GeoInformatica manuscript No.
(will be inserted by the editor)

High-dimensional kNN Joins with Incremental Updates

Cui Yu · Rui Zhang · Yaochun Huang · Hui Xiong ·

Received: date / Accepted: date

Abstract The k Nearest Neighbor (kNN) join operation associates each data object in one data set with its k nearest neighbors from the same or a different data set. The kNN join on high-dimensional data (*high-dimensional kNN join*) is an especially expensive operation. Existing high-dimensional kNN join algorithms were designed for static data sets and therefore cannot handle updates efficiently. In this article, we propose a novel kNN join method, named *kNNJoin*$^+$, which supports efficient incremental computation of kNN join results with updates on high-dimensional data. As a by-product, our method also provides answers for the reverse kNN queries with very little overhead. We have performed an extensive experimental study. The results show the effectiveness of kNNJoin$^+$ for processing high-dimensional kNN join queries in dynamic workloads.

Keywords query optimization · storage & access · optimization and performance

1 Introduction

Modern database applications, such as multimedia, bioinformatics, and time series analysis, need powerful database query tools [16,3]. The data objects in these applications are typically represented by high-dimensional vectors. A common operation performed on the data is to find objects similar or close to a given object, which

Cui Yu
Monmouth University, West Long Branch, NJ 07764, USA
Tel.: +1-732-571-4460
Fax: +1-732-263-5202
E-mail: cyu@monmouth.edu

Rui Zhang
University of Melbourne, Carlton Victoria 3053, Australia
E-mail: rui@csse.unimelb.edu.au

Yaochun Huang
University of Texas - Dallas, Dallas, TX 75080, USA
E-mail: yxh038100@utdallas.edu

Hui Xiong
Rutgers, the State University of New Jersey, Newark, NJ 071012, USA
E-mail: hxiong@andromeda.rutgers.edu

translates to a k nearest neighbor (kNN) query in high-dimensional spaces. Extensive studies [15, 13, 5, 7] have attacked the kNN query. However, no technique solves the "dimensionality curse" [4, 19] due to the intrinsic difficulty of processing the kNN query in high-dimensional spaces. Consequently, the kNN query remains one of the most expensive operations in high-dimensional databases.

Recently Böhm and Krebs [6] proposed the *k-nearest neighbor join* (*kNN join*) operation, which is a single-run operation to facilitate a large number of single kNN query processes. Such one-time computation of the nearest neighbors for all data objects can dramatically decrease the average processing time of a single kNN query compared to the traditional single-query-based approach. Indeed, the kNN join can be implemented as a primitive operator to support various applications [9, 11, 6, 21, 23], such as k nearest neighbor classification, k-means clustering, sample assessment, sample post-processing, missing value imputation, and k-distance diagrams (a graph linking each point of a data set to its k nearest neighbors). These applications almost cover all stages of knowledge discovery process [6].

While the kNN join improves the per query performance, the one-time join computation is expensive, especially for high-dimensional data sets. The kNN joins can also be used for low-dimensional data. For instance, one-NN join on two-dimensional data was studied in [14]. This article, however, focuses on high-dimensional data with updates. We refer to the problem of the kNN join on high-dimensional data as the *high-dimensional kNN join*. All existing high-dimensional kNN join algorithms [6, 21, 23] were designed for static data sets. If there is an update[1], the expensive kNN join computation has to be performed again. This deficiency restricts the use of the kNN join in practice since updates are usually unavoidable. For example, sampling-based data mining methods often require a kNN join between the set of sample points and the original data set to assess the quality of the sample [6]. In an environment where the data sets are not static, the kNN join may be performed periodically to provide accurate measure for the quality of the sample points, which is inefficient. Our technique always provides the correct kNN join result as the data objects get updated. Many of these data mining tasks involve high-dimensional data from various sources such as medical data, scientific observations and multimedia data. Other data mining applications requiring kNN joins described earlier may also have the need for updatable data sets, and therefore will benefit significantly from our technique. Consequently, high-dimensional kNN join with updates is a compelling problem.

In this article, we address the above problem and make the following contributions:

– We propose kNNJoin$^+$, a technique to process the kNN join query on high-dimensional data with updates. The main idea is to maintain a carefully designed kNN join table which can be updated efficiently when updates on the data set happen.
– We propose optimization strategies for improving the efficiency of basic update operations on the kNN join table exploiting *distance based pruning* and *shared query processing*.
– We introduce the *RkNN join* operation, which is a counterpart of the kNN join operation with regard to Reverse kNN queries. Interestingly, while providing the kNN join result dynamically, kNNJoin$^+$ can also produce the RkNN join result dynamically with little additional computational cost. As a result, both kNN queries and RkNN queries can be answered efficiently.

[1] In this paper, an update means an insertion, a deletion or a change to an existing data object.

– Through extensive experiments on both synthetic and real-world data sets, we show the effectiveness of kNNJoin$^+$ for processing high-dimensional kNN join queries in dynamic workloads. We observe a performance improvement of two orders of magnitude compared to a naive way of updating kNN join results by traditional techniques.

The remainder of this article is organized as follows. Section 2 introduces the kNN and RkNN join operations. Section 3 reviews related work. Section 4 describes the kNNJoin$^+$ data structures and algorithms. Section 5 presents the experimental results. Finally, Section 6 concludes the article.

2 Problem Definition

This section gives the definition of the kNN join operation and introduces the reverse kNN (RkNN) join operation. For the rest of this article, R and S denote the two data sets in a kNN join. If R=S, then the join is a self-join. We assume a metric space and every object is represented by a multidimensional point in the space. Therefore, we also refer to an object as a point when the context is clear.

2.1 The kNN Join Operation

The kNN query is defined as follows. Given a query point p, an integer k and a set S, the *kNN query* returns the set, $kNN(p) \subset S$, that contains k points from S and satisfies: $\forall q \in kNN(p), \forall q' \in S - kNN(p) : \|p - q\| \leq \|p - q'\|$.

The *kNN join* of R and S, denoted by $R \times_{kNN} S$, produces a full collection of k nearest neighbors from S for each data point of R. It is formally defined as follows.

Definition 1

$$R \times_{kNN} S = \{(p, kNN(p)) | p \in R \wedge kNN(p) \subset S\},$$

It is easy to observe that the result of a kNN join is asymmetric. If $R = S$, the kNN join is a *self kNN join*. For static join methods, such as MuX [6], Gorder [21] and iJoin [23], there is no difference between a self-join and a join of two different data sets. However, when joins are processed dynamically in an incremental manner with one insertion after another, self-joins are more complicated and more costly per update. The reason is that an insertion to R or S has individual impacts on the join result. For a self-join, one insertion is causes “two” insertions: one to R and another to S. It needs to deal with the impact of insertions to both of the “two” data sets (it is the same for deletion – one deletion causes “two”). Therefore, while self-join is a special case of *kNN Join*, it is generally the most expensive form of *kNN join* operation. This becomes even clearer when the details of the update operations are explained in Section 4.2.

2.2 The RkNN Join Operation

The **RkNN join** of R and S, denoted by $R \times_{RkNN} S$, produces a full collection of lists with each point q of S and the points from R that have the point q as one of their k nearest neighbors. It is formally defined as follows.

Definition 2

$$R \times_{RkNN} S = \{(q, RkNN(q)) | q \in S \wedge$$
$$RkNN(q) \subset R \{\forall p \in RkNN(q) : p \in R \wedge q \in kNN(p)\}\},$$

where $RkNN(q)$ denotes the list of reverse k nearest neighbors of q. In another words, $RkNN(q)$ is the result of RkNN query of q.

If R=S, RkNN join is *self RkNN join.* The RkNN query is more expensive than the kNN query even for low-dimensional data. The relationship between kNN and RkNN is not symmetric and the number of RkNNs can't be predicted. Current work for RkNN query processing is mostly limited to RNN queries [22] or low-dimensional data [17], where the RNN query is a special case of the *RkNN query* when $k = 1$ [22]. The generalized RkNN query was introduced in [2], which allows the value of k to be specified at query time as long as k is smaller than a preset number k_{max}. Our proposed kNNJoin$^+$ also supports such flexible RkNN queries. The RkNN query is useful for many applications such as decision support systems [18,2].

In analogy to the kNN join operation with regard to kNN, the RkNN join operation greatly improves the performance of RkNN queries. As we will see, while kNNJoin$^+$ provides the kNN join result dynamically, it can also produce the RkNN join result dynamically almost for free.

3 Related Work

The *kNN join* was initially proposed as an alternative of indexing to support fast kNN queries because kNN queries are expensive to process, especially for high-dimensional data sets. There are not many techniques for high-dimensional kNN joins; MuX [6], Gorder [21] and iJoin [23] are the recently proposed kNN join algorithms for high-dimensional data.

