Nearest Group Queries

Dongxiang Zhang, Chee-Yong Chan, Kian-Lee Tan
School of Computing, National University of Singapore
13 Computing Drive, 117417, Singapore
{zhangdo,chancy,tankl}@comp.nus.edu.sg

ABSTRACT

k nearest neighbor (kNN) search is an important problem in a vast number of applications, including clustering, pattern recognition, image retrieval and recommendation systems. It finds k elements from a data source D that are closest to a given query point q in a metric space. In this paper, we extend kNN query to retrieve closest elements from multiple data sources. This new type of query is named k nearest group (kNG) query, which finds k groups of elements that are closest to q with each group containing one object from each data source. kNG query is useful in many location based services. To efficiently process kNG queries, we propose a baseline algorithm using R-tree as well as an improved version using Hilbert R-tree. We also study a variant of kNG query, named kNG Join, which is analagous to kNN Join. Given a set of query points Q, kNG Join returns k nearest groups for each point in Q. Such a query is useful in publish/subscribe systems to find matching items for a collection of subscribers. A comprehensive performance study was conducted on both synthetic and real datasets and the experimental results show that Hilbert R-tree achieves significantly better performance than R-tree in answering both kNG query and kNG Join.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Algorithm, Performance

Keywords

kNG Query, kNG Join, Publish/Subscribe System, Hilbert R-tree

1. INTRODUCTION

k nearest neighbor (*k*NN) search [33] is an important and fundamental problem in many branches of computer science, including clustering, pattern recognition, image retrieval and recommendation systems. Given a query point q in a metric space, it finds k closest elements from a data source D. For example, Figure 1 shows the spatial distribution of two data sources R and S. R consists of western restaurants and S represents barber shops. If a user is located at point q and wants to find the nearest western restaurant, R_7 will be returned as it is nearest to q in data source R. If the user is interested to find the nearest barber shop, then S_2 is the result.

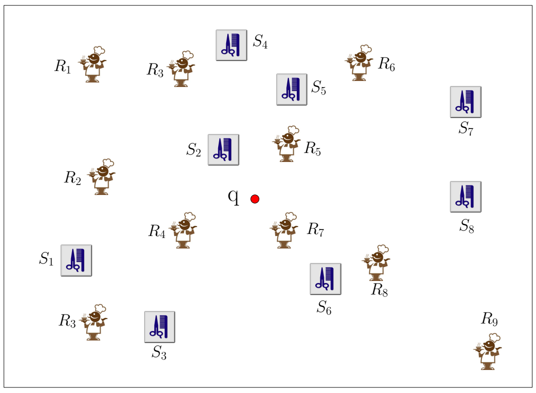


Figure 1: Spatial distribution of western restaurants (R_i) and barber shops (S_i)

In this paper, we extend *k*NN query to another type of useful query named *k*NG query. Unlike a *k*NN query, a *k*NG query finds k "closest" group of objects from multiple data sources, where each group consists of exactly one object from each data source.

DEFINITION 1 (*k*NG QUERY). *Consider a spatial database with m data sources* $D_1, D_2, \ldots, D_m$, *and let S denote a collection of groups of m objects where each group consists of one object from each data source; i.e.,* $S = \{\{o_1, o_2, \ldots, o_m\} \mid o_i \in S_i\}$. *Given a query object q, a kNG query returns a set of k groups* $S' = \{g_{i_1}, g_{i_2}, \ldots, g_{i_k}\}$, $S' \subseteq S$, *such that for any* $g_i \in S'$ *and* $g_j \in S - S'$, $\| q, g_i \| \leq \| q, g_j \|$, *where* $\| q, g \|$ *is some measure of the distance from query q to a group g.*

To calculate $\| q, g \|$ in a reasonable way, we take into account of both inner-group distance and inter-group distance. The inner-group distance (denoted by $\| q, g \|_{inner}$) captures the distance among the objects in group $g = \{o_1, \ldots, o_m\}$, while the inter-group distance (denoted by $\| q, g \|_{inter}$) represents the distance from the query object q to the objects in g.

