

4.4.2 Advantages of DSI

DSI forms the basis of our approach to distributed k-NN processing, and has the following advantages.

- **Parallelizable**: DSI's partitioning strategy makes it easy to be deployed in a distributed system: individual strips can be maintained at different nodes in a cluster. The strips do not overlap, making it possible to perform query processing in parallel.
- **Scalable**: Each server can handle a certain number of strips of DSI. Since the number of objects in each strip falls into a given range, the capacity of DSI is directly proportional to the number of servers, lending it well to large-scale data processing.
- **Skew-resistant**: The use of both a vertical and a horizontal strip gives the index the ability to handle data skew along x and y directions. As an example, consider cases where all objects have very close y coordinates but are more or less uniformly distributed along the x axis. In this scenario, the horizontal index does not provide an effective partitioning of the data. Instead, the vertical index would provide a nice partitioning.
- **Light-weight** The index has a small storage overhead. Besides a list of objects, we only need to store the id and two boundaries for each strip, this is a nice feature for an in-memory index.
- **Efficient** Having a minimum occupancy for each strip makes it possible to directly determine the strips that can contain at least k neighbors of a given query, without invoking excessive iterations. This advantage will become more clear in next section.

TABLE 1
Summary of notations

Notation	Meaning
$\mathcal{C}^V$	the set of vertical candidate strips
$\mathcal{C}^H$	the set of horizontal candidate strips
$\mathcal{C}^T$	the set of all candidate strips
Υ	the set of supporting objects
$\mathcal{F}$	the strips that intersect with C_q

5 A DISTRIBUTED k-NN SEARCH ALGORITHM

We propose a distributed k-NN search algorithm (DKNN algorithm) that uses DSI for processing k-NN queries. The notions used in this algorithm are summarized in Table 1.

5.1 The DKNN algorithm

The DKNN algorithm follows a filter-and-refine paradigm. For a given k-NN query q, the algorithm first prunes the search space by identifying the strips that are guaranteed to contain at least k neighbors of q. It then examines the objects contained in this set of strips and identify the k-th nearest neighbor found so far. Using the position of this neighbor as a reference point, we continue to identify the strips that can potentially contain objects that are closer to q than this reference point, and obtain the final result. The algorithm is presented in Algorithm 1. Now we present the details of the algorithm. Without loss of generality, we assume that $N_p \geq k$ where N_p is the number of objects.

5.1.1 Calculating candidate strips

For a given query q, DKNN can directly identify the set of strips that are guaranteed to contain k neighbors of q, which we call the candidate strips. Here, we take determining vertical candidate strips for instance.

Step 1: Calculating the number of candidate strips. Assume that the number of candidate strips is c. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

idea is that from each strip we select χ ($1 \leq \chi \leq \xi$) objects that have the shortest Euclidean distances to q, such that $\chi * c \geq k$, where χ can be specified by users. This way, we have found at least k neighbors for q. Of course, these objects may not be the final k-NNs, but they can help us prune the search space and serve as the starting points for computing the final k-NNs. Hence, the number of candidate strips c is set to be $\lceil k/\chi \rceil$. In the rare case where $c > |L_V|$, c is set to $|L_V|$, i.e., all vertical strips will be considered as candidate strips.

One might wonder why we not choose to include all objects in a candidate strip in this step but to include only χ objects from each strip. The reason is that by picking only χ objects from each strip, we can already guarantee that we have found k neighbors of q. Choosing only the closest χ objects to q from each candidate strip helps narrow down the searching space as small as possible for the next step.

Step 2: Identifying the set of candidate strips. In this step, we identify the strips that are considered to be "closest" to q based on their boundaries. We use d_i^l and d_i^u to denote the distances from q to the lower and upper boundaries of S_i respectively. For the example shown in Fig. 3, the line l_i is perpendicular to lb_2 of S_2 and the distance from q to lb_2 is d_2^l. The distance between S_i and q is defined to be $dist(S_i, q) = \max\{d_i^l, d_i^u\}$.

If query q is located in S_i, then S_i is automatically a candidate strip and inserted into $\mathcal{C}^V$. Next, we decide whether its neighboring strips are candidate strips. Starting from the immediately adjacent strips, we expand the scope of search, adding to $\mathcal{C}^V$ the strip j that has the next least $dist(S_j, q)$. This procedure terminates when $|\mathcal{C}^V| = c$ or all the strips have been processed. Fig. 3 gives an example, in which S_3 is determined to be a candidate strip first. Then by comparing d_2^l with d_4^u, we decide S_4 to be the next candidate strip. Next, we find S_2 also a candidate strip. The algorithm of determining the vertical candidate strips is shown in Algorithm 2. The horizontal candidate strips can be computed in the same way.

This procedure describes the details of *DCS* algorithm and it can be implemented on the *master-slaves* setting as shown in 5(a). The DSI index is maintained in a distributed fashion by multiple slaves, where each slave is responsible for a set of strips. The master is the entry point for the queries and object updates. It maintains L_V and L_H, the lists of strip ids and their boundaries. When the master receives a query q, it can immediately determine the candidate strips by running the *DCS* algorithm, and then send q to the shaded slaves that hold the candidate strips.