MuX uses an index nested loop join to process the kNN join query. The index, called a *multi-page index (Mux)*, is essentially an R-tree [10] based structure employing large-sized pages (the hosting page) to keep I/O time low. It also uses a secondary structure of buckets, which are smaller minimum bounding rectangles, to partition the data with finer granularity for lower CPU cost. Each set of objects (R and S) is organized in a multi-page index. In the index nested loop join, MuX iterates the index pages on R; for the pages of R held in main memory, pages of S are retrieved through the index and searched for k nearest neighbors. MuX terminates when all pages of R and S are either visited or pruned. MuX optimizes performance using page-loading and bucket-selecting strategies.

Gorder[21] is a method based on block-nested loop join. It exploits a number of techniques to achieve high performance. First, it sorts data points of R and S based on a grid-ordering, during which principal component analysis and transformation are performed. Then it conducts scheduled block nested join, during which data sets are partitioned into blocks consisting of several physical pages, and the data loaded in main memory are partitioned again into smaller blocks for a similarity join.

The iJoin [23] is a kNN join approach that exploits the data partition feature of iDistance [24] on join data sets. Conceptually, iJoin is also an index nested loop join. For each data point in R, its kNN candidates are found from the same or most similar partitions of S, based on the positions of the partition reference points. As a result, iJoin can efficiently filter out false join candidates during join processing. The iJoin method has a basic version and two enhanced versions. One applies dimensionality reduction to decrease disk I/O and CPU cost. The other version uses approximation cubes to reduce unnecessary kNN computations on real feature vectors. Both of the enhancements add an extra level in the iDistance tree structure.

MuX, Gorder and iJoin all utilize the nested loops join strategy, which requires scanning the whole dataset. Obviously, if there is any update on either of the joining sets, the nested loops join has to be performed again to obtain the new result. There is no straightforward way to extend any of these techniques to process the kNN join incrementally in face of updates. In summary, they are designed for static data sets and cannot handle dynamic data sets without re-computation of the whole kNN join result.

4 The ***kNNJoin***$^+$ Technique

For static kNN join methods, re-computing the whole join result is necessary for each new insertion, deletion or change of a data point. Even if delayed modification is permitted in some cases, a whole re-computation is still needed for each time of batch processing. Since high-dimensional kNN joins are expensive, static kNN join strategies are usually not practical for many real databases that typically grow with new insertions and also allow deletions and changes. (Changing of an existing object can be handled as a deletion of the old object and then an insertion of the object with updated information.) For the databases that frequently gets new insertions and deletions, incremental kNN joins are essential. In other words, it is important to have a kNN join technique that can build kNN join result on the existing result, rather than re-compute everything.

Motivated by those facts, we design kNNJoin$^+$, a new kNN join method for high-dimensional data. It can dynamically join the similar objects together, keep the whole join result updated upon each new insertion, and allow data objects to be dynamically deleted from the existing join result. The idea of kNNJoin+ is applicable to data of any dimensionality, although its strength is more noticeable for high-dimensional data.

The join result of kNNJoin$^+$ includes a kNN join table and an RkNN join table. They are always updated and available for quick and accurate kNN and RkNN queries. RkNN join tables are not mandatory for kNNJoin$^+$; it is an option to support quick RkNN queries.

Some notations used in this section are given in Table 1.

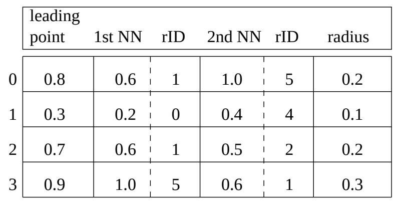
4.1 Data Structures

The kNNJoin$^+$ is a dynamic kNN join scheme for high-dimensional data. We propose to maintain the kNN join results in two carefully designed *join tables* (kNN join table and RkNN join table) for incremental join processing. In particular, for $R \times_{kNN} S$, we maintain a *kNN join table* for R. If the RkNN join result ($R \times_{RkNN} S$) is desired, we also maintain an *RkNN join table*. Both tables are updated dynamically for each insertion and deletion. Updates on join tables are efficient because only the rows that need to be updated are accessed for processing; there are no unnecessary reads regardless the size of join result. In other words, I/O cost is not an issue.

Table 1 Some Notations.

R, S	data sets
k	the number of nearest neighbors
o, p, q, q', x, y,	data points
NN_i	the ith nearest neighbor
rID	the sequence number of a data point in its data set
rID_{NN_i}	the sequence number of the ith NN in its data set
$order$	the sequence ordering number as a NN
r	radius
$R \times_{kNN} S$	kNN join of R and S
k'	the sequence number as a NN for a reverse nearest neighbor point
RNN_j	the jth reverse nearest neighbor
rID_{RNN_j}	the sequence number of point RNN_j in its data set
$R \times_{RkNN} S$	RkNN join of R and S
$more$	indicator of overflow
$kNN(p)$	the k nearest neighbors of p
$RkNN(p)$	the points that have p in their k nearest neighbors list
$T1$	kNN join table
$T2$	RkNN join table

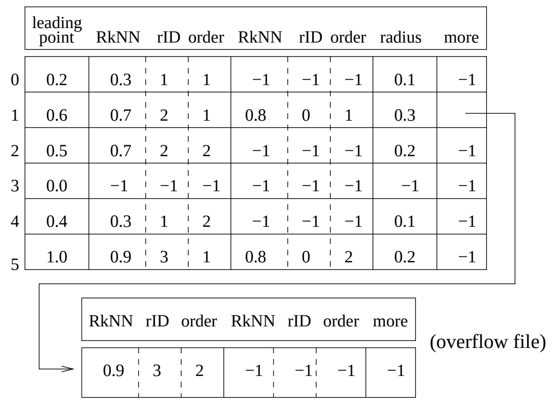
Maintaining the join tables requires considerable space ($\mathcal{O}(kn)$, where n is the cardinality of R in case of the kNN join table or the cardinality of S in case of the RkNN join table). Maintaining kNN join result is a requirement of the kNN join operation. The basic idea is to trade space for high efficiency. In another words, the space cost is intrinsic to the kNN join operation. As we will see below, with better access convenience, these join tables of kNNJoin$^+$ are however not much bigger than the basic join results of kNN join operations in terms of storage space. Since the space has to be used whenever presenting the join result, the space usage of the kNNJoin$^+$ is acceptable.



	leading point	1st NN	rID	2nd NN	rID	radius
0	0.8	0.6	1	1.0	5	0.2
1	0.3	0.2	0	0.4	4	0.1
2	0.7	0.6	1	0.5	2	0.2
3	0.9	1.0	5	0.6	1	0.3

Fig. 1 A kNN join table (k=2)

The table structures are designed to support fast access needed for join processing, and yet be as simple as possible. For ease of illustration, we use a simple example with R = { 0.8, 0.3, 0.7, 0.9 } and S = { 0.2, 0.6, 0.5, 0.0, 0.4, 1.0 }, where only one-dimensional data points are used. Figure 1 shows the kNN join table with k=2. The kNN join table contains a collection of kNN join rows. Each row has the form $\{o, \langle NN_i, rID_{NN_i}\rangle, radius\}$. o is a data point of R, called the *leading point* to differentiate it from its neighbor points from S. $\langle NN_i, rID_{NN_i}\rangle$, where $i = \{1, 2, \ldots, k\}$, is the list of o's k nearest points from S. NN_i means the i-th NN of o. rID_{NN_i} is the point's row number in S. $radius$ is the distance between point o and its k-th nearest point, called the *kNN boundary sphere radius* of o, or simply the *kNN sphere radius* of o.



	leading point	RkNN	rID	order	RkNN	rID	order	radius	more
0	0.2	0.3	1	1	−1	−1	−1	0.1	−1
1	0.6	0.7	2	1	0.8	0	1	0.3	
2	0.5	0.7	2	2	−1	−1	−1	0.2	−1
3	0.0	−1	−1	−1	−1	−1	−1	−1	−1
4	0.4	0.3	1	2	−1	−1	−1	0.1	−1
5	1.0	0.9	3	1	0.8	0	2	0.2	−1

(overflow file)

RkNN	rID	order	RkNN	rID	order	more
0.9	3	2	−1	−1	−1	−1

Fig. 2 An RkNN join table (k=2)

With the same sample, Figure 2 shows the corresponding RkNN join table. The RkNN join table contains a collection of RkNN join rows. Each row has the form of $\{o, \langle RNN_j, rID_{RNN_j}, k'\rangle, radius, more\}$, where $j = \{1, 2, \ldots, k\}$. o is a leading point from S. $\langle RNN_j$,rID_{RNN_j}, $k'\rangle$ is the list of its reverse neighbor points from R. RNN_j is the j-th reverse neighbor point from R. rID_{RNN_j} is the row number of RNN_j in R. k' means o is the k'-th nearest neighbor of point RNN_j. For example, if point o has two reverse nearest neighbors: x (o is the 3rd NN of x) and y (o is the 1st NN of y), the reverse neighbor list of o will be $\{\langle x, rID_x, 3\rangle$, $\langle y, rID_y, 1\rangle\}$. *radius* is the distance between o and its furthest RkNN point, and is called *RkNN boundary sphere radius*, or simply the *RkNN sphere radius*. The *RkNN sphere radius* is not necessary for join processing, so it can be omitted. We use it to indicate when a point o has no RkNN points by setting it to -1.

