Efficient Snapshot KNN Join Processing for Large Data Using MapReduce

Conference Paper · December 2016

DOI: 10.1109/ICPADS.2016.0098

5 authors, including:

Yupeng Hu
Shandong University
32 PUBLICATIONS **182** CITATIONS
SEE PROFILE

Cun Ji
Shandong Normal University
42 PUBLICATIONS **306** CITATIONS
SEE PROFILE

Yang Xu
Shandong Normal University
13 PUBLICATIONS **102** CITATIONS
SEE PROFILE

Xueqing Li
INTI International University and Colleges
56 PUBLICATIONS **209** CITATIONS
SEE PROFILE

Efficient Snapshot KNN Join Processing for Large Data Using MapReduce

Yupeng Hu[1,2], Chong Yang[1], Cun Ji[1], Yang Xu[1], Xueqing Li[1,*]
[1]*School of Computer Science and Technology, Shandong University, P.R. China*
[2]*State Key Lab. for Novel Software Technology, Nanjing University, P.R. China*
{*huyupeng, jicun, xuyang0211, xqli*}*@sdu.edu.cn yangchong.sdu@163.com*

***Abstract*—The kNN join problem, denoted by $R \times_{KNN} S$, is to find the k nearest neighbors from a given dataset S for each point in the query set R. It is an operation required by many big data applications. As large volume of data are continuously generated in more and more real-life cases, we address the problem of monitoring kNN join results on data streams. Specifically, we are concerned with answering kNN join periodically at each snapshot which is called snapshot kNN join. Existing kNN join solutions mainly solve the problem on static datasets, or on a single centralized machine, which are difficult to scale to large data on data streams. In this paper, we propose to incrementally calculate the kNN join results of time t_i from the results of snapshot t_{i-1}. Typically, for the data continuously generated on the data stream, we can get $S_i = S_{i-1} + \Delta S_i$ for the valid datasets of adjacent snapshots, where ΔS_i denotes the updated points between time t_{i-1} and t_i. Our basic idea is to first find the queries in R whose kNN results can be affected by the updated points in ΔS_i, and then update the kNN results of these small part of queries respectively. In this way, we can avoid calculating the kNN join results on the whole dataset S_i in time t_i. We propose an implementation of searching for affected query points in MapReduce to scale to large volume of data. In brief, the mappers partition the datasets into groups, and the reducers search for affected queries separately on each group of points. Furthermore, we present the enhanced strategies of data partitioning and grouping to reduce the shuffling cost and computational cost. Extensive experiments on real-world datasets demonstrate that our proposed methods are efficient, robust, and scalable.**

***Keywords*-KNN Join; Data Streams; MapReduce; Large Data**

I. INTRODUCTION

The k nearest neighbor join (kNN join) is an important and frequently-used operation for numerous data mining applications. It is an expensive operation, especially on large data sets and/or in multi-dimensions. As data are continuously generated rather than in a static dataset in more and more applications in the internet, it calls for solutions to continuously answer kNN join queries at each snapshot on data streams. For example, social network websites often provide recommendation functions. If we represent item and users all by feature vectors in the same dimensional space, a sensible and robust approach to support content-based filtering is to recommend to the user the items that are his/her nearest neighbors, for a given distance metric. Development of feature vectors and similarity metrics have been extensively studied in literature. When there are newly added items in the period of time, it is essential to recommend to the users new contents that are of interest while the contents are still fresh, as there is a clear tendency for users of such applications to prefer newer contents. Thus, the main issue we tackle is to efficiently process kNN join periodically at given snapshots to ensure new contents are recommended to the users.

We address the problem of snapshot kNN join on data streams, which answers kNN results from the dataset S_i, representing the valid dataset of time t_i on the data stream, for a given query set R at each time snapshot $t_i (i \in \{0, 1, \cdots\})$. Typically, $S_i = S_{i-1} + \Delta S_i$, where ΔS_i denotes the updated points with respect to the last snapshot. The naive solution is to process kNN join at each time snapshot t_i, where most existing kNN join methods on static datasets can be adopted, including methods designed for a single server [1][2] and methods using MapReduce [4][5]. However, the cost is often huge for large data as we have to process the expensive kNN join repeatedly at each snapshot. On the other hand, existing solutions for kNN join on data streams [9] are mainly designed for a singe centralized server, which is hard to scale to large volume and velocity of data streams.

In this paper, we propose an incremental solution to avoid computing kNN join repeatedly at each snapshot and implement it in MapReduce. The basic idea of our approach is to firstly identify the queries in R whose kNN results of time t_{i-1} can be affected by the updated points in ΔS_i, followed by updating the kNN results for this subset of queries. Considering there are large volume of query points and updates, we introduce an implementation using MapReduce to efficiently search for affected queries. Specifically, the mappers assigns a key to each points from R and ΔS_i; the points with the same key are distributed to the same reducer in the shuffling process; the reducer performs the search for affected queries over the points that are assigned to it.