5.1.2 Determining the final search region

After the candidate strips are determined, we form the set of *supporting objects* Υ by selecting from each candidate strip χ objects that are closest to q. We then identify the supporting object $o \in \Upsilon$ that is the k-th closest to q. Let the distance between o and q be r_q. The circle with (q_x, q_y) as the center and r_q as the radius is thus guaranteed to cover the k-NNs of q. Next, we identify the set of strips $\mathcal{F}$ (including both vertical and horizontal strips) that intersect with this circle, and search foFig. 4 shows an example, where the query q is a 3-NN query and $\chi = 1$. We find three closest supporting objects (o_3, o_2, o_4) in its candidate strips $\{S_2, S_3, S_4\}$ and set the radius r_q to be the distance between q and o_3. The circle C_q is guaranteed to contain the 3-NNs of q. After scanning all objects that are located within C_q (by examining all strips intersecting C_q), we find that the 3-NNs are o_1, o_2, and o_4. r the final k-NNs within the objects in $\mathcal{F}$. Since some of the strips in $\mathcal{F}$ are also in $\mathcal{C}^T = \mathcal{C}^V \cup \mathcal{C}^H$ and therefore the distances between the objects in $\mathcal{C}^T$ and q are already computed, we do not need to repeat the computation. We only have to compute the distances between q and the objects in the set $\mathcal{F}$ - $\mathcal{C}^T$. The set of k-NNs can be obtained by maintaining a priority queue of objects based on their distances to q.

Note that the reason of maintaining two strip lists (both horizontal and vertical) instead of one becomes even more clear with the description of DKNN. This strategy is able to better handle the skewness in data

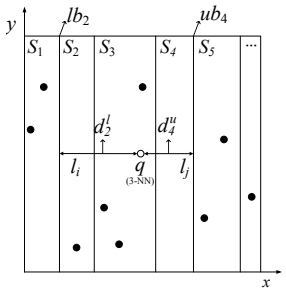


Fig. 3. Determining the set of candidate strips

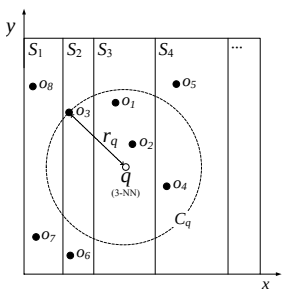
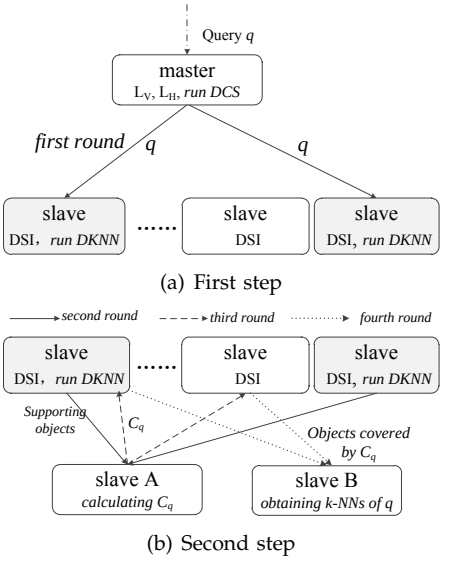


Fig. 4. An example of finding 3-NN using DKNN



(a) First step

(b) Second step

Fig. 5. Processing queries on the *master-salves* model

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

Algorithm 1 *DKNN* Algorithm

Input:
The query q (q_x, q_y); vertical strips L_V; horizontal strips L_H; the parameter χ
Output: :
k nearest neighbors of q.
1: Set both the number of vertical and horizontal candidate strips to c;
2: $\mathcal{C}^V$=*DCS*(q_x, q_y, L_V, c);
3: Compute $\mathcal{C}^H$ in the same way;
4: Set $\mathcal{C}^T$ =$\mathcal{C}^V \cup \mathcal{C}^H$;
5: **if** $c \leq |\mathcal{C}^T| \leq 2c$ **then**
6: Find χ supporting objects in every candidate strip, and put them into Υ;
7: Compute the distances from the supporting objects in Υ to q;
8: Let o be the k-th nearest neighbor of q ($\exists S_i \in \mathcal{C}^T$, s.t. $o \in \Gamma_i$)
9: Set r_q = distance(o, q);
10: Determine circle C_q centered at q with radius r_q;
11: Let $\mathcal{F} = \{S | S \in L_V \cup L_H, S \text{ intersects } C_q\}$;
12: Find k-NNs from the objects covered by strips in $\mathcal{F}$;
13: **end if**
14: **if** $0 < |\mathcal{C}^T| < c$ **then**
15: Search all strips in L_V or L_H to obtain k-NNs;
16: **end if**
17: Return k-NNs;

distribution. Very often, it can help to determine a smaller circle C_q than what a single index can do. Fig. 6 shows an example. We assume q is a 3-NN query and $\chi = 1$. If there are only horizontal strips, we have to select $\{o_1, o_2, o_4\}$ to determine the circle C_1 as the region for the final k-NN search. However, if vertical strips exist, we can determine the circle C_2 according to objects $\{o_1, o_2, o_3\}$. C_2 is obvious smaller than C_1, and can therefore lead to less strips having to be examined.

Fig. 5(b) shows this step running in the *master-slaves* setting. The shaded slaves that maintain the candidate strips choose χ supporting objects of query q from each of their respective candidate strips, and send these supporting objects to the slave A. This slave can then compute the circle C_q and identify the final set of strips $\mathcal{F}$ intersected by C_q that are guaranteed to contain the k-NNs. It sends C_q to the slaves holding the strips in $\mathcal{F}$. Finally, k nearest neighbors are chosen from each strip in $\mathcal{F}$ (or all the objects in the strip if it contains less than k objects) and sent to slave B, where the final k-NNs are computed.

5.2 Analysis of the DKNN algorithm

5.2.1 The correctness of DKNN

Theorem 2: For a given query q, the algorithm is guaranteed to find its k nearest neighbors, if every strip contains at least ξ objects.