The number of RkNN points for a point is unpredictable. Overflow nodes may be used when a leading point has more than k RkNN points; *more* is the pointer to an overflow node. When there is no overflow node, *more*'s value is -1. The structure of an overflow node is $\{\langle RNN_j, rID_{RNN_j}, k'\rangle, more\}$. All overflow nodes are kept in a single separate file.

For self kNN join, to minimize searching time, each point o has the same row number in kNN join table and RkNN table. That is, if o is the leading point in 568th row in the kNN join table, it is also the 568th point in RkNN table.

The kNNJoin$^+$ exploits high-dimensional indexing to support kNN queries during the join processing. We chose iDistance [24] because it is efficient and easy to implement in real systems. The main idea of iDistance is transforming high-dimensional data into single values (iDistance values) based on each point's relative distance to its closest reference point, and indexing the iDistance values in a B$^+$-tree. The data space is partitioned into clusters because each partition has its own iDistance value range. Figure 3 illustrates how a kNN query on iDistance is answered by expanding the search area in the each effected data partition, which is simplified into range searches on the leaf-node level of the base B$^+$-tree.

The kNNJoin$^+$ uses an auxiliary structure, named *Sphere-tree*, built on R to facilitate searching for all the possible RkNNs of a new point to be inserted into S. The Sphere-tree organizes data points and their RkNN sphere based on coverage. For opti-

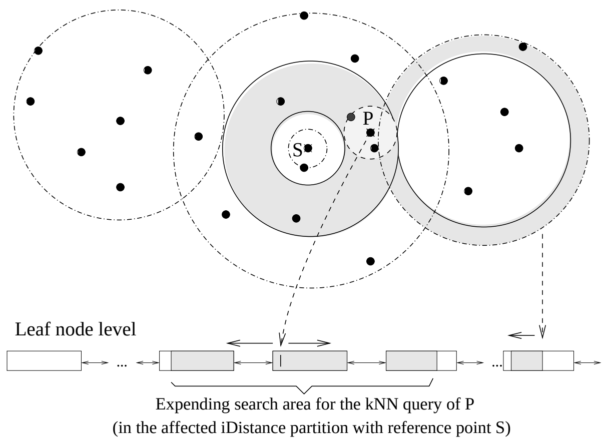

S
P
Leaf node level
Expending search area for the kNN query of P
(in the affected iDistance partition with reference point S)

Fig. 3 A kNN query using iDistance

mization, the Sphere-tree does not index the RkNN spheres of the points in S; instead, it indexes the kNN spheres of the points in R according to Lemma 1.

Lemma 1 *A new point p inserted to S will become a new kNN point of a point o from R, if and only if p is within the* kNN sphere *of o.*

When a new point is added to S, all the data points of R that will have the new point as one of their kNNs need to be found. In other words, the "to-be" RkNNs of the new point need to be found before the new point is inserted to join tables. Since the new point hasn't been inserted yet, there is no RkNN sphere radius of the new point available. But we have the following observations. First, every data point o in R has its own kNNs. Second, any new point inserted to S, if it is within the *kNN boundary sphere* of o, will become a new kNN point for o. So, for $R \times_{kNN} S$, it is optimized to build Sphere-tree indexing the kNN spheres of points in R. Any point whose kNN boundary sphere covering the new point are exactly the points that will be the RkNN points of the new point. No points will be missing and no redundant points will be unnecessarily checked.

An example is given in Figure 4. It shows the kNN spheres of o_1 and o_3 cover the new point p_{new}, so the kNN list of o_1 and o_3 will be updated. The kNN spheres of o_2 and o_4 don't cover the new point, so their kNN lists are not affected and won't be checked. This proof and example work for general joins and self-joins. (In a self-join, S is also R.)

The Sphere-tree is quite similar to the M-tree [8]. It is organized like an R-tree [10], but handling spheres instead of rectangles. Figure 5 shows a simple example. In leaf nodes, each entry is $\langle o, rID, r \rangle$ representing a sphere with center point o and radius r. rID is the row position of o in the join table. In non-leaf nodes, each entry contains a sphere and a pointer pointing to a child node. This sphere, represented by a center point and a radius, covers (bounds) all the spheres in its child node. We name it *covering sphere*. A *covering sphere* is incrementally generated, by including the spheres from its child node one by one. (We follow the strategy of R-tree to pick a *covering sphere* for a new sphere.) A *covering sphere* can be generated incrementally using the function in Figure 6, which shows how the center point and radius of a *covering sphere* are updated

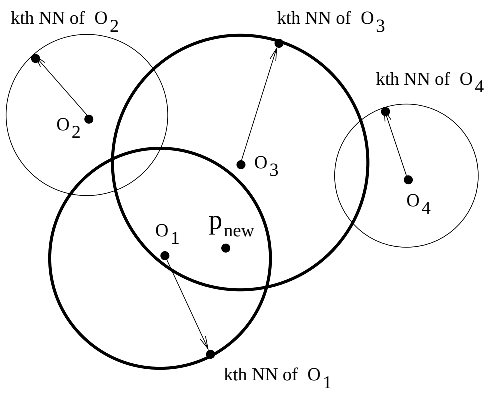


Fig. 4 Points with kNN spheres covering a new point need to update their kNN lists

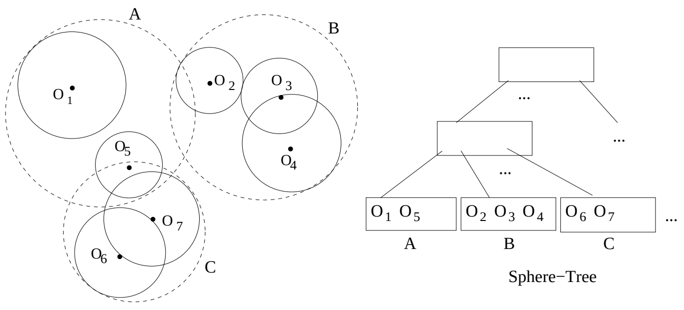


Fig. 5 An example of the Sphere-tree

with a new sphere added in its child node. The main difference between the M-tree and the Sphere-tree is that the leaf nodes of the M-tree contains data objects while those of the Sphere-tree contains $\langle o, rID, r\rangle$ entries. The insertion and deletion operations of Sphere-tree are very similar to the M-tree and therefore we omit the details here.

```
Algorithm CoveringSphere
Input: new sphere $O$, current covering sphere $C$
Output: updated covering sphere $C$
1    if $C$ covers $O$
2        return $C$;
3    else if $O$ covers $C$
4        center point of $C \leftarrow$ center point of $O$;
5        radius of $C \leftarrow$ radius of $O$;
6    else
7        center point of $C \leftarrow$ the middle point of $O$ and $C$;
8        radius of $C$ = (radius of $C$ + radius of $O$ +
                 distance between center of $O$ and center of $C$)/2;
9    return $C$;
End CoveringSphere;
```

Fig. 6 Updating a covering sphere.

In summary, we use four data structures for the dynamic kNN join processing: the kNN join table, the RkNN join table, an iDistance and a Sphere-tree. The design of kNNJoin$^+$ has primarily two goals: efficiency and feasibility. We designed the table structures to be as simple as possible for good capacity and easy access; we picked iDistance for good performance and easy implementation; and we also designed Sphere-tree to be simple and helpful for searching RkNN points.

4.2 Algorithms for Insertions

The kNNJoin$^+$ is a new kNN join method that can dynamically produce kNN join tables and RkNN join tables. For $R \times_{kNN} S$, there are three cases of join updating upon insertions. Deletions will be discussed in the next subsection.