To guarantee the correctness of the kNN join results, for each updated point s' in partition ΔS_{iy}, the affected queries of s' on R should be sent to the same reducer as s' does. As a result, query points in R may be replicated and distributed into multiple reducers. The extensive of replicas can increase the shuffling cost and the computational cost. Thus, we need

* To whom correspondence should be addressed.

to design good mapping functions to minimize the number of replicas to improve the performance.

In summary, we make the following contributions.

1) We propose the problem of snapshot kNN join processing, and present efficient solution and implementation in MapReduce. The implementation defines the mapper and reducer jobs and requires no modifications to the MapReduce framework.
2) We design an efficient mapping method that divides points into groups using clustering technique. The proposed method can significantly reduce the number of replicas, and thus incurs less shuffling cost and computational cost.
3) We conduct extensive experiments on a real-life dataset and results show that our proposed methods are robust, efficient, and scalable.

II. Related work

Index structures designed to cope with kNN join have been extensively studied in literature. Bohm et al. [1] first propose the kNN join problem which finds the k nearest neighbors for a set of queries in a single-run operation. The kNN join algorithms like MuX [1], Gorder [2], and iJoin [3] mainly use nested loop searching techniques, which are designed for static dataset and on a single server. As the amount of data to be processed increases significantly, distributed processing of kNN join for massive data using MapReduce has attracted great attention. PGBJ [4] uses a preprocessing and distance based partitioning strategy (based on Voronoi diagram) to reduce the number of tasks. Zhang et al. [5] present two kNN join processing methods in MapReduce, H-BNLJ and H-BRJ, which adopts the block nested loop methodology. Although H-BNLJ and H-BRJ need no preprocessing, multiple successive MapReduce jobs are required. Furthermore, they propose an approximate method named H-zkNNJ based on z-value preprocessing for more efficient processing. In RankReduce [6], the authors preprocess and partition data into buckets and process kNN join using LSH. However, their methods are designed for static datasets and do not work well for our problem because we have to process kNN join repeatedly at every snapshot.

The problem of kNN join processing on data streams are mostly studied in a single centralized setting. Tok and Bressan [7] introduce the framework of progressive and approximate join algorithms on data streams, but the efficiency of their methods drops greatly as the dimensionality of data increases. Yu et al. [8] consider the incremental update of high-dimensional kNN joins. They use Sphere-tree and iDistance structures to reduce I/O cost. Yang et al. [9] propose to use continuous kNN join processing for real-time recommendation. They propose two index structures, HDR-tree and HDR*-tree, to reduce in-memory searching cost. Unfortunately, their consider a different problem on data streams rather than snapshot kNN join processing, and the methods are difficult to scale to large data sets.

Besides, our work is also related to reverse kNN (RkNN) query problem, which finds all the objects that have the query as one of their k nearest neighbors. Korn et al. [10] propose the Rnn-tree which index the precomputed distance of nearest neighbor. Tao et al. [11] propose a TPL algorithm to deal with multi-dimensional RkNN query. Zhang and Alhajj [12] present the NAQ-tree to handle RNN query in high-dimensional metric space. Again, existing methods are mainly designed for a single centralized setting.

III. Preliminaries

In this section, we give the formal definition of snapshot kNN join problem and briefly introduce the MapReduce framework.

A. Problem definition

Considering two datasets R and S, with each $r \in R$ and $s \in S$ to be points in the d-dimensional metric space, we give the definition of kNN join in Definition 1. Let $dist(r, s)$ be the distance between r and s. We adopt the frequently-used Euclidean distance as the distance measure in this paper. Without loss of generality, our solutions can be easily generalized to other distance metrics as well.

Definition 1: (kNN join) Given two d-dimensional point datasets R and S, the kNN join of R and S, denoted by $R \times_{KNN} S$, is to find the k nearest neighbors for each $r \in R$ on S.

We are concerned with snapshot kNN join problem on the data stream, which means that the dataset is continuously generated. Let S_0 be the valid dataset in the beginning time t_0. **Snapshot kNN join** returns the kNN join results of R at each time snapshot $t_i (i \in \{0, 1, \cdots\})$. Typically, $S_i = S_{i-1} + \Delta S_i$, where ΔS_i denotes the updated points with respect to the last snapshot. As the data sets are often very large in real-life applications, we use MapReduce to distributedly process the problem.

B. MapReduce framework

MapReduce [13] is a programming framework to support processing large datasets using share-nothing clusters. In MapReduce, input data are represented as key-value pairs. A MapReduce programme typically consists of a pair of user-defined Map and Reduce functions. Map function performs filtering and sorting, which takes an input key-value pair and produces a set of key-value pairs in a different domain. Reduce performs a summary operation, which accepts an intermediate key and its corresponding values produced from the Map function, applies the processing logic, and produces the final result which is typically a list of values. Apache Hadoop is one of the common-used distributed processing platform that adopts MapReduce framework and provides good reliability and fault-tolerance, and thus our experiments are conducted on Apache Hadoop.