Proof: Assume that χ ($\chi \leq \xi$) objects are selected from each candidate strip as supporting objects. If we choose $\lceil k/\chi \rceil$ vertical candidate strips, then $\mathcal{C}^V$ must contain $\lceil k/\chi \rceil \cdot \chi \geq k$ neighbors of q. Since the center of circle C_q is the point (q_x, q_y) and the radius is the distance from the k-th closest supporting object to q, C_q contains at least k neighbors. We obtain the final k-NNs from the objects that are contained in the strips intersecting C_q. Assume that there exists an object $o \in S_i$ where S_i does not intersect C_q, and $o \in k$-NN of q, then we must have $dist(o, q) < r_q$. Contradiction. Therefore, the DKNN algorithm is correct. □

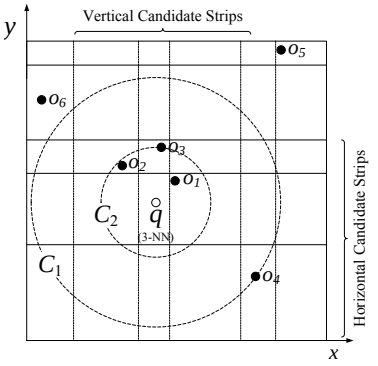


Fig. 6. Benefits of having both vertical and horizontal indexes

5.2.2 Time cost of the DKNN algorithm

Theorem 3: Let N_p and N_s be the number of objects and strips respectively. For a given k-NN query q, the query processing time (barring the communication cost) by DKNN is $T_{query} = T_d + T_c + T_l$, where $T_d \approx a_0 \log N_s + a_1 \lceil k/\chi \rceil$, $T_c \approx a_2 \lceil k/\chi \rceil \cdot N_p/N_s \cdot \chi + a_3 k \log k$, $T_l \approx a_4 N_p \sqrt{k/(\pi N_p)} \log k$, and $a_i (i = 1, \ldots, 4)$ are constants.

Proof: Let T_d be the time of determining the candidate strips, T_c be the time of obtaining the circle C_q, and T_l be the time of searching k-NNs from $\mathcal{F}$ (the set of strips covered by C_q). The time of finding the strip covering q is $a_0 \log N_s$, and the time of finding the remaining candidate strips is $a_1 \lceil k/\chi \rceil$; therefore, $T_d \approx a_0 \log N_s + a_1 \lceil k/\chi \rceil$. To compute the circle C_q, we need time $a_2 \lceil k/\chi \rceil \cdot N_p/N_s \cdot \chi$ to find the χ closest objects (to q) from each candidate strip. Obtaining the radius of the circle C_q then takes time $a_3 k \log k$. Therefore, $T_c \approx a_2 \lceil k/\chi \rceil \frac{N_p}{N_s} \chi + a_3 k \log k$. Finally, as we assume a uniform distribution of the data, the expected area of C_q is k/N_p. Thus, the radius $r_q = \sqrt{k/(\pi N_p)}$. Then, C_q covers approximately $2\lceil N_s \sqrt{k/(\pi N_p)} \rceil$ vertical strips. Therefore, the time of obtaining the k-NNs $T_l \approx a_4 N_p \sqrt{k/(\pi N_p)} \log k$. □

5.2.3 Effects of ξ and θ

The minimum occupancy ξ influences the frequency of the merge operation. To simplify the analysis, we assume that the N_p objects are uniformly distributed in a unit square, and the width of any strip s_i is w_s. Then for a given object, the probability that it falls into s_i is w_s. Thus, the probability of ξ objects falling into s_i is $w_s^\xi (1 - w_s)^{N_p - \xi}$. Therefore, the probability that s_i contains less than ξ objects (and thus has to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

be merged) is $\sum_{i=1}^{\xi-1} w_s^i(1-w_s)^{N_p-i}$. We can infer that more merges will happen when ξ increases. On the other hand, there will be a negative impact on the query performance if ξ is too small. Recall that in DKNN, the number of candidate strips for a k-NN query is $\lceil k/\chi \rceil$ ($\chi \leq \xi$), which means that excessive strips have to be processed as candidate strips if ξ is too small.

The maximum occupancy θ affects the splitting of strips. The probability that s_i contains more than θ objects is 1-$\sum_{i=1}^{\theta} w_s^i(1-w_s)^{N_p-i}$, which indicates that more strips will be split when θ decreases. Meanwhile, an overly large θ will also affect the cost of query processing. This is because for any candidate strip, the distances between the objects contained therein and query points need to be calculated, and a larger θ will lead to more objects in each strip.

5.2.4 Advantages of DKNN

The most notable advantage of DKNN is that even though the master node does not store the positions of objects, it can still determine the search space that contains the k-NNs in just two steps, by first directly determining the candidate strips using the *DCS* algorithm, and then identifying the final set of strips to search by computing the circle C_q. This is highly beneficial when the algorithm is running in a distributed system.

In contrast, most existing search algorithms do not have this property. With those algorithms, the master cannot determine the final region for k-NN search without involving an uncertain number of rounds of communication between the master and slaves, incurring significant communication costs.

To be more specific, we take YPK-CNN as an example and compare it with DKNN in terms of computation time. As explained in Section 1, the communication cost is the dominating factor. Therefore, we only consider the communication cost in the following analysis. DKNN can process an arbitrary query q with only four rounds of communication, which is shown as Fig. 5. The first round involves the master sending the query q to the slaves that hold the candidate strips. In the the second round the slaves receiving the query q send to a slave A the supporting objects of q selected from their respective candidate strips. In the third round, slave A sends C_q, the final search region, to the slaves that hold the strips in $\mathcal{F}$ that are intersected with C_q. In the last round, k neighbors are chosen from each strip in $\mathcal{F}$ (or all the objects in the strip if it contains less than k objects) and sent by the salves maintaining them to another slave B, where the final k-NNs can be obtained.