Given a monotonic function $\mathtt{F}_1$ such that $\mathtt{F}_1(d_1,d_2) \leq \mathtt{F}_1(d_1',d_2')$ if $d_1 \leq d_1'$ and $d_2 \leq d_2'$, we can define

$$\| q,g \| = \mathtt{F}_1(\| q,g \|_{inner}, \| q,g \|_{inter}) \quad (1)$$

Similarly, we can define the inner-group distance and inter-group distance based on two monotonic functions $\mathtt{F}_2$ and $\mathtt{F}_3$ as follows:

$$\begin{aligned} \| q,g \|_{inner} &= \mathtt{F}_2(\| o_1,o_2 \|, \| o_1,o_3 \|, \ldots, \| o_{m-1},o_m \|) \quad (2) \\ \| q,g \|_{inter} &= \mathtt{F}_3(\| q,o_1 \|, \| q,o_2 \|, \ldots, \| q,o_m \|) \quad (3) \end{aligned}$$

Here, $\mathtt{F}_2$ takes $\frac{m(m-1)}{2}$ inputs and $\mathtt{F}_3$ takes m inputs. For example, given two groups $g = \{o_1,\ldots,o_m\}$ and $g' = \{o_1',\ldots,o_m'\}$, if $\forall 1 \leq i \leq m$, $\| q,o_i \| \leq \| q,o_i' \|$, we have $\| q,g \|_{inter} \leq \| q,g' \|_{inter}$. The distance between two objects $\| \cdot,\cdot \|$ can be defined in any metric space which satisfies the triangle inequality.

In this paper, we propose a general framework to process *k*NG queries with distance measures defined in terms of monotonic functions as described above. To make the presentation concrete and without loss of generality, we will assume the following distance measures in the rest of this paper. Note that other definitions of distance measures which satisfy the above monotonic constraints are also applicable.

$$\begin{aligned} \| q,g \| &= \| q,g \|_{inner} + \| q,g \|_{inter} \quad (4) \\ \| q,g \|_{inner} &= \max_{o_i,o_j \in g} \| o_i,o_j \| \quad (5) \\ \| q,g \|_{inter} &= \min_{o_i \in g} \| q,o_i \| \quad (6) \end{aligned}$$

Both $\| o_i,o_j \|$ and $\| q,o_i \|$ are calculated based on the Euclidean distance measure.

*k*NG query has a number of applications in location based services. For example, a local resident may want to find a western restaurant for dinner and a barber shop for a hair cut after his dinner. He prefers the two locations to be close to each other and to be near his house. If we use the spatial database in Figure 1 to answer this query, a group of two elements (S_2,R_5) will be returned when $k=1$. *k*NG is also useful in several other applications:

- **Location-Based Service**. When a user has a shopping list in which each item can be purchased from multiple candidate shops, he can utilize *k*NG query to select the nearest shops to cover all the items. Another example query is to find the nearest “gas station”, “barber shop” and “Burger King” so that the user can fill the car, get a hair cut and then have a quick lunch to save time.

- **Trip Planning**. Travellers are likely to have time budget when making a schedule. *k*NG query can be used to facilitate trip planning so that a visitor can participate in as many activities as possible. For example, a visitor may be interested to find “Sabarsky Cafe”, “Shopping Mall” and “Flea Market” around his hotel.

- **Collective Spatial Keyword Search**. This is an interesting application proposed recently in [6]. Given a set of keywords and a query location, it finds a group of geo-documents close to the query location and covering all the query keywords. We found that textual relevance was not taken into account in the ranking function defined in [6]. If only spatial distance is considered, this is essentially a *k*NG query.

In addition to *k*NG query, we also study another variant named *k*NG Join. This is similar to *k*NN Join with respect to *k*NN query. Given two datasets Q and D, a *k*NN Join query retrieves for each object in Q its k nearest neighbors in D. *k*NN Join is used in numerous applications including knowledge discovery [19], data mining [36] and geographical information systems [41]. Similarly, we define *k*NG Join as follows:

DEFINITION 2 (*k*NG JOIN). *Given a query set Q and m data sources $\{D_1,D_2,\ldots,D_m\}$, kNG Join finds k nearest groups for each query object $q \in Q$.*

Example *k*NG Join queries include

- For each primary school, find the nearest piano tutoring class and swimming pool so that the children have time to participate in both after class.

- Groupon subscribers may have common interest in taking a massage after dining at a Japanese restaurant. The publish/subscribe system can take advantage of *k*NG Join to notify each subscriber the closest Japanese restaurant and massage shop that are on sale based on their locations.