IV. Snapshot kNN join processing using MapReduce

A. Overview

As there have been extensive studies on the kNN join problem on static datasets, we can easily get the initial kNN join results at time t_0 using existing methods like PGBJ [4]. The naive solution to get the kNN join results at snapshot $t_i (i \geq 1)$, is to re-compute the kNN join results at each snapshot similar to the process of the initial kNN join. However, we have to repeatedly compute the unchanged part of dataset in the naive solution, which is very expensive.

Thus, we propose to incrementally update the kNN join results to sufficiently reuse the results of last snapshot. Our solution for kNN join of snapshot t_i consists of two steps: (1) finding a subset of R, denoted by R'_i, whose kNN results can be affected by the updated points in ΔS_i with respect to the kNN join results of snapshot t_{i-1}, (2) then updating the kNN results of R'_i, where $|R'_i|$ is often much smaller than $|R|$. Step (1) is actually a RkNN join problem as presented in Definition 2.

Definition 2: (RkNN join) Given two d-dimensional point datasets R and S, the reverse kNN join of R and S, denoted by $R \times_{RKNN} S$, is to find all points in S that have r as one of their k nearest neighbors for each $r \in R$.

As we already have the kNN join results of snapshot t_{i-1}, we can get the following observation. For any points in ΔS_i, if it does not fall into the sphere that centers at $r \in R$ with a radius equals to the distance between r and its current k-th NN of snapshot t_{i-1}, it must not become one of the kNN results of r in time t_i. So we have the following lemma, which has been exploited in previous work [8].

Lemma 1: In the processing of snapshot kNN join in time t_i, an updated point s' in ΔS_i will affect the kNN join results of r with respect to time t_{i-1}, only if s' is in the sphere centered at r with radius $dknn_r$, where $dknn_r$ is the distance between r and its k-th nearest neighbor in time t_{i-1}.

According to Lemma 1, we can find the RkNN results $R'_{s'}$ for any point s' in ΔS_i by searching for the queries in R that satisfy $dist(s', r) \leq dknn_r$. Then, for any point s' in ΔS_i, update the kNN results of all the queries in $R'_{s'}$, and update $dknn_r$ as well.

Therefore, the main problem now is to efficiently search for the RkNN join results $\Delta S_i \times_{RKNN} R$. We propose two solutions in MapReduce to efficiently find the RkNN join results of ΔS_i , and then get the kNN join results of R in time t_i.

B. Basic solution

In the basic solution, the mappers assign each point in ΔS_i and R a key. ΔS_i and R are split into disjoint subsets based on the key, i.e., $R = \bigcup_{1 \leq x \leq N} R_x$, $\Delta S_i = \bigcup_{1 \leq x \leq N} \Delta S_{ix}$ where $R_x \cap R_y = \Phi, \Delta S_{ix} \cap \Delta S_{iy} = \Phi, x \neq y$. Each subset of R_x and ΔS_{iy} is distributed to a reducer. A reducer performs a RkNN join between a pair of R_x and ΔS_{iy}, i.e., computes the distance between each $r \in R_x$ and each $s' \in \Delta S_{iy}$; and if $dist(r, s') \leq dknn_r$, add r into the RkNN results list of s'. Finally, the RkNN results of all pairs of subsets are merged and $\Delta S_i \times_{RKNN} R = \bigcup_{1 \leq x,y \leq N} R_x \times_{RKNN} \Delta S_{iy}$.

The performance of the process is affected by two major considerations: the shuffling cost of sending intermediate results from mappers to reducers and the cost of performing the RkNN join on the reducers. There are two MapReduce jobs in the basic solution. In the first MapReduce job, each point in ΔS_i and R is sent to the reducers N times, thus the shuffling cost is $N(|R| + |\Delta S_i|)$. Since R is partitioned into several subsets, the RkNN join results of ΔS_{iy} are incomplete, and another MapReduce is required to combine the results of $\Delta S_{iy} \times_{RKNN} R_x$ for all $1 \leq x \leq N$, whose shuffling cost is $\sum_x \sum_y |\Delta S_{iy} \times_{RKNN} R_x|$. Therefore, the shuffling cost of the basic solution is $N(|R| + |\Delta S_i|) + \sum_x \sum_y |\Delta S_{iy} \times_{RKNN} R_x|$. Accordingly, the distances between all points in ΔS_i and R are computed once, thus the computational cost is $a|\Delta S_i||R|$, where a is a constant.