In contrast, YPK-CNN algorithm requires an uncertain number of iterations of enlarging the search region to process a newly arrived query q. In the best case that the initial rectangle R_0 already contains k neighbors of q, the number of iterations can be equal to 1. But this happens only when the data is uniformly distributed and the cell size is perfectly optimized, which is rare in practice. In the worst case, the number of iterations can reach $(1+max\{\lceil(q_x-l/2)/\delta\rceil, \lceil(1-q_x-l/2)/\delta\rceil, \lceil(q_y-l/2/\delta)\rceil, \lceil(1-q_y-l/2)/\delta\rceil\})$, where l is the size of R_0 and δ is the step size for the enlargement of R_0 in each iteration. When l and δ take the typical values of 0.02 and 0.01 respectively and the location of q is (0.3, 0.4), the number of iterations is 70, which is much larger than that of DKNN.

Algorithm 2 *DCS* Algorithm

Input:
The query q (q_x, q_y); L_V; c.
Output: :
vertical candidate strips $\mathcal{C}^V$.
1: $\mathcal{C}^V = \phi$
2: Sort L_V according to the low boundaries of strips;
3: **if** strip S_i satisfies $lb_i \leq q_x < ub_i$ **then**
4: Insert S_i into the set $\mathcal{C}^V$;
5: $\mathcal{C}^V$= *findCS*($i-1$, $i+1$);
6: **end if**
7: **if** There exist two strips S_i, S_{i+1}, which satisfy $S_i.ub < q_x < S_{i+1}.lb$ **then**
8: $\mathcal{C}^V$= *findCS*($i-1$, $i+1$);
9: **else**
10: **if** $\forall S_i$ ($0 \leq i \leq n$), $S_i.lb > q_x$ **then**
11: Insert the first c elements of L_V into $\mathcal{C}^V$;
12: **else**
13: $\forall S_i$ ($0 \leq i \leq n$), $S_i.ub < q_x$
14: Insert the last c elements of L_V into $\mathcal{C}^V$;
15: **end if**
16: **end if**
17: **return** $\mathcal{C}^V$;

5.2.5 Scalability of DKNN

The DKNN algorithm is easily parallelizable and scales well horizontally (scale-out) with respect to the number of servers to handle increases in data volumes. Let $\mathcal{F}$ be the set of candidate strips for a given query. These strips in general reside on different servers, and the the process of searching them for the k-NNs can take place simultaneously on individual servers. More processing power can be obtained by simply adding more servers to the cluster. The throughput of DKNN is roughly proportional to the number of servers.

Algorithm 3 *findCS* Algorithm

Input:
int j, h. *//j and h are the indexes of strips*
Output: :
vertical candidate strips set $\mathcal{C}^V$.
1: **while** $j \geq 0$ and $h \leq n$ and $|\mathcal{C}^V| \leq c$ **do**
2: **if** dist(s_j, q)<dist(s_h, q) **then**
3: $\mathcal{C}^V = \mathcal{C}^V \cup S_j$; j++;
4: **else**
5: $\mathcal{C}^V = \mathcal{C}^V \cup S_h$; h++;
6: **end if**
7: **end while**
8: **return** $\mathcal{C}^V$;

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

6 Implementation on the S4 Platform

We implement our method on Apache S4, an open-source general-purpose distributed platform originally developed by and successfully deployed at Yahoo! in the context of search applications. The main reason for choosing S4 as the basis for our implementation is that it allows for the processing of unbounded streams data, which is a desirable property in spatio-temporal applications. While S4 is chosen as the implementation platform, other similar stream processing engines such as Storm [36] can also be used for this purpose.

6.1 Overview of S4 and Deploying DSI

S4 adopts the Actor model, and data flow within the system as "events". An event takes the form of ⟨*event type, a key-value pair, message entity*⟩. The basic logical computing unit in S4 is called a *Processing Element* (PE). A PE can selectively consume events output from other PEs, and emit new events that contain new messages, which can then be consumed by other PEs. PEs are designed to be very light-weight; thus one server (called a *processing node*) can run thousands of PEs.

The task of deploying DSI on S4 involves two types of PEs, *EntrancePE* and *IndexPE*. The EntrancePE stores the lists L_V and L_H. IndexPEs are assigned to different nodes in the cluster through hashing, and each IndexPE manages one strip.

6.2 Deploying DKNN

We now discuss how to run the DKNN algorithm on S4 to process k-NN queries. The DKNN algorithm can be decomposed into several steps, and each step can be handled by a particular type of PEs. We call the resulting scheme DKNN-S4. In what follows, we explain in more detail this platform.

6.2.1 The functionality of EntrancePE

The EntrancePE is the entrance to receive all object updates and queries. It sends NewObEvents and OldObEvents to the corresponding IndexPE to update the locations of objects. In addition, when a query q (q_x, q_y) arrives, the EntrancePE will send $2\lceil k/\chi \rceil$ QueryPieceEvents that carry the information of q to the corresponding IndexPEs that should process q.

6.2.2 The functionalities of IndexPE and RadiusPE

Each IndexPE maintains only one strip. An IndexPE receives the NewObEvent and the OldObEvent to insert or delete the corresponding object. When ipe_i receives a QueryPieceEvent, it will process this query, then send a SingleObEvent to the RadiusPE.

When RadiusPE receives all SingleObEvents about query q, it will compute the radius r_q. Next, the it determines which IndexPEs should continue to compute k NNs for q, and sends a RadiusEvent to them. When these IndexesPEs receive the RadiusEvent, they will find the objects within the circle C_q and send a CanObEvent containing these objects to the MergePE.