Case 1: A *self kNN join* implies each point in the data set will be combined with k nearest neighbors in the same data set. In this case, there is only one data set used for the join operation. Inserting p requires a single *kNN query* to find $kNN(p)$ and a single *RkNN query* to find $RkNN(p)$, where $kNN(p)$ are all the kNN points of p and $RkNN(p)$ are are all the points that will use p as one of their kNNs. Point p and its $kNN(p)$ are appended at the end of the kNN join table. p is also added as a new neighbor point to the kNN list of every point in $RkNN(p)$. At the same time, their old k-th nearest neighbors are eliminated from their kNN lists. If the RkNN join table is desired, reflect each change in the RkNN join table. — If, in the kNN join table, p is added as new i-th neighbor of point o and its old k-th neighbor x is eliminated, then, in the RkNN join table, o is added as a new RkNN point of p and x is removed.

Case 2: Inserting a new point q in S for $R \times_{kNN} S$, where $R \neq S$. In this case, the procedure becomes slightly easier than in Case 1. First find $RkNN(q)$ from R, who will have the new point q as one of their new kNN points. Then add q to the kNN list of every point in $RkNN(p)$, with their old k-th neighbors removed at the same time. If the RkNN join table is desired, each change in the kNN join table requires an update in the RkNN join table. Case 2 uses only the Sphere-tree for R.

Case 3: Inserting a new point p in R for $R \times_{kNN} S$, where $R \neq S$. In this case, updating the KNN join table and RKNN join table is much easier than in Case 1. First, find p's k nearest neighbors from S. Second, append the combination of p and its $kNN(p)$ to the end of the kNN join file. If the RkNN join table is being used, add p into the reverse neighbors lists of the points in $kNN(p)$. Case 3 uses the iDistance index for S.

From Case 2 and Case 3, we can see, if both R and S ($R \neq S$) get new point insertions, we need to have the iDistance index on S and the Sphere-tree for R.

Case 1, the *self kNN join*, is the most complicated and expensive of the three cases. For that reason, our study focuses more on *self kNN join*.

Next, we present the kNNJoin$^+$ insertion algorithms. Upon each new insertion, both the iDistance and the Sphere-tree remain available and updated during join processing. Because a RkNN join table is optional, the algorithm steps (with an asterisk on the line number) that are only for the RkNN table can be omitted when there is no RkNN join table.

Figure 7 gives the kNNJoin$^+$ algorithm for Case 1: a new point p is inserted to R, $R = S$. $T1$ and $T2$ respectively represents the kNN join table and the RkNN join table. In this algorithm, steps 3,4,7,9,10 and 11 are needed only for the RkNN table. It is obvious and hence not specified in the algorithm, whenever a data point gets its

```
Algorithm $kNNJoin^{+}_{1}$
Input: current $T1$ and $T2$, new point $p$ to $R$ and $S$ ($R$=$S$)
Output: updated $T1$ and $T2$
1       find $kNN(p)$;
2       append $p$ and its $kNN(p)$ to $T1$;
3*      append $p$ and an empty RkNN list to $T2$;
4*      add $p$ to RkNN lists of points in $kNN(p)$ (in $T2$);
5       find $RkNN(p)$;
6       for each point $o$ in $RkNN(p)$
7*          $x \leftarrow$ the old $k$-th NN of $o$;
8           add $p$ into the kNN list of $o$ (in $T1$);
9*          add $p$ to the RkNN list of $o$ (in $T2$);
10*         add $o$ into the RkNN list of $p$ (in $T2$);
11*         remove $o$ from the RkNN list of $x$ (in $T2$);
End $kNNJoin^{+}_{1}$;
```

Fig. 7 kNNJoin^{+} Algorithm for Case 1.

k-th NN changed, it is necessary to modify its kNN sphere radius in the kNN join table. As described in the previous section, row numbers ($rIDs$) are stored together with the data points in the join tables. For example, when the points of $RkNN(p)$ are found, their positions in the kNN join table are known too. For another example, if the old k-th NN of o is found, its position in RkNN table is known. In other words, once a point is found, updating its kNN or RkNN list in either table requires no extra searching cost. As a result, join processing is highly optimized, and the RkNN join table is maintained almost for free, since no extra searching cost is needed.

For Case 1, finding $kNN(p)$ in Step 1 and finding $RkNN(p)$ in Step 5 can be done quickly because the iDistance indexing tree is used to find the kNN points of the new point and the Sphere-tree is used to find the points having the newly inserted point as a kNN point.

During join processing, if new point is inserted to S, iDistance tree is updated; if new point is inserted to R, Sphere-tree is updated with the new insertion. Therefore, self-join requires (1) inserting the new point into the iDistance tree; (2) inserting the new point and its kNN sphere radius to Sphere-tree. None of them are computationally complicated.

```
Algorithm $kNNJoin^{+}_{2}$
Input: current $T1$ and $T2$, new point $q$ to $S$
Output: updated $T1$ and $T2$
1*      append $q$ and an empty RkNN list to $T2$;
2       find $RkNN(q)$;
3       for each point $o$ in $RkNN(q)$
4*          $x \leftarrow$ the old $k$-th NN of $o$;
5           add $q$ into the kNN list of $o$ (in $T1$);
6*          add $o$ into the RkNN list of $q$ (in $T2$);
7*          remove $o$ from the RkNN list of $x$ (in $T2$);
End $kNNJoin^{+}_{2}$;
```

Fig. 8 kNNJoin^{+} Algorithm for Case 2.

Figure 8 gives the kNNJoin$^+$ algorithm for Case 2: a new point is inserted to S, $S \neq R$. In this algorithm, steps 1, 4, 6 and 7 are only needed for the RkNN table. For Case 2, the Sphere-tree indexes the kNN spheres of the points of R.

```
Algorithm $kNNJoin^+{}_3$
Input: current $T1$ and $T2$, new point $p$ to $R$
Output: updated $T1$ and $T2$
1       find $kNN(p)$;
2       append $p$ and its $kNN(p)$ to $T1$;
3*      add $p$ to RkNN lists of points in $kNN(p)$ (in $T2$);
End $kNNJoin^+{}_3$;
```

Fig. 9 kNNJoin$^+$ Algorithm for Case 3.

Figure 9 gives the kNNJoin$^+$ algorithm for Case 3: a new point is inserted to R, $R \neq S$. In this algorithm, step 3 is needed only for the RkNN join table. Unlike Case 1 and Case 2, Case 3 does not need to use a Sphere-tree.

4.3 Algorithms for Deletions

In this section, we present the deletion algorithms, which can dynamically delete any data point from the existing join result. (A deletion does not simply remove all unnecessary data points. It is more of a disjoin, the inverse operation of join.) For $R \times_{kNN} S$, there are three cases of deletions to consider. We will discuss each of them, but focusing more on the complicated cases.

Case 1: Delete a data point from *self kNN join*, $R \times_{kNN} S$, where $R = S$.

A straightforward way to do a deletion for *self kNN join* involves four main steps. Suppose a point p is deleted from data set R. (1) Make a point query to find point p in the kNN join table and delete its row. (2) With the same row number, get p's RkNN list, $RkNN(p)$, from RkNN join table and delete the row. (If the RkNN join table doesn't exist, the Sphere-tree can be used to find all the points with kNN spheres covering the point p.) (3) For each point in $RkNN(p)$, do a kNN query[2] and update its kNN list in the kNN join table. (4) For each update made in kNN join table, make a corresponding update in the RkNN join table, if the RkNN join table exists. Steps (3) and (4) can be done together as one step. Apart from those four major steps, the index structures used, e.g. the iDistance tree and the Sphere-tree, should also be updated.

Removing a row from either table only requires setting a deleted indication; for example, set the first dimension of the data point to be -1. Deleted rows can be used by the next new insertions. Most databases grow, so the space used by deleted rows should not be an issue. If there are both frequent insertions and deletions, a simple space management strategy can be applied. For instance, keep a record of unused rows for new insertions.

In this straightforward way, a deletion requires mainly one point query and possibly some kNN queries as well. A point query requires low cost. These kNN queries are considerably expensive, even though each of these kNN queries for a kNN join deletion

[2] Here kNN query only needs to find a new k-th NN.