C. Enhanced solution

1) Data preprocessing: The main drawback of the basic solution is that R has to be replicated to all reducers of ΔS_{iy} for $1 \leq y \leq N$. A better strategy is that ΔS_i is partitioned into N disjoint subsets and for each subset ΔS_{iy}, find a subset of R_y such that $\Delta S_{iy} \times_{RKNN} R = \Delta \times_{RKNN} R_y$. Then, instead of sending the entire R to each reducer, R_y is sent to the reducer that ΔS_{iy} belongs to. Notice that $R_y \cap R_x$ may not be empty as it is possible that a query r can be the RkNN results of both $s'_0 \in \Delta S_{ix}$ and $s'_1 \in \Delta S_{iy}$. Thus, we can reduce the shuffling cost to $|\Delta S_i| + \alpha R$, where α is the average number of replicas of a query point in R.

The main issue of this solution is how to efficiently find a subset R_y for each partition ΔS_{iy}. To solve the problem we make some preprocessing on R. We first find a set of N pivots on R. The methods adopted is to do sampling on R and apply k-means clustering on the sample. The pivots are set to the centers of all the clusters. Then, each query point in R is assigned to the nearest pivot. Let $radius_p$ be the radius of a cluster C^p, i.e., the maximum distance between the pivot p and the points assigned to it. Let the $maxdknn_p$ be the maximum $dknn$ value of the query points in a cluster in time t_{i-1}. We can easily get the following theorem according to the triangle inequality of distance metric.

Theorem 1: If the distance between a point s' and the cluster (centered at p) is larger than $maxdknn_p$, the distance between p and any point r in the cluster is greater than $dknn_r$. That is,

$$\begin{aligned} &dist(s', p) - radius_p > maxdknn_p \\ &\quad \Longrightarrow dist(s', r) > dknn_r, \forall r \in C^p \end{aligned} \tag{1}$$

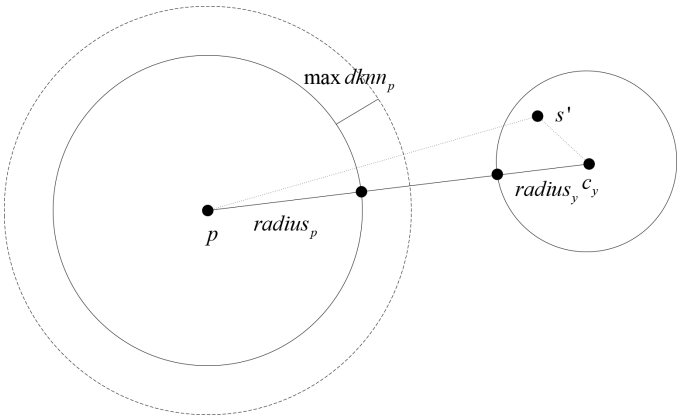

Figure 1. Prune a cluster of R for a partition

We also partition ΔS_i using clustering technique with each cluster as a partition, so that the points in the cluster are more likely to have the same RkNN results. Note that although the pivots are selected based on R, they can be used to partition ΔS_i as well to reduce the clustering cost on ΔS_i. For simplicity of analysis, we set the number of partitions on ΔS_i equivalent to N, where N is the number of pivots on R. Let the cluster center and radius of partition ΔS_{iy} be c_y and $radius_y$. We can get the following lemma according to Theorem 1.

Lemma 2: Let the distance between the cluster center of ΔS_{iy} be c_y and the center of a cluster C^p be p. If $dist(c_y, p) > radius_y + radius_p + maxdknn_p$, then for any point $s' \in \Delta S_{iy}$, we have $dist(s', r) > dknn_r$, $\forall r \in C^p$.

Proof: As shown in Fig. 1, for any point $s' \in \Delta S_{iy}$, we have $dist(s', p) \geq dist(c_y, p) - radius_y$ (based on the triangle inequality). Because of the condition $dist(c_y, p) > radius_y + radius_p + maxdknn_p$, we can get $dist(s', p) > radius_p + maxdknn_p$. Therefore, according to Theorem 1, we can get $dist(s', r) > dknn_r$, $\forall r \in C^p$.

Then, we can find a subset R_y which consists of only those clusters C^p satisfying

$$dist(c_y, p) \leq radius_y + radius_p + maxdknn_p \quad (2)$$

for each partition ΔS_{iy} according to Lemma 2. We can guarantee that the subset R_y consists of all the RkNN results for ΔS_{iy}.

2) Data processing: The enhanced solution also consists of two MapReduce jobs.

1) **First MapReduce job.** The first MapReduce job is to perform partitioning on R and ΔS_i and to collect some statics for each partition.
Specifically, each mapper loads the selected pivots into main memory, computes the distance between each point and all the pivots, and assigns the point to the nearest pivot. The mapper outputs each point with its clustering id, original dataset name (R or ΔS_i), the distance to its closest pivot. If a point is from R, there is an extra column of $dknn$ value for the point.
The reducer's job is to merge the statistic from the input data splits. For data points from ΔS_i, the reducers summarize the points in each cluster, the total number of points in the cluster, the radius of the cluster (the maximum distance between the pivot and the points in it). For points from R, the reducers summarize the $maxdknn$ value for each cluster in addition.