6.2.3 The functionalities of MergePE and OutputPE

When a MergePE receives all CanObEvents regarding q, it will compute the k nearest neighbors of q. Then the OutputPE outputs the results to the client.

7 Experimental Evaluation

We conduct experiments to evaluate the proposed DSI index and DKNN algorithm. For DSI, we mainly test two aspects: (1) the performance of DSI, and (2) the effect of various parameters on the performance.

To evaluate the performance of DKNN, we implement three other algorithms on S4 as baseline methods. The first method, NS, is a naive search algorithm which does not use any index. For any object, it uses a hash function to determine which server should store it. Processing the k-NN queries thus involves scanning objects stored in all servers. The second method, Grid, is a distributed search algorithm that employs the same search strategy proposed by [3] based on the grid index, as discussed in Section 3. In the Grid algorithm, we set the length of cells as 0.001. In the third method, Grid-Modified, each server maintains a copy of a grid index on all of the objects, and queries are randomly routed to different servers for processing using the YPK-CNN method [3].

For the DKNN algorithm, we first evaluate its performance with varying parameter values, and then compare it with other three algorithms. Every experiment is repeated ten times, and the average values are recorded as the final results.

7.1 Experimental Setup

The experiments are conducted on a cluster of 8 Dell R210 servers with Gigabit Ethernet interconnect. Each node has a 2.4GHz Intel processor and 8GB of RAM.

We use the German road network data to simulate four different datasets for our experiments. In all datasets, all objects appear on the roads only. In the first dataset (UD), the objects follow a uniform distribution. In the second dataset (GD1), 70% of the objects follow the Gaussian distribution, and the other objects are uniformly distributed. The third dataset (GD2) also has 70% of the objects following the Gaussian distribution, but they are more concentrated. The objects in the fourth dataset conform to the Zipf distribution and is generated in the following way. We first rank the roads in descending order based on their length. For any road r_i, we use h_i to represent its rank and assign to it a probability $P(r_i) = \frac{C}{h_i^\alpha}$, where C and α are constants set as 0.001 and 0.9 respectively. Next, we let the percentage of objects appearing on road r_i be equal to $P(r_i)$, and the objects are randomly distributed on r_i. In all four datasets, the whole area is normalized to a unit square, and the objects move along the road network, with the velocity uniformly distributed in [0, 0.002] unless otherwise specified.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

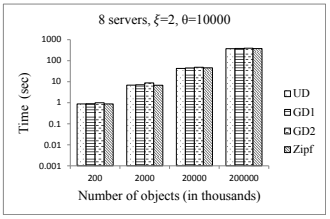


Fig. 7. Computation time for building DSI

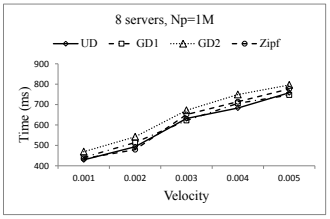


Fig. 8. Maintenance cost w.r.t velocity

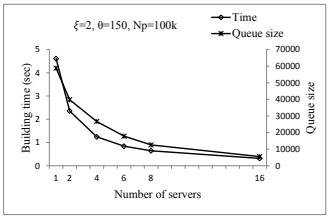


Fig. 9. Scalability of DSI w.r.t. the number of servers

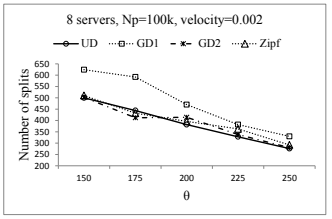


Fig. 10. Number of splits w.r.t θ

7.2 Performance of index construction and maintenance

Time of building DSI. We first test the time of building DSI from scratch to index different numbers of objects. Fig. 7 shows the time of building DSI as we vary the number of objects. In our study, the size of each object is approximately 50B. So in this experiment we handle about 10GB of data when the number of objects takes the maximum value. The time it takes to build the index increases almost linearly with the increasing number of objects. The time is slightly higher for GD2, indicating that a more clustered distribution may lead to a higher cost of building DSI. Unless otherwise noted, the following results are based on experiments carried out using the GD2 dataset.

Effect of the velocity of objects. Fig. 8 demonstrates the effect of the velocity of objects on the computation time for maintaining DSI based on four datasets. In this set of experiments, we first build the DSI for 1M objects, and then 100K object are chosen to move continuously with varying velocities. As expected, the faster the objects move, the more split and merge operations happen, leading to an increase in maintenance time.

Scalability of DSI. Fig. 9 shows the scalability of DSI with varying number of servers, where the number of objects to be indexed is 100K. As shown in Fig. 9, the time for building the index drops almost linearly and the maximum size of the queue that buffers the objects to be processed also decreases notably when the number of servers increases, testifying to the good scalability of the index.

Effect of θ on DSI. We next study the effect of various parameters of DSI on its performance. Fig. 10 shows the effect of the maximum capacity, θ, on the frequency of split. The number of moving objects indexed is 100K. As can be observed from Fig. 10, the split frequency is approximately reversely proportional to the value of θ; a greater θ value would result in a reduction in the number of splits. Of course, θ cannot be overly large, because that will increase the time for processing queries, which will be shown in the following experiments. In addition, we find that the split frequency is also influenced by the distribution of objects. Since the objects are more clustered in GD2, there is a higher probability of some strip becoming full, which results in more splits.

Effect of ξ on DSI. Fig. 11 shows the influence of the minimum occupancy, ξ, on the frequency of merge operations. A larger value of ξ means that underflow will occur more often and thus cause more merge operations.