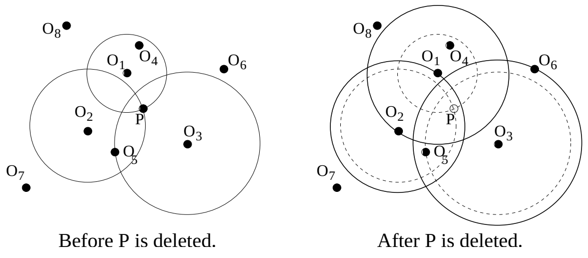


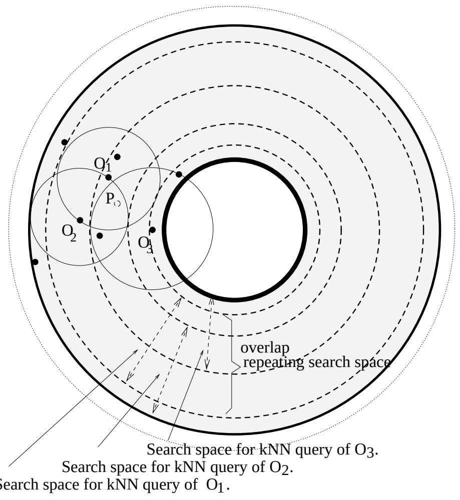
Fig. 10 Deletion of p (k=2)

just finds its new k-th NN and its current (k-1)-th NN distance can be used as pruning distance to start the search. Because the search region spans in iDistance are centered at the reference points of the affected iDistance partitions, not centered at kNN query points, the kNN queries for kNN join deletion can't find their k-th NN by simply increasing original the annulus excluding the area where the $k-1$ NNs reside.

For each data point deletion, the number of kNN queries depends on the number of RkNN points of the data point deleted; this number is unpredictable. If the point is not in any kNN list of any other data point, no kNN query is needed and deletion can be done quickly. If the point only has a small number of RkNN points, deletion cost won't be high if an index (e.g. iDistance index) to support kNN queries is available. While it is almost certain that one dynamic deletion is faster than re-performing a join operation, it is still necessary to consider the worse case. For example, there could be many deletions issued in short period time and each point to be deleted might have a long list of RkNN points. Optimization is needed.

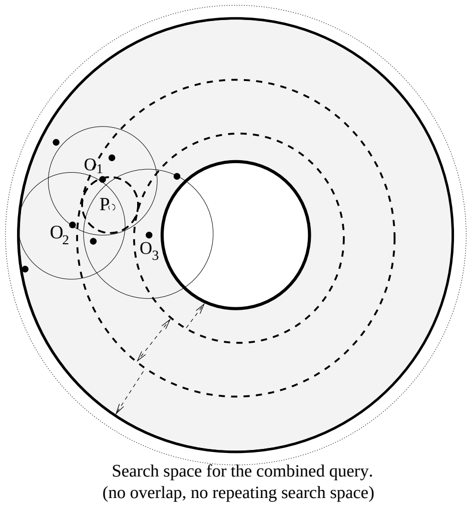
We propose an optimization strategy exploiting *shared query processing* based on the fact that neighbors are often close to each other. When a data point p is deleted, its RkNN points will be affected; and these RkNN points are most likely (as experimentally proved) surrounding p. So, a deletion of p is like that the center point (not exactly at the center) of those points is removed. Each of the RkNN points needs to find just one new neighbor to complete their kNN lists. Figure 10 gives a 2-dimensional example. Points o_1, o_2 and o_3 are affected by the deletion of point p. Each of them needs to find a new k-th nearest neighbor to complete its kNN list.

The *shared query processing* described here is unique, because it exploits both the neighboring feature and the iDistance structure. It is different from other shared computation techniques [12,25]. As illustrated in Figure 11 (with only one iDistance partition space shown), multiple kNN queries for all the affected points repeatedly process overlapping search spaces, causing repeated page accesses. To reduce the cost, we use one single combined and expandable similarity query to find all the new neighbors for all the affected points. As shown in Figure 12, the query starts as a similarity query of p, with starting radius of its kNN sphere radius. The query then expands its search sphere to find new neighbors for all the affected points o_1,o_2 and o_3. To avoid unnecessary search, we move the query center when a point finds a possible new k-th NN with the least kNN radius increase. When a point has found its accurate new k-th nearest neighbor, it is eliminated from the combined query. The query ends when all the points of $RkNN(p)$ are guaranteed to have found their accurate new k-th NNs. A movable and expandable similarity query with *shared query processing* can be easily and efficiently done with the support of an iDistance structure, where expanding query space is in the unit of leaf-node along the leaf-node level [24]. A point is guaranteed to



O1
P
O2
O3
overlap
repeating search space
Search space for kNN query of O3.
Search space for kNN query of O2.
Search space for kNN query of O1.

Fig. 11 Deletion of p, using multiple kNN queries (only one iDistance partition shown)



O1
P
O2
O3
Search space for the combined query.
(no overlap, no repeating search space)

Fig. 12 Deletion of p, using one combined kNN query (only one iDistance partition shown)

have found its kNNs with 100% accuracy if the kNN sphere of the point is covered by the search space of an iDistance tree [24].

The optimized deletion algorithm for Case 1 is shown in Figure 13.

Case 2: Delete a data point from S for $R \times_{kNN} S$, where $R \neq S$.

With a deletion of q from S, kNNJoin$^+$ does the following two steps. (1) Find the points that have q in their kNN lists. On the RkNN join table, do a point query of q to locate all its RkNN points and delete the row at the same time. If no RkNN join table is available, the Sphere-tree on R can help to find the RkNN points. (2) Update the kNN lists of the points in $RkNN(q)$. If no point in R uses q as a neighbor point, nothing further needs to be done. Otherwise, kNN queries needs to be conducted for the RkNN points of q. The iDistance tree on S can be helpful for those kNN queries.

```
Algorithm kNNJoin+_del
Input: current T1 and T2, the point p to be deleted
Output: updated T1 and T2

1     make a point query to find p and get its rID;
2     get kNN(p) and radius at the row rID in T1;
3     delete row rID in T1;
4     L ← RkNN(p) at row rID from T2;
5     delete row rID in T2;
6     if L is empty, goto the end.
7     else // L contains a list of o_i, where i=1, 2, 3, ...
8         start combined similarity query of p;
9         do {
10            expand query space;
11            for each point o in L
12                if o finds its exact new k-th NN, q
13                    insert p as k-th NN of o in T1;
14                    add o to RkNN list of q in T2;
15                    eliminate o from L;
16            if point o in L finds a possible new k-th
              NN with least radius increase
17                move the query center as o;
18        } while (L is not empty);
End kNNJoin+_del;
```

Fig. 13 The optimized deletion for self-join.

If many kNN queries are needed, an optimized query with *share query processing* can save execution cost. Apart from those two steps required for updating the join tables, the iDistance tree should also be updated.

Case 3: Delete a data point p from R for $R \times_{kNN} S$, where $R \neq S$.

The kNNJoin$^+$ processes a deletion of p from R with two steps. (1) Use a point query to find p and its kNN list in the kNN join table and delete the row. (2) For each point in $kNN(p)$, locate its row in RkNN join table and remove p from its RkNN list. If the RkNN join table does not exist, nothing further needs to be done. Apart from those two steps required for updating the join tables, the Sphere-tree should also be updated. No kNN queries are needed for this case. The deletion algorithms for Cases 2 and 3 are simpler than that of Case 1. We omit the detailed algorithms for Case 2 and 3. However, in terms of computational complexity, as long as the iDistance index on S is available, Cases 1 and 2 are similar. Our experiment also suggests it. Case 3 is the simplest of the three cases.

4.4 Correctness of kNNJoin$^+$

Lemma 2 *All algorithms proposed for kNNJoin$^+$ produce the correct join results for all insertion and deletion cases.*

Proof: Here we consider insertion Case 1. We prove the correctness of the kNNJoin$^+$ algorithm for an insertion to a self kNN join by mathematical induction.
First, for the data set with initially $n = k + 1$ data points, all the points are k nearest neighbors (kNNs) of the other points. The algorithm is trivially correct for the initial

state. Next, assume the algorithm is correct for the current data set with $n = m$ data points, where $m > k+1$. Now we need to prove that the algorithm is still correct when $n = m+1$. For the new point, p_{m+1}, the algorithm finds the k nearest neighbors by the iDistance tree, and then adds p_{m+1} and its kNN list to the kNN join table. Because the iDistance returns the correct kNNs of p_{m+1}, p_{m+1} can also be correctly added as an RkNN point of the kNN points of p_{m+1}. For every other point q, we have:

1. if p_{m+1} is one of the kNN points of q,
   $\Rightarrow$ p_{m+1} is closer to q than the current k-th nearest neighbor of q;
   $\Rightarrow$ p_{m+1} should appear in the kNN sphere of q;
   $\Rightarrow$ (a) p_{m+1} will be put as a kNN point of q, and the new kNN sphere radius of q will be set as distance(q, the new k-th NN of q); (b) since only the old k-th nearest neighbor is removed, the other $k-1$ nearest neighbors has been reserved and p_{m+1} has been set as a new kNN point of q;
   $\Rightarrow$ therefore, the correct kNN points of q is reserved.
2. if p_{m+1} is not a new kNN point of q,
   $\Rightarrow$ distance(p_{m+1}, q) $>$ distance(q, the current k-th NN of q);
   $\Rightarrow$ p_{m+1} does not appear in the kNN list of q;
   $\Rightarrow$ the kNN set of q is not changed;
   $\Rightarrow$ therefore, the correct kNN set of q is reserved.