Algorithm 1 Map$< k1, v1 >$

```
1: if k1 ∈ ΔS_i then
2:    partitionID ⟵ ClusterID(k1)
3:    output(partitionID, < k1, v1 >)
4: else
5:    C^p ⟵ PivotID
6:    for each cluster ΔS_iy do
7:       y ⟵ partitionID of ΔS_iy
8:       if C^p satisfies Equation 2 then
9:          output (y, < k1, v1 >)
10:      end if
11:   end for
12: end if
```

2) **Second MapReduce job.** The second MapReduce performs the RkNN join. The mapper is to find a subset of R_y for each partition ΔS_{iy}. The reducer performs RkNN according to Lemma 1 by directly computing distance between each $s' \in \Delta S_{iy}$ and $r \in R_y$ to identify whether r is one of the RkNN results of s'.
The algorithm on the mapper is shown in Algorithm 1. If the input point is from ΔS_i, a new key-value pair is generated with the partition id as the key. If the point is from R, we find the pivot p that it belongs to and search for all the partitions ΔS_{iy} that satisfy Equation 2. The partition id of each ΔS_{iy} is set as the key of the output.
In the reducer, for each point s' in ΔS_{iy}, we first prune those clusters in R according to Theorem 1. For the rest clusters C^p, we directly compute the distance between s' and each $r \in C^p$, and identify the RkNN results of s' based on Lemma 1. The algorithm on reducers is shown in Algorithm 2.

3) Cost Analysis: In the first MapReduce job, the reducers merge the statistics from the input data splits, thus the shuffling cost is $|R| + |\Delta S_i|$. Let the number of pivots be N_p. In the mappers we compute the distance between the input points and the pivots, and find the nearest pivot for each points. The computational cost is $b_0 N_p(|R| + |\Delta S_i|)$ for a constant b_0. In the second MapReduce job, the shuffling cost is $|\Delta S_i| + \alpha|R|$, where α denotes the average replicas of each point in R. Let the number of partitions be N. As we need to find a subset R_y for each partition ΔS_{iy} and compute the distance between each point in R_y and ΔS_{iy},

Algorithm 2 Reduce$< k2, v2 >$

```
parse R_y and ΔS_iy from all the < k2, v2 >
for each s' ∈ ΔS_iy do
    for each C^p ∈ R_y do
        if dist(s', p) ≤ radius_p + maxdknn_p then
            for each r ∈ C^p do
                if dist(s', r) < dknn_r then
                    add r into s''s RkNN results
                end if
            end for
        end if
    end for
end for
output(s',RKNN(s'))
```

the computational cost is about $b_1N^2 + b_2\alpha|R||\Delta S_i|/N$ for some constants b_1, b_2. Thus, in the enhanced solution, the shuffling cost is $2|\Delta S_i| + (1+\alpha)|R|$, and the computational cost is about $b_0N_p(|R| + |\Delta S_i|) + b_1N^2 + b_2\alpha|R||\Delta S_i|/N$.

D. Compute the kNN join results in time t_i

After we get the RkNN result $RKNN(s')$ for each point s' in ΔS_i, we can easily update the kNN join results of time t_{i-1}. Suppose the kNN results for each query in R are maintained in a ranking list. We compute the distance between each query point in $RKNN(s')$ and s', and if it is smaller than the $dknn$ value of the query point, we add s' into its kNN results list and update the $dknn$ value as well. The procedure ends when we go through all the RkNN results for all the points in ΔS_i. In this way, the updated kNN join results must be the actual kNN join results in time t_i. Apparently, our method is more efficient than the naive solution in which we need to compute the distance between all points in R and S_i, and sort the distances to get the kNN join results in every snapshot t_i.

V. Experimental Results

We conduct experiments to validate the efficiency of our proposed methods on snapshot kNN join processing in MapReduce. We compare our methods of the **basic solution (BS)** and **enhanced solution (ES)** to the **naive solution (NS)**, which is to re-compute the kNN join results at each snapshot using PGBJ [4], an existing kNN join methods on static datasets in MapReduce. The experiments are on a cluster of 8 Dell R210 severs with Gigabit Ethernet interconnect. Each node has a 2.4GHz Intel processor and 8GB of RAM, running the 64-bit Ubuntu 10.10 OS and Hadoop 2.6.0. Each node is configured to run one map and one reduce task.

Our experiments are conducted on a real dataset [15] which is used to predict the forest cover type. The dataset contains 581,012 objects with 54 attributes (10 integer, 44 binary). We use the 10 integer attributes in the experiments.