Effect of update frequency. The cost of DSI maintenance is also affected by the number of objects to be updated at each step. Here, we first build the DSI for 100K objects, and vary the percentage of objects whose positions are concurrently updated with the maximum velocity being 0.002. As can be observed from the results shown in Fig. 12, the time for maintaining DSI grows almost linearly with the percentage of objects with position updates, regardless of the data distribution.

Comparison of DKNN and Grid-Modified. We evaluate the maintenance costs of DKNN and Grid-Modified using the same experimental setting as the preceding experiment, i.e. we vary the percentage of objects with concurrent position updates, and the results are shown in Fig. 13. For a given set of objects to be updated, the maintenance cost includes the time of transmitting these objects from the master to the corresponding slaves and the time of accomplishing the updates of all objects by all slaves. The results demonstrate that the maintenance cost of Grid-Modified increases linearly with the percentage of objects with position updates, but the maintenance cost of DKNN is only slightly influenced by the varying number of objects. The reason is that the set of moving objects are separately maintained by multiple servers with DSI in DKNN and each server only needs to handle a subset of the moving objects, while in the case of Grid-Modified, every server has a complete index and thus needs to update the positions of all moving objects.

We further measure the largest datasets that can be supported by DKNN and Grid-Modified with varying number of servers, where each objects contains only its identifier and location, and the results are shown in Fig. 14. With more servers being used, the size of the largest dataset supported by Grid-Modified almost does not change because it is determined by the size of the main memory of a single server, while the scale of the largest dataset that DKNN can support increases linearly, as the data are distributedly indexed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

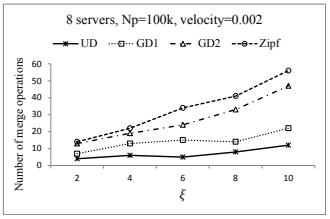


Fig. 11. Number of merges w.r.t ξ

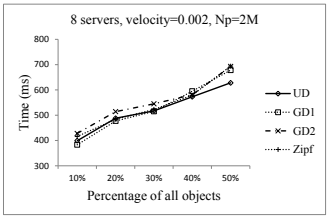


Fig. 12. DSI maintenance cost vs. percentage of updated objects

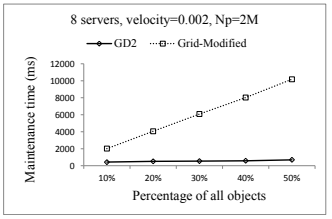


Fig. 13. Comparison of DKNN and Grid-Modified w.r.t maintenance cost

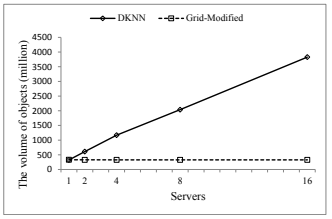


Fig. 14. The largest datasets supported by DKNN and Grid-Modified

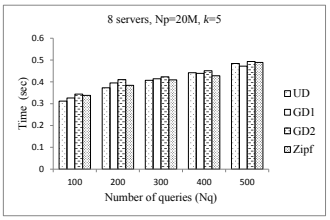


Fig. 15. Performance of DKNN w.r.t number of queries

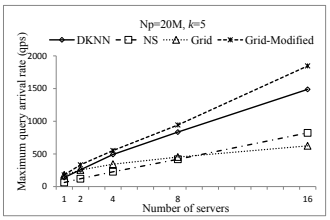


Fig. 16. Throughput of DKNN

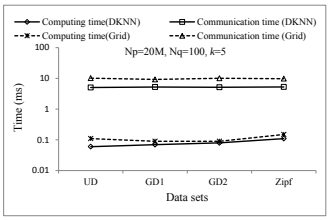


Fig. 17. Comparison of computing time and communication time

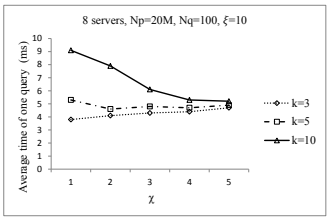


Fig. 18. Performance of DKNN w.r.t χ

Although the size of the largest dataset supported by Grid-Modified seems huge in this evaluation, the real dataset that Grid-Modified can support cannot hardly reach this massive scale because the objects in real datasets not only contain the identifier and location but also include more information. For example, the volume of the Tiger Files that is utilized in SpatialHadoop [35] reaches 60GB with only 70 million spatial objects. Therefore, the largest dataset that Grid-Modified can support will be in general much smaller than that in Fig. 14.

7.3 Performance of query processing

We now perform experiments to evaluate the performance of the DKNN algorithm.

Processing time. We feed a batch of queries into our system in one shot, and measure the time between the first query entering the system and the k-NN results of all queries having been obtained. We vary the number of queries and data distributions, and the results are shown in Fig. 15. As can be observed from Fig. 15, DKNN achieves similar performance for different distributions, with GD2 being slightly more time consuming. This is because every strip in DSI contains at most θ objects and typically each query only involves a few strips. Therefore, the data distribution only has a slight impact on the query processing time. This is in contrast to space-partitioning approaches such as the grid index, which cannot adapt to data distributions.

Throughput of DKNN. We record the maximum sustainable query arrival rate (with N_p=20M) for varying number of servers, and the results are shown in Fig. 16. The maximum sustainable rates are determined by observing whether an overflow has occurred in a processing node's buffer queue, which is used to buffer queries yet to be processed by the server. As can be observed from Fig. 16, the maximum query arrival rate of Grid-Modified increases linearly as more nodes are used and it is greater than that of other methods. Because each server in Grid-Modified processes queries independently , its throughput grows proportionally to the number of servers. The throughput of DKNN is not as good, but close to that of Grid-Modified, because as a distributed solution, more communication cost is involved. The trend, however, is still roughly linear with respect to the number of servers.