By 1. and 2., we know that all the existing points will keep their correct kNN sets. As a result, it is proved that the algorithm can get the correct KNN join table for all the points with new insertions. Finally, since we update the RkNN join table based on the correct changes on the kNN table, the join algorithm also result in the correct RkNN join table. □

Similarly, we can prove the correctness of kNNJoin$^+$ algorithms for other insertion cases and the deletion cases.

5 Experimental Study

In this section, we present an experimental study that demonstrates the effectiveness of the proposed kNNJoin$^+$ technique.

All the experiments were performed on a 2.66G HZ Pentium 4 machine with 1 Gigabyte main memory running the Linux Redhat operating system. The synthetic data sets are of 20, 30, 40 and 50-dimensional uniformly distributed data. The real-life KDDCUP data are from the UCI Data Repository [1]. The KDDCUP data set contains 42-dimensional data points. In the experiments, we used various number of data points up to 400,000. Without loss of generality, we use the Euclidean distance as the distance function between two points, although any metric distance function can be used. The default k for kNN and RkNN is 10. The node size of the iDistance tree and the Sphere-tree is 4 KB. The iDistance trees we built use 100 reference points.

To the best of our knowledge, there is no existing dynamic join methods for high-dimensional data that we can compare with. Therefore, we conducted experiments to evaluate the insertion performance of kNNJoin$^+$ by comparing with the following alternatives: (1) sequential scan, which is often one of the only choices for high-dimensional kNN queries due to the “curse of dimensionality”; (2) “iDistance only”, which exploits only the iDistance tree to speed up the searches of k nearest neighbors for new insertions; (3) “Sphere-tree only”, which uses only the Sphere-tree to speed up the searches of the points that will change their kNN lists because of new insertions; (4) Gorder and iJoin, which are recently proposed static kNN join methods. Both Gorder and iJoin outperform MuX, so we chose to omit it. Because a large number of continuous insertions are the worst-case scenario for dynamic join, we measure the join cost for various numbers of insertions (up to 1000) for testing.

We also evaluated the deletion performance of kNNJoin$^+$ via experimental analysis and comparison with Gorder and iJoin, both of which need to re-compute join results.

For a kNN join, I/O time (around 1% of total time) is much less significant than CPU time [21,23], so we measure the total execution cost (elapsed time) for most of our experiments.

Our experimental study of kNNJoin$^+$ has focused on self-join because it is more expensive to process dynamically. We first investigate how kNNJoin$^+$ behaves under various parameters. Then we also study the general case, that is, $R \neq S$ for all the operations and the results are reported accordingly.

5.1 The Effect of Number of Insertions

We begin by first looking at the effect of a large number of data insertions. We measure the total join cost needed to update the kNN join table and RkNN join table. Figure 14 shows the examples using real-life data sets, receiving a sequence of 100 to 1000 insertions. The kNNJoin$^+$ is shown by far more efficient than the sequential scan. The performance gap becomes larger when the number of insertions increases. In other words, we can observe the improvement of the performance gain of kNNJoin$^+$ when insertion volume grows. Such observations are consistent for data sets with different (20,000 and 70,000) existing number of data points already joined. Sequential scan performs poorly because each insertion requires a distance computations with every existing point. For example, when there are 20,100 existing data points joined, there will be 20,100 distance computations for the next coming data point. Therefore, we omit sequential scan for comparison in the later experiments.

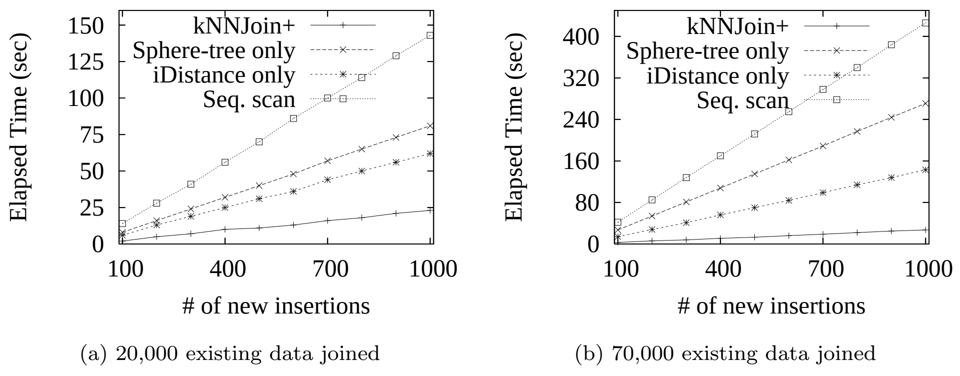


Fig. 14 Effect of number of continuous insertions (real-life data set).

Figure 14 shows: (1) Both “iDistance only” and “Sphere-tree only” are faster than sequential scan; both iDistance and Sphere-tree can speed up join processing, and so make kNNJoin$^+$ superior for kNN join updates. (2) “iDistance only” is faster than “Sphere-tree only”, which means the iDistance contributes more than the Sphere-tree to kNNJoin$^+$ efficiency.

We also tested kNNJoin$^+$ for the effect of number of continuous insertions by using larger synthetic data sets that contain 100,000 to 400,000 20-dimensional existing data points. Figure 15 shows a linear increase of execution time when there are more insertions.

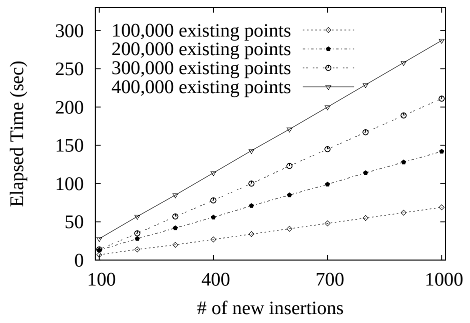


Fig. 15 Effect of number of continuous insertions (20-d synthetic data).

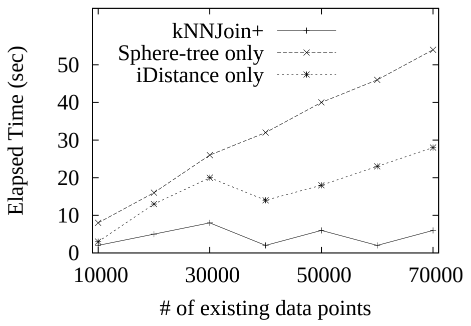
5.2 The Effect of Existing Data Set Size

Figure 16(a) shows the effect of existing data set size for 200 continuous insertions using real-life data. The kNNJoin$^+$ is fastest and only has minor change of execution cost with the growth of data set size. This experiment used real-life data sets, which may be skewed or clustered, resulting in some varying execution time.

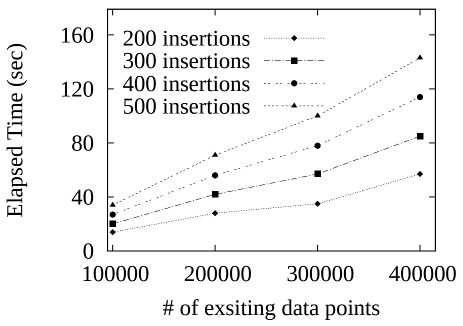
To further measure kNNJoin$^+$ scalability, we used 20-dimensional synthetic data sets with 100,000 to 400,000 existing data points for testing. Figure 16(b) shows the increase of the execution time is much slower than the growth of data set size.

Next, we compare kNNJoin$^+$ with Gorder and iJoin. For Gorder, we set the number of segments per dimension to 100, which ensures fine granularity for optimization. Also, we use a sub-block size big enough for 42-dimensional data points, which is within the best setting range recommended in [21]. For iJoin, we use its basic join algorithm, because the effect of the enhancements proposed for iJoin are from the improvement of the iDistance index. Since kNNJoin$^+$ used the original iDistance, for fair comparison, we don’t apply these enhancements for testing. In this experiment, kNNJoin$^+$ incrementally updates both kNN join table and RkNN join table for new insertions; Gorder and iJoin produce just one kNN join table, but they re-compute it once for each batch of new insertions.