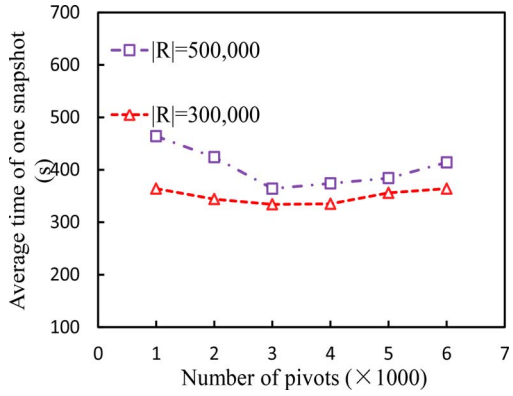

Figure 2. Processing time w.r.t number of pivots

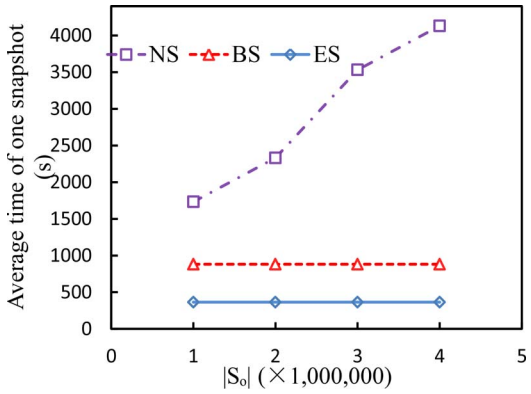

Figure 3. Processing time w.r.t size of $|S_0|$

To evaluate the performance on large datasets, we expand the dataset to 6,000,000 points using similar methods to the work of Vernica et al. [14] which maintains the same distribution to the real world dataset. We randomly select points from the dataset and set the default query set size to be 500,000, i.e., $|R| = 500,000$; the initial dataset size to be 4,000,000, i.e., $|S_0| = 4,000,000$. For simplicity, we set the same number of updated points between any two adjacent snapshots and the default value is 200,000, i.e., $|\Delta S| = 200,000$. The default value of k is 10. We evaluate the kNN join processing time for the next 10 snapshots from the initial time (i.e., $t_1, \cdots, t_{10}$). The performance is measured by the average processing time of one snapshot.

A. Effect of the pivot number

Fig. 2 shows the performance of the enhanced solution with respect to the pivot number. We find that the processing time drops by varying the pivot number from 1000 to 3000, but increases when varying the pivot number from 3000 to 6000. The reason is that when the number of pivots increases, the whole space will be split to a finer granularity. Thus, the pruning power of Theorem 1 will be enhanced and we can find a smaller R_y for each partition ΔS_{iy}. This leads to a reduction in both shuffling cost and computational cost