Comparison of communication and computing costs. We evaluate the communication and computing time of processing one query with DKNN and Grid approaches based on four datasets in Fig. 17. The experimental results show that the communication cost is almost two orders of magnitude greater than the computing cost for processing one query, which verify our view that the communication between different servers in a distributed cluster is very expensive compared with the CPU cost.

Effect of χ on DKNN. In DKNN, we select χ objects from each candidate strip to form the set of supporting objects. Fig. 18 shows the influence of χ on the cost of processing queries. In this set of experiments, we feed 100 queries into the system and record the average processing time of a query. The results show that χ has little effect on the processing time when k takes smaller values (3 and 5). When k increases, the influence becomes more obvious. This observation confirms our analysis about χ in Section 5. In general, χ does not have a significant impact on the performance of DKNN.

Effect of θ on DKNN. Fig. 19 shows that the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

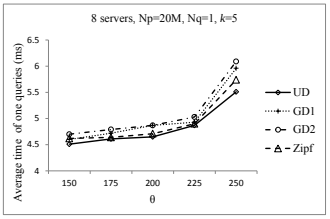


Fig. 19. Performance of DKNN w.r.t θ

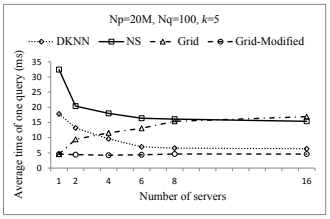


Fig. 20. Query evaluation time w.r.t number of servers

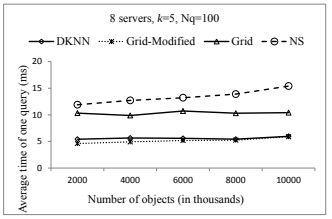


Fig. 21. Query evaluation time w.r.t N_p

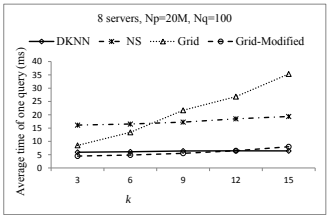


Fig. 22. Query evaluation time w.r.t k

value of maximum capacity, θ, has a clear effect on the performance of query processing regardless of the data distribution. The processing time increases rapidly with increasing θ. Recall from a previous set of experiments that increasing θ reduces the number of splits in DSI. Therefore, the optimal value of θ has to be chosen based on the actual workload.

7.4 Comparison with baseline methods

In this section, we compare DKNN with three baseline methods, NS, Grid and Grid-Modified.

Scalability w.r.t. number of servers. We evaluate the scalability of the four algorithms with respect to available hardware. We vary the number of servers, and measure the average response time per query. As shown in Fig. 20, with more servers being employed, DKNN and NS algorithms all enjoy a decrease in processing time, but DKNN performs much better than NS. The processing time of Grid-Modified stays level as we vary the number of servers. The reason is that Grid-Modified handles each query in a centralized way by a single server; thus the number of servers has little impact on the average response time per query. Furthermore, the performance of Grid deteriorates when the number of servers increases, because it involves more iterative communications among more servers. In this evaluation, the performance of DKNN gets very close to that of Grid-Modified with more than four servers.

Effect of the number of objects. Next, we examine the influence of the number of objects on the performance of the four algorithms. As shown in Fig. 21, the NS algorithm suffers greatly from an increasing number of objects, because without an index, it has to go through all objects. The indexes employed by DKNN, Grid, and Grid-Modified all can prune the search space, and therefore they are not heavily affected by the number of objects. In this experiment, DKNN still performs close to Grid-Modified.

Effect of k. Finally, we study the influence of k on the four algorithms. As shown in Fig. 22, the processing time of DKNN almost remains unchanged as k increases, the reason being that we can adjust the value of χ accordingly to accommodate the increase in processing time. When k increases, Grid needs more iterations to compute the results, and thus its processing time increases more rapidly than the other methods. For the same reason, the performance of Grid-Modified also degrades with increasing k and it takes more time than DKNN to process the queries with k being 15. Because NS always searches all objects for each query, k only slightly influences its performance.

In the above experiments, we find that Grid-Modified performs slightly better than DKNN in terms of query processing cost, but it incurs higher maintenance cost than DKNN for updating the same scale of moving objects and the largest dataset that Grid-Modified can support is bounded by the available memory in a single server. Compared with DKNN, each server in Grid-Modified has to consume more memory and has a higher maintenance cost to index all objects, which restricts its scalability to handle the datasets with massive volumes because the huge datasets (e.g. Tiger Files) can hardly fit in the main memory of a single server. In summary, DKNN is more suitable for processing k-NN queries over large volumes of moving objects using a distributed cluster.

8 Conclusions

The problem of processing k-NN queries over moving objects is fundamental in many applications. The large volume of data and heavy query workloads call for new scalable solutions. We propose DSI, a distributed strip index, and DKNN, a distributed k-NN search algorithm, to address this challenge. Both DSI and DKNN are designed with distributed processing in mind, and can be easily deployed to a distributed system. DSI is a data partitioning index and is able to adapt to different data distributions. Based on DSI, we present the DKNN algorithm that can directly determine a region that is guaranteed to contain the k-NNs for a given query with only two iterations. This has a clear cost benefit when compared with existing approaches, such as grid-based methods, which require an uncertain number of iterations. We show how the proposed index and algorithm can be implemented with S4. Extensive experiments confirm the superiority of the proposed method.

For future work, we would like to explore how to evaluate continuous k-NN queries over moving objects using the strip index. For a given k-NN query q, it is very possible that its result (a list of objects)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2364046, IEEE Transactions on Knowledge and Data Engineering

remains relatively stable when objects move with reasonable velocities. Therefore, it is promising to investigate how the k-NN results can be incrementally updated as objects move.