To efficiently answer *k*NG and *k*NG Join queries, we propose a best-first search strategy as our baseline algorithm, which is shown to be I/O optimal in answering *NN* query. It builds an R-tree [18] on each data source and traverses the trees in a top-down manner to enumerate the candidate node sets. These candidates are visited in increasing order of their distance to the query location. However, when m is large, a huge number of node sets at the higher levels of the R-trees will be generated, incurring very high CPU cost. To reduce the computational cost, we propose a solution using Hilbert R-tree [24]. The method utilizes the Hilbert value to guide the order of candidate enumeration so that promising node sets can be examined as early as possible. In summary, we make the following contributions in this paper:

1. We propose two new types of useful queries, named *k*NG query and *k*NG Join respectively.
2. We formally define the two types of queries and show that the problem of processing each type of queries is NP-hard.
3. We propose a best-first baseline algorithm using R-tree as well as an improved solution using Hilbert R-tree.
4. We present a cost model to estimate the CPU and I/O cost of *k*NG query.
5. Extensive experiments were conducted on both synthetic and real datasets to show the effectiveness of our proposed solutions.

The rest of the paper is organized as follows. In Section 2, we review various types of spatial queries that are related to *kNG* query and *k*NG Join. In Section 3, we show that each of the two query processing problems is NP-hard. The baseline algorithm using R-tree is proposed in Section 4. In Section 5, we present our improved method using Hilbert R-tree. The query cost model analysis is provided in Section 6. Results of an extensive experimental study on both synthetic and real datasets are reported in Section 7. Section 8 concludes the paper.

2. RELATED WORK

Intuitively, *k*NG query is proposed by extending *k*NN query to retrieve candidates from multiple data sources. *k*NG Join further extends *k*NG query from searching for one query point to searching for a bundle of points simultaneously. If we organize the related works based on these two dimensions (the number of data sources in a result *m* and the number of specified query points $|Q|$), we can obtain a framework as shown in Figure 2. In the following, we will introduce these queries as well as their solutions that are widely adopted.

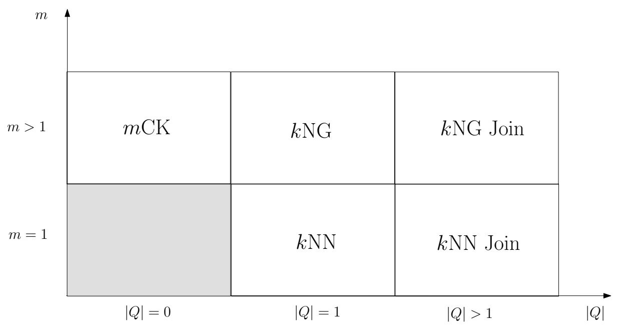


Figure 2: Related spatial queries

2.1 k Nearest Neighbor Query

Nearest neighbor search [33] is a well studied problem with numerous applications. It can be efficiently solved in low dimensional space [33, 16, 20]. The branch-and-bound R-tree traversal algorithm proposed in [33] provides suboptimal performance in answering nearest neighbor queries [31]. Hjaltason et al. proposed a best-first search strategy by maintaining a priority queue in which the nodes or objects are sorted based on their distance to the query object [20]. However, answering the query is challenging in high dimensional space and various indices have been proposed based on the ideas of space partitioning [32, 27, 3, 8], data approximation [35, 2] or distance based indexing [23]. Another branch of works [17, 34] find approximate nearest neighbors via hashing for similarity query.

*k*NG query is a generalization of *k*NN query for *m* data sources, $m \geq 1$, such that each candidate answer consists of one object from each data source. Thus, when $m = 1$, *k*NG query reduces to *k*NN query.

2.2 kNN Join

*k*NN Join [4, 36, 37, 38, 28] or all-nearest-neighbors queries [41, 7] find nearest neighbors for all the points in a query set. It is an extension of *k*NN query in terms of the number of query points and leads to much more CPU overhead in query processing. In [4], the multipage index (MuX) together with an optimal page loading strategy were proposed to reduce both CPU and I/O cost. Xia et al. [36] optimized join scheduling by hashing the high dimensional points into blocks and sorting the blocks for a nested loop join. Tao et al studied all-nearest-neighbors queries in [41] which is a special case of *k*NN Join by setting $k = 1$. They used a depth-first traversal algorithm for datasets that are indexed by R-tree; otherwise, a hash-based algorithm is used. In [7], Chen et al. proposed the minimum bounding rectangle enhanced Quadtree as well as a new distance measure named NXNDIST for pruning. Recently, solutions utilizing MapReduce [10], a well-designed programming platform for large-scale parallel data processing, have been proposed in [38, 28].