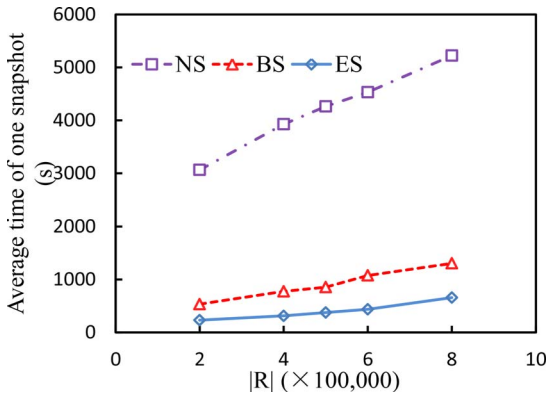

Figure 4. Processing time w.r.t size of query set

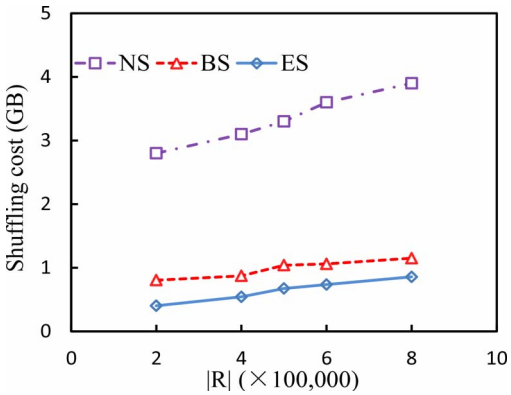

Figure 5. Comparison of shuffling cost

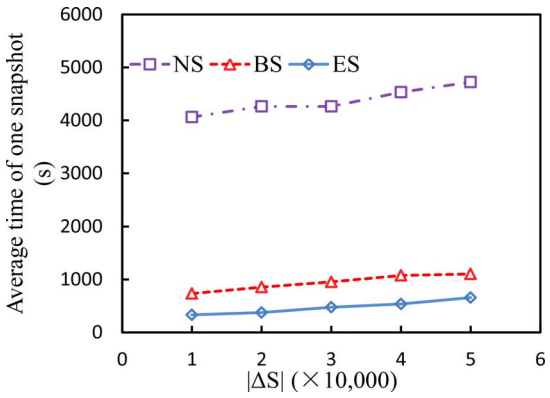

Figure 6. Processing time w.r.t number of updated points

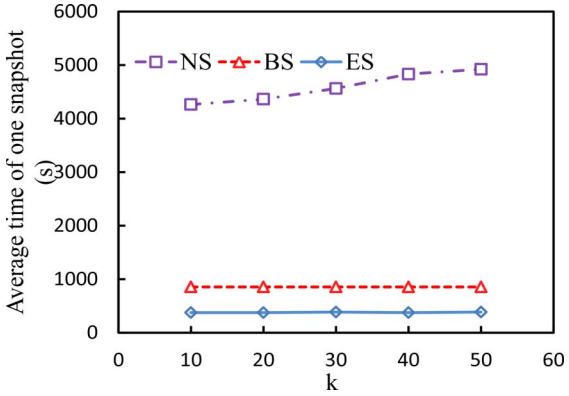

Figure 7. Processing time w.r.t k

according to the cost analysis in Section IV-C3. However, when the number of pivots becomes too large, the increasing of computational cost between points and the pivots in the first MapReduce job and the computational cost between partition centers and the pivots in the second MapReduce job becomes dominant. Thus, the curves go upward after the number of pivots reaches 3000. We choose 3000 as the default pivot number of the enhanced solution in the rest of experiments.

B. *Effect of the initial dataset size* $|S_0|$

We vary the initial dataset size $|S_0|$ from 1,000,000 to 4,000,000 and the running time is shown in Fig. 3. We can see that the running time of naive solution increases as the initial dataset size grows. The reason is that we have to repeatedly computing kNN join on the valid dataset S_i of each snapshot where S_i contains the unchanged part of data including points in S_0. While the basic solution and the enhanced solution performs RkNN join of ΔS_i on R, the two methods do not influenced by the size of S_0, so that the running time of BS and ES remains the same.

C. *Effect of the query set size* $|R|$

We vary the query set size from $200,000$ to $800,000$ and the results are shown in Fig. 4. We can see that the processing time of all the solutions grow with increasing $|R|$. The cost is roughly linear with respect to $|R|$ since each query point requires a single kNN processing. The basic solution and the enhanced solution incur less processing time than the naive solution as we do not have to repeatedly compute the distances between R and the unchanged part of data. The enhanced solution performs better than the basic solution for the reason of the adopted mapping strategy. Fig. 5 shows the comparison of shuffling cost. Clearly, the enhanced solution incurs the least shuffling cost, as the data partitioning strategy using cluster technique helps to reduce the size of intermediate data in MapReduce. Remember that in the enhanced solution only a subset of R_y is sent to the reducer of each partition ΔS_{iy}. Thus, the shuffling cost of the enhanced solution is less than that of the basic solution which has to send all points in R to the reducers of each partition ΔS_{iy}.

D. *Effect of the number of updated points* $|\Delta S|$

We vary the number of updated points between adjacent snapshots and the average processing time is shown in Fig. 6.

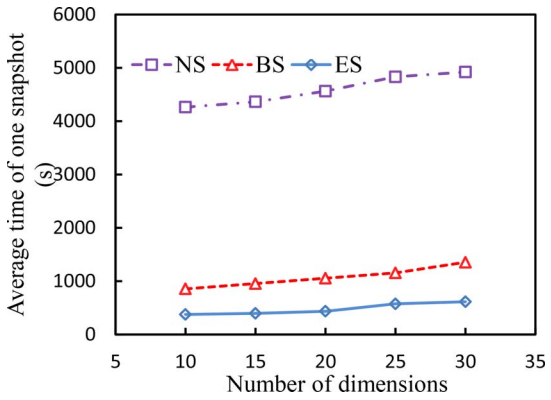

Figure 8. Processing time w.r.t number of dimensions

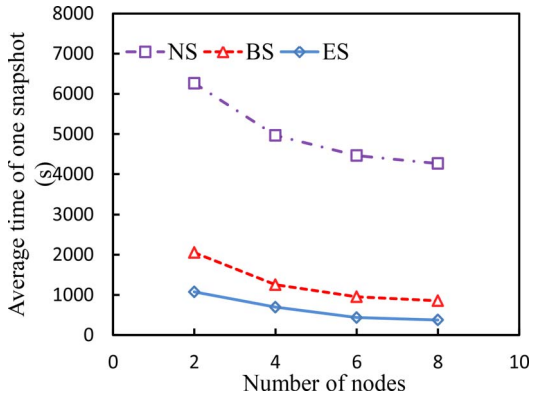

Figure 9. Scalability w.r.t number of nodes

The processing time of the three solutions increase as $|\Delta S|$ increases. The time of the naive solution increases because we need to process kNN join repeatedly on $|S_i|$ in each snapshot, where $|S_i|$ includes the updated point set, i.e., $S_i = S_{i-1} + \Delta S$. In the basic solution and enhanced solution we need to find the affected queries of each points in ΔS, thus the time of the two solutions increase linearly with increasing $|\Delta S|$.