Our experiments show the huge benefit we gain from the incremental updates for kNN join provided by kNNJoin$^+$. Figure 17 shows the performance comparisons of kNNJoin$^+$ for 500 insertions, kNNJoin$^+$ for 1000 insertions, one single static Gorder



(a) 42-d real-life data, 200 new insertions



(b) 20-d synthetic data

Fig. 16 Effect of existing data set size.

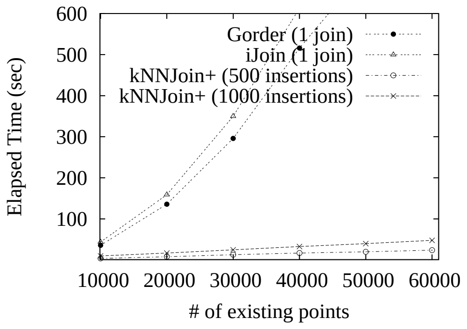


Fig. 17 Comparison with Gorder and iJoin (40-d synthetic data).

join, and one single static iJoin join, using 40-dimensional synthetic data. The same test on the real-life data set has similar result, so the figure is omitted here.

5.3 The Effect of Parameter k

In this subsection we study the effect of parameter k on kNNJoin$^+$ performance. We test the join cost for 200 continuous insertions, when the join tables already having 70,000 to 100,000 existing data points joined. Figure 18 shows the increase of execution time with the larger value of k. When the existing data set size is smaller, the increase is slower. This result is as we expected because larger k means more nearest neighbors need to be found for each new data point; and more points need to be identified that will have the new data point as their nearest neighbors.

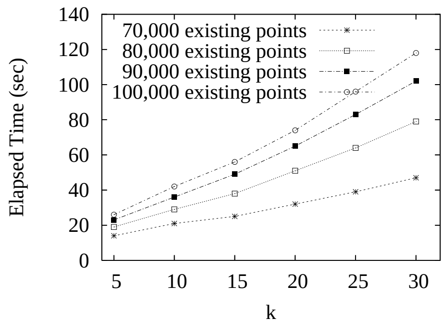
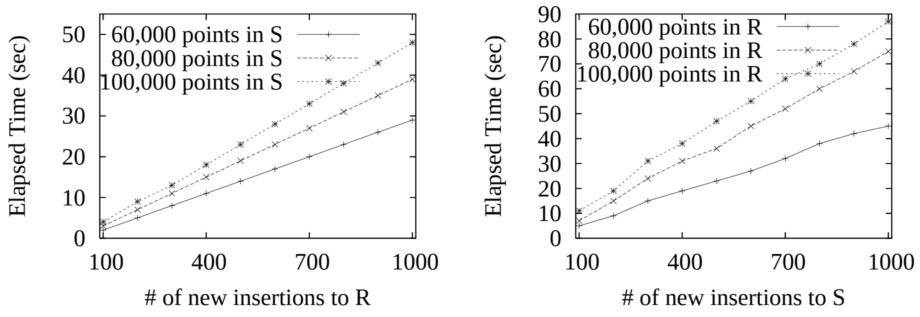


Fig. 18 Effect of k (40-d synthetic data; 200 insertions).

5.4 General kNN Join

In a general kNN join, R and S can be different. The properties of kNNJoin$^+$ make general kNN joins of two different data sets simpler than self-joins, and insertion Case 2 more complicated than insertion Case 3. Our experiment confirms these observations. Figure 19(a) shows the results of Case-3 insertions made to R when S is fixed for each test and Figure 19(b) shows the results of Case-2 insertions made to S when R is fixed for each test. The processing time of all tests gradually increases as the insertion number grows, and is consistently lower than the processing time of self-joins compared with similar tests for self-join. For example, let R and S ($R \neq S$) both have 100,000 40-dimensional synthetic points. When 1000 insertions are made to R, it takes 48 seconds to update the join result; when 1000 insertions are made to S, it takes 93 seconds. However, if $R = S$, 1000 insertions takes 213 seconds. If we compare Figure 19(a) and Figure 19(b), we can see insertions to S are more expensive than insertions to R. This behavior conforms to our analysis in section 4.2.



(a) Insertions to R, not S.

(b) Insertions to S, not R.

Fig. 19 General kNN join, $R \neq S$ (40-d synthetic data).

5.5 A Performance Analysis for Deletions

This experiment evaluates kNNJoin$^+$ deletion performance. When a data point is removed from a data set, it needs to be deleted (dis-joined) from the join tables. The deletion cost depends on the length of RkNN list of the data point to be deleted because only its RkNN points need to update their kNN lists. We use both real-life and synthetic data sets for testing to gather RkNN list length statistics.

The real-life data sets we used have 20,000, 50,000 and 70,000 42-dimensional data points. The experiments show that a high percentage of data points have no RkNN points. Less than 1% of the data points have more than 17 RkNN points when k=10, more than 11 RkNN points when k=5. Although some points have up to 26 RkNN points for k=10 and some points have up to 15 RkNN points for k=5, they are only 1 or 2 out of our testing data sets. Figure 20 shows statistics of real-life data, for k=5 and k=10. An interesting observation is that high percentage of data has 10 RkNNs for k=10 and 5 RkNNs for k=5 in our experiments.

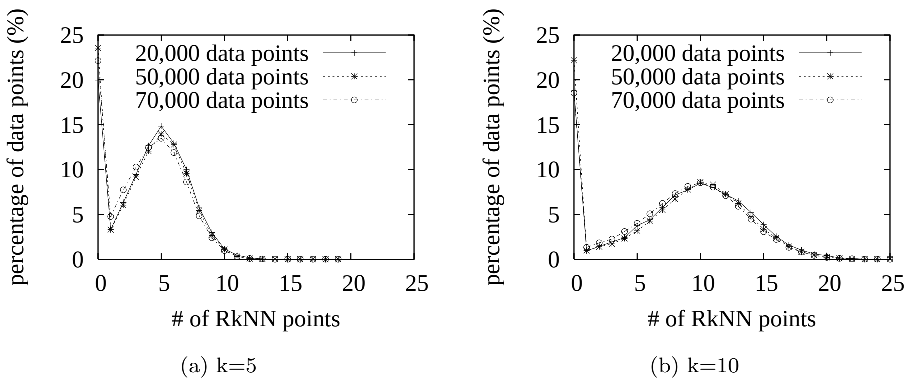


Fig. 20 Statistics for RkNN length (real-life data).

The synthetic data sets have 100,000 data points each, with different dimensionality (20, 30, 40 and 50). Our experimental results on these data sets show that over 81% of data points have fewer than 16 RkNN points. There are over 90%, 91%, 92% and 95% of data points have fewer than 25 RkNN points, for 50, 40, 30 and 20-dimensional data sets respectively. Figure 21 shows the percentage of data points with various RkNN lengths for k is 10. The highest percentage of 40 and 50-dimensional data has 2 RkNNs; the highest percentage of 30-dimensional data has 4 RkNNs; and the highest percentage of 20-dimensional data has 5 RkNNs.

To explain the distribution of the number of RkNNs, note the number of RkNNs characterizes the clustering of the points. If a point is close to a cluster, then it should be close to many points, which means it is more possible that this point is in the kNNs of other points; otherwise, if a point is an outlier, it is unlikely that the point is in the kNNs of other points. In short, the more clustered the data, the more the RkNNs. The real-life data sets are usually skewed and often have more clusters, therefore there

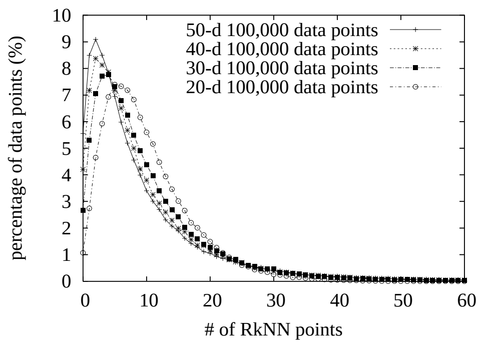


Fig. 21 Statistics for RkNN length (synthetic data; k=10).

are more points with large number of RkNNs compared to the synthetic uniform data sets. In all the uniform data sets, as dimensionality gets higher, the points become more equal-distant to each other. This is why more points in higher dimensional space have fewer RkNNs. As for the distributions of the RkNNs, it's not surprising that the curve looks like a Gaussian distribution. Since we look for kNN, on average, every point should have k RkNN. Therefore, the mean (expected number of RkNNs) of the bell-shaped curve is k for a Gaussian distribution. Compared to the synthetic uniform data sets, the real-life data sets fit better to the Gaussian distribution. The spike at 0 is due to the outliers. One possibility is correlation in the pseudo-random data.