*k*NG Join is similar to *k*NN Join in that they both need to retrieve *k* nearest results for all the points in a query set. Some of the techniques applied in *k*NN Join can be adopted for the processing of *k*NG Join. If we set $m = 1$, *k*NG Join becomes *k*NN Join.

2.3 mCK Query

*m*CK query was proposed by Zhang et al. in [39, 40]. Given *m* data sources, the result contains a group of *m* tuples from different sources with minimum inner-group distance. The query was initially proposed for location based keyword search and mapped resource locating. It also attracted some interests from the theoretical area [14, 12, 15]. They modelled the problem as computing minimum diameter color-spanning sets and proved that problem of processing *m*CK queries is NP-hard [14] for L_p metric ($1 < p < \infty$), even in two dimensional space.

Compared to *k*NG query, *m*CK query does not have a specified query object. It can be considered as a special case of *k*NG query. If we only consider inner group distance in *k*NG query, i.e., the distance from all the points to the query location is the same, then *k*NG query reduces to *m*CK query.

2.4 Team Formation

The team formation problem was proposed in [25]. Given a set of *N* individuals, each has a set of skills. The query is a task with a set of required skills. A team formation problem finds a group of people covering all the required skills and their group distance is minimized. This problem is similar to our *k*NG query except that their inner-group distance is measured by the social distance, which is more complex to handle because it is difficult to map the objects from social network to low-dimensional space and still keep the distance between pair objects.

2.5 Spatial Join

Given two datasets, spatial join [5, 21, 22] retrieves all pairs of objects satisfying a given join predicate. Common join predicates include overlap and containment. In [22], Jacox et al. provided a comprehensive study of various techniques proposed for spatial join, considering whether the datasets are indexed or not. Papadias et al. extended spatial join to multiway spatial join [30, 29] in which the spatial predicate is a function over multiple datasets. *k*NN Join and *k*NG Join can be considered as special types of multiway spatial join with complex join predicate.

3. COMPLEXITY RESULTS

In this section, we prove that both evaluating *k*NG query and *k*NG Join problems are NP-hard based on the distance definitions in Equations 4– 6.

LEMMA 3.1. *The problem of processing kNG query is NP-hard for L_p metric ($1 < p < \infty$), in two or higher dimensions.*

PROOF. In [14], Fleischer et al. proved that *m*CK query is NP-hard by reducing from 3SAT. The boolean variables in 3SAT clauses are mapped to points on a circle. We can adopt exactly the same reduction strategy by setting *q* as the center of the circle to prove our claim; details are given elsewhere [14]. □

LEMMA 3.2. *The problem of processing kNG Join is NP-hard for L_p metric ($1 < p < \infty$), in two or higher dimensions.*

PROOF. This can be easily established by contradiction. If the problem of processing kNG Join is not NP-hard, then it follows that processing kNG query is also not a NP-hard problem which contradicts Lemma 3.1. □

4. BASELINE ALGORITHM

In this paper, we study index-based query processing methods for kNG query and kNG Join because our proposed new queries are mainly applied in location based services and a spatial index can be built beforehand to improve efficiency. The methods designed for datasets without index are beyond the scope of the paper.

Recently in [6], Cao et al. proposed collected spatial keyword queries. Given a set of keywords and a query location q, it finds a group of spatial documents which together match all the query keywords and the distance is minimized. The problem can be modelled as a kNG query. We first split a spatial document into multiple tuples, each tuple with only one keyword. For each keyword w_i, we group all of its occurrence locations into a dataset D_{w_i}. The problem becomes finding k nearest groups among $D_{w_1}, D_{w_2}, \ldots, D_{w_m}$ where $\{w_1, w_2, \ldots, w_m\}$ represent m query keywords. The authors in [6] used IR-tree [9] to solve the collected spatial keyword queries. IR-tree is a single R-tree (compared to building one tree for one keyword) that stores all the spatial documents. The documents are inserted into the R-tree using the spatial attribute. Each node in the tree is augmented with an inverted file, which refers to a pseudo-document representing all the documents whose associated locations fall inside the node's MBR. IR-tree is widely used for top-k spatial keyword search [13, 9] as it can conduct spatial pruning and textual pruning at the same time.