REFERENCES

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, *Big data: The next frontier for innovation, competition, and productivity*. McKinsey Global Institute, 2011.

[2] Wechat. http://en.wikipedia.org/wiki/WeChat.

[3] X. Yu, K. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *ICDE*, 2005, pp. 631–642.

[4] B. Zheng, J. Xu, W.-C. Lee, and L. Lee, "Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services," *The VLDB Journal*, vol. 15, no. 1, pp. 21–39, 2006.

[5] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of spatial queries in wireless broadcast environments," *IEEE Transactions on Mobile Computing*, vol. 8, no. 10, pp. 1297–1311, 2009.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SIGOPS*, 2003, pp. 29–43.

[8] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[9] R. S. Norbert Beckmann, Hans-Peter Kriegel and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.

[10] Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: an optimized spatio-temporal access method for predictive queries," in *VLDB*, 2003, pp. 790–801.

[11] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching," in *SODA*, 1994, pp. 573–582.

[12] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring," in *SIGMOD*, 2005, pp. 634–645.

[13] M. A. Cheema, X. Lin, Y. Zhang, W. Wang, and W. Zhang, "Lazy updates: An efficient technique to continuously monitoring reverse knn," in *PVLDB*, 2009, pp. 1138–1149.

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

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDM Workshop*, 2010, pp. 170–177.

[16] K. L. Cheung and A. W.-C. Fu, "Enhanced nearest neighbour search on the r-tree," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 16–21, 1998.

[17] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002, pp. 287–298.

[18] K. Mouratidis and D. Papadias, "Continuous nearest neighbor queries over sliding windows," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 6, pp. 789–803, 2007.

[19] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos, "Fast nearest-neighbor query processing in moving-object databases," *GeoInformatica*, vol. 7, no. 2, pp. 113–137, 2003.

[20] T. Seidl and H. Kriegel, "Optimal multi-step k-nearest neighbor search," in *SIGMOD*, 1998, pp. 154–165.

[21] S. Chaudhuri and L. Gravano, "Evaluating top-k selection queries," in *VLDB*, 1999, pp. 399–410.

[22] B. Gedik, K. Wu, P. Yu, and L. Liu, "Processing moving queries over moving objects using motion-adaptive indexes," *Knowledge and Data Engineering*, vol. 18, no. 5, pp. 651–668, 2006.

[23] C. Yu, B. Ooi, K. Tan, and H. Jagadish, "Indexing the distance: An efficient method to knn processing," in *VLDB*, 2001, pp. 421–430.

[24] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: scalable incremental processing of continuous queries in spatio-temporal databases," in *SIGMOD*, 2004, pp. 623–634.

[25] X. Xiong, M. F. Mokbel, and W. G. Aref, "SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *ICDE*, 2005, pp. 643–654.

[26] D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," in *SIGMOD*, 2012, pp. 37–48.

[27] H. Wang and R. Zimmermann, "Snapshot location-based query processing on moving objects in road networks," in *SIGSPATIAL GIS*, 2008, pp. 50:1–50:4.

[28] M. F. Mokbel and W. G. Aref, "SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams," *The VLDB Journal*, vol. 17, no. 5, pp. 971–995, 2008.

[29] R. V. Nehme and E. A. Rundensteiner, "Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects," in *EDBT*, 2006, pp. 1001–1019.

[30] B. Gedik and L. Liu, "Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system," in *EDBT*, 2004, pp. 523–524.

[31] H. Wang, R. Zimmermann, and W. Ku, "Distributed continuous range query processing on moving objects," in *DEXA*, 2006, pp. 655–665.

[32] W. Wu, W. Guo, and K. Tan, "Distributed processing of moving k-nearest-neighbor query on moving objects," in *ICDE*, 2007, pp. 1116–1125.

[33] B. Bamba, L. Liu, A. Iyengar, and P. Yu, "Distributed processing of spatial alarms: a safe region-based approach," in *ICDCS*, 2009, pp. 207–214.

[34] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *EDBT*, 2012, pp. 38–49.

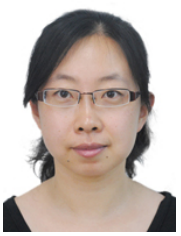
[35] A. Eldawy, "A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data," in *VLDB*, 2013.

[36] Twitter storm. https://github.com/nathanmarz/storm.

[37] L. Lin, X. Yu, and N. Koudas, "Pollux: Towards scalable distributed real-time search on microblogs," in *EDBT*, 2013, pp. 335–346.

Ziqiang Yu received the BSc degree from Jinan University, China, and the MSc degree from Lanzhou Jiaotong University, China. He is currently a PhD Candidate in the School of Computer Science and Technology, Shandong University, China. His research interest is in the area of database systems.

Xiaohui Yu received his BSc degree from Nanjing University, China, MPhil degree from The Chinese University of Hong Kong, and PhD degree from the University of Toronto, Canada. His research interests are in the areas of database systems and data mining. He is a Professor in the School of Computer Science and Technology, Shandong University, China, and an associate professor in the School of Information Technology, York University, Canada (on leave).



Yang Liu received the BSc degree from the Harbin Institute of Technology, China, and the MSc and PhD degrees from York University, Canada. She is currently an associate professor in the School of Computer Science and Technology, Shandong University, China. Her main areas of research are data mining and information retrieval.

Ken Q. Pu is an associate professor of Computer Science at the University of Ontario Institute of Technology, Canada. His current research interest is database systems that can scale from embedded systems to distributed systems seamlessly.