E. Effect of k

We now study the effect of k on the performance of all the solutions. Fig. 7 shows the results by varying k from 10 to 50. The processing time of the basic solution and enhanced solution remains nearly the same. The main cost of the two solutions is to find the affected queries for the updated point. We can reuse the kNN join results of time t_{i-1} for time t_i which means the processing is insensitive to k. From Fig. 7 we can see that the processing of the naive solution is more sensitive to k as the shuffling cost increases with increasing k in the kNN join processing on static datasets which has been exploited in previous work [4].

F. Effect of dimensionality

To evaluate the performance on the dimensionality, we generate several random clustered datasets with different dimensionality using similar methods to the work of Chakrabarti et al. [16]. The default values are same to that of the expanded real-world dataset. The results of performance on the dimensionality are shown in Fig. 8. We can see that the processing time of all the solutions increase as the dimensionality grows. The reason is that the computational cost between points increases exponentially when the number of dimensions increases. We can see that the processing time of the enhanced solution increases slowest which means our proposed solutions incur the least computational cost, indicating that our methods are more robust.

G. Scalability w.r.t number of nodes

We now measure the effect of the number of computing nodes. From Fig. 9 we can see that all methods enjoy a decrease in the processing time with more nodes being employed. Notice that all the methods cannot speed up linearly with the number of nodes because the shuffling cost will be increased with more nodes being involved. Clearly, the enhanced solution has well scalability and performs much better than the baseline methods.

VI. Conclusion

We address the snapshot kNN join problem on data streams in this paper, which appears in a wide range of real-life applications. Existing kNN join solutions mainly solve kNN join problem on static datasets, or on a single centralized machine, which are difficult to scale to large data on data streams. In this paper, we propose an approach to solve the snapshot kNN join problem incrementally. We generate the kNN join results of time t_i by updating the join results of time t_{i-1}. The intuition is that the data on the data stream are continuously generated, which means typically there is $S_i = S_{i-1} + \Delta S_i$, where S_i is the valid data to be queried in time t_i and S_{i-1} is the valid data in t_{i-1}. ΔS_i denotes the updated points between time t_{i-1} and t_i. As ΔS_i is often much smaller than S_i, we propose to firstly find the queries in the query set R whose kNN results can be affected by the updated points in ΔS_i, and then update the kNN results of these small part of queries respectively. In this way, we can avoid computing the kNN join results on the whole dataset S_i. We propose a basic implementation of searching for affected query points using MapReduce to scale to large volume of data. Furthermore, we analyse the cost in the processing and present the enhanced strategies of data partitioning and grouping to reduce the shuffling cost and computational cost. Our experiments on real-world datasets demonstrate that our proposed methods are efficient, robust, and scalable.

REFERENCES

[1] C. Bohm, F. Krebs, The k-nearest neighbour join: Turbo charging the kdd process, KAIS, 2004, pp. 728-749

[2] C. Xia, H. Lu, BC. Ooi, J. Hu, Gorder: An efficient method for knn join processing, VLDB, 2004, pp. 756-767

[3] C. Yu, B. Cui, S. Wang, J. Su, Efficient index-based knn join processing for high-dimensional data, Information and Software Technology, 2007, pp. 332-344

[4] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, Efficient processing of k nearest neighbor joins using MapReduce, VLDB, 2012, pp. 1016-1027

[5] C. Zhang, F. Li, and J. Jestes, Efficient parallel knn joins for large data in MapReduce, EDBT, 2012, pp. 38-49

[6] A. Stupar, S. Michel, and R. Schenkel, Rankreduce-processing k nearest neighbor queries on top of MapReduce, LSDS-IR, 2010

[7] WH. Tok, S. Bressan, Progressive and approximate join algorithms on data streams, Advanced Query Processing, 2013, pp. 157-185

[8] C. Yu, R. Zhang, Y. Huang, H. Xiong, High-dimensional knn joins with incremental updates, Geoinformatica, 2010, pp. 55-82

[9] C. Yang, X. Yu, Y. Liu, Continuous KNN Join Processing for Real-Time Recommendation, ICDM, 2014, pp. 640-649

[10] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, SIGMOD, 2010, pp. 201-212

[11] Y. Tao, D. Papadias, X. Lian, X. Xiao, Multidimensional reverse knn search, VLDB, 2007, pp. 293-316

[12] M. Zhang, R. Alhajj, Effectiveness of NAQ-tree in handling reverse nearest-neighbor queries in high-dimensional metric space, KAIS, 2012, pp. 307-343

[13] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, OSDI, 2004, pp. 137C150

[14] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce, SIGMOD, 2010, pp. 495-506

[15] Covertype Data Set, 1998, https://archive.ics.uci.edu/ml/datasets/Covertype

[16] K. Chakrabarti, S. Mehrotra, Local dimensionality reduction: A new approach to indexing high dimensional spaces, VLDB, 2000, pp. 89-100.