Figure 3 illustrates an example spatial database as well as an IR-tree built on top of it. In this simple database, R_1 represents the entire spatial region which contains a total of 12 objects, each associated with one keyword. The keywords are represented by different shapes and each keyword happens to occur three times. Hence, for a keyword w_i, we can obtain a data source D_{w_i} containing three locations. As shown in Figure 3, the spatial objects are inserted into an IR-tree with node capacity set as 3. Each tree node is augmented with an inverted file indicating the existence of keywords in that node.

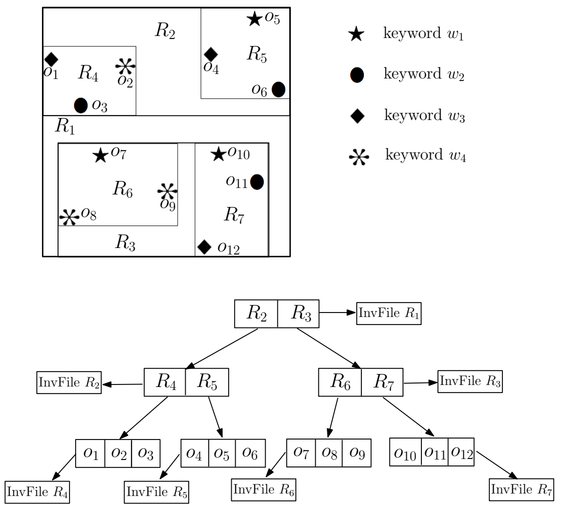


Figure 3: IR-tree example

Although the collective spatial keyword query can be modelled as a kNG query, we argue that IR-tree is not an appropriate index for kNG query for two main reasons:

1. IR-tree is useful when spatial pruning and textual pruning can be performed at the same time when answering top-k spatial keyword queries. However, in kNG query, the ranking function does not take into account textual relevance. Instead, it only considers spatial distance. Thus, it is not necessary to use such a hybrid index with high maintenance cost when the augmented inverted files do not play an important role in the pruning stage.

2. IR-tree is a single R-tree integrating all the data sources D_{w_i}. In Figure 3, all the objects are inserted into one R-tree. Compared to building one R-tree for one data source, such an index design incurs much more CPU and I/O cost. For instance, suppose we want to conduct a kNG query on two data sources D_{w_1} and D_{w_2}. Since these two keywords appear in all the leaf nodes, the whole tree will be accessed and a large number of tree node sets will be generated. An alternative solution is to build one R-tree for each data source D_{w_i}. If we use the same node capacity, which is 3 for this example, we only need to check two tree nodes.

In the following, we assume that a spatial index has been built for each data source. More specifically, we use R-tree as our spatial index because it is widely used for spatial queries like kNN queries [33, 20], kNN Join [41, 7] and spatial join [5, 21, 30, 29]. We propose a baseline algorithm for kNG query and kNG Join respectively.

4.1 Answering kNG Query Using R-tree

In the baseline algorithm to process a kNG query, we adopt the best-first strategy proposed in [20] which is used to answer nearest neighbor queries. The algorithm starts from the root node of the R-tree and maintains a priority queue in which the nodes or objects are sorted based on their minimum distance (*MINDIST*) to the query location in increasing order. The R-tree is traversed iteratively by removing one entry from the priority queue at a time. If the removed entry is an index node, its child nodes are inserted into the priority queue based on their distance to the query location. If the removed entry is an object, then the retrieved object is the i^{th} nearest neighbour if it is the i^{th} object being retrieved. The algorithm is I/O optimal because only the necessary index nodes are accessed.

Since we are handling multiple data sources in kNG query, we need to make some modifications to the algorithm. First, each entry in the priority queue is no longer an index node or a spatial object. Instead, it becomes a node set $\mathtt{N}$ which contains m nodes from different R-trees. The minimum distance from the query object q to the node set $\mathtt{N}$ is defined as follows:

$$MINDIST(q, \mathtt{N}) = \max_{n_i, n_j \in \mathtt{N}} MINDIST(n_i, n_j) + \min_{n_i \in \mathtt{N}} MINDIST(q, n_i) \tag{7}$$

where $MINDIST(n_i, n_j)$ and $MINDIST(q, n_i)$ are defined in line with previous works [33] to determine the minimum distance between two MBRs or between a point and an MBR. Figure 4 shows an example