We also did some experiments to evaluate kNNJoin$^+$ deletion performance with multiple deletions compared with one kNN join re-computation using Gorder and one kNN join re-computation using iJoin. Figure 22 shows the result using 40-dimensional synthetic data. Our deletion cost to disjoin the deleted points from join tables stays low and stable for data sets with various number of the existing data already joined. The same test on real-life data produced similar result.

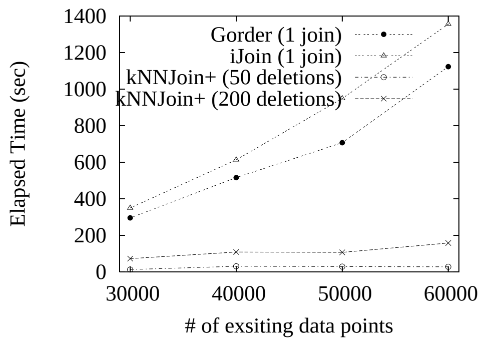


Fig. 22 Continuous deletion (40-d synthetic data).

5.6 The Impact of Shared Query Processing for Deletions

We apply *shared query processing* as an optimization strategy to improve the join deletion speed. In this experiment, we test its impact. As explained in Section 4.3, the cost of a join deletion for Case 1 and 2 depends on the number of RkNN points involved because each RkNN point causes a kNN query. If the number of RkNN points is high, a join deletion would be slow.

In this experiment we use 60,000, 70,000, 80,000, 90,000 and 100,000 40-dimensional synthetic data points for R and S. For Deletion Case 1, points are deleted from R, which is also S; for Deletion Case 2, points are deleted from S, $R \neq S$. Deletion Case 3 is not tested since it is simple and not relevant to this optimization. The value of k is 10. For each test, first, we experimentally pick 10 points from S, with approximately 10 RkNN points on average; then we process the 10 deletions with optimization and non-optimization. Figure 23 shows the *shared query processing* optimization can greatly improve the speed of these deletions. The speed-up is more than 10 times because, for each optimized deletion, one kNN query via the iDistance index and on average 10 updates on the Sphere-tree are done. For each non-optimized deletion, about 10 kNN queries via iDistance index and 10 updates on Sphere-tree executed alternatively. For non-optimized deletions, because the number of kNN queries for each deletions is around (not exact) 10, we can see some variation in elapsed time.

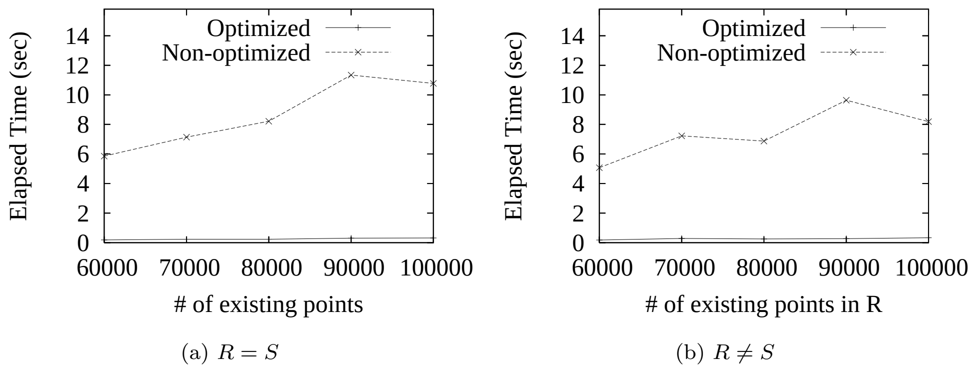


Fig. 23 Impact of Optimization for Deletions.

5.7 Pruning Power of Sphere-trees

The Sphere-tree is designed for kNNJoin$^+$ to speedup the searches of possible RkNNs for a new insertion during join processing, yet to be as simple as possible to keep the maintenance cost low. It works like an R-tree, but handles spheres. To evaluate the pruning power of Sphere-trees we did an experiment using 40-dimensional synthetic data for the case of $R \neq S$ and new points inserted into S. As explained in Sections 4.1 and 4.2, a Sphere-tree indexes the kNN spheres of points in R.

We used two testing groups. In one group, R has 100,000 existing data points, and S varied among 60,000, 70,000, 80,000, 90,000 and 100,000 existing data points. In the second group, S has 100,000 existing data points, and R varied among 60,000, 70,000, 80,000, 90,000 and 100,000 existing data points. The existing points are already joined. The experiment results depicted in Figure 24 show the number of spheres accessed for one insertion (as the average of 1000 insertions) and the total number of spheres in all levels of the Sphere-tree for each test. We observe: (1) the number of accessed spheres for an insertion and total number of spheres in a Sphere-tree are more when size of R is larger; (2) the size of S has little affect on both the number of accessed spheres for an insertion and the number of total spheres in a Sphere-tree; (3) consistently, around 6.1% spheres are accessed for one insertion in both groups of these tests. These observations can be explained by the fact that Sphere-tree is built with the kNN spheres of the points in R, and high percentage of points have relatively low number of RkNNs (see Section 5.5). In short, the pruning power of Sphere-tree is good and stable for kNNJoin$^+$.

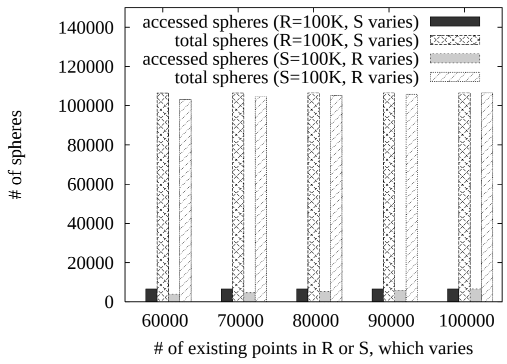


Fig. 24 Pruning Power of Sphere-tree.

6 Conclusions

The *kNN join* was initially proposed as a one-time-computation operation to support fast kNN queries. It is an excellent choice for high-dimensional kNN queries because most of the existing indexing techniques don't scale well and perform worse than sequential scan when dimensionality is high. However, all existing kNN join algorithms for high-dimensional data were designed for static data sets. In this article, we proposed a dynamic high-dimensional kNN join method, called kNNJoin$^+$, which enables efficient incremental updates on kNN join results. Each update can be completed within a quarter second for all our tests and only affects a minimal part of join results – the rows are necessary to be changed. As demonstrated by our experiments on both synthetic and real-world data sets, kNNJoin$^+$ is efficient and scalable for handling dynamic high-dimensional data. It is a promising supporting tool for advanced applications demanding fast kNN and RkNN queries.

References

1. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. *The UCI KDD Archive [http://kdd.ics.uci.edu].*
2. E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD'06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 515–526, 2006.
3. S. Berchtold and D. A. Keim. High-dimensional index structures database support for next decade's applications (tutorial). In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, page 501, 1998.
4. K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbor meaningful? In *Proceeding of the 7th International Conference on Database theory (ICDT)*, pages 217–235, 1999.
5. C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.
6. C. Böhm and F. Krebs. The k-nearest neighbor join: Turbo charging the kdd process. *Knowledge and Information Systems (KAIS)*, 6(6):728–749, 2004.
7. C. Böhm and H.-P. Kriegel. Dynamically optimizing high-dimensional index structures. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT)*, pages 36–50, 2000.
8. P. Ciaccia, M. Patella and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, 1997.
9. B. V. Dasarathy. Nearest neighbor (nn) norms - nn pattern classification techniques. IEEE Computer Society Press, 1991.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
11. J. Hartigan and M. Wong. A K-means clustering algorithm. In *Applied Statistics, 28*, pages 100–108, 1979.
12. X. Huang and C. S. Jensen and S. Saltenis. Multiple k Nearest Neighbor Query Processing in Spatial Network Databases. In *ADBIS '06: Proceedings of 10th East European Conference of Advances in Databases and Information Systems*, pages 266-281, 2006.
13. H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B^{+}-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.
14. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 201–212, 2000.
15. K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3:517–542, 4 1994.
16. D. Rafiei and A. Mendelzon. Querying time series data based on similarity. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):675–693, 2000.
17. Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 744–755, 2004.
18. Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 18(9):1239–1252, 2006.
19. R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, 1998.
20. R. Wong, Y. Tao, A. Fu, X. Xiao, A. Pryakhin, and M. Renz. On efficient spatial matching. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 579–590, 2007.

21. C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 756–767, 2004.
22. C. Yang and K. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 485–492, 2001.
23. C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based knn join processing for high-dimensional data. *Information & Software Technology*, 49(4):332–344, 2007.
24. C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 166–174, 2001.
25. R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple Aggregations Over Data Streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 299–310, 2005.
26. N. Koudas and K. C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 466-475, 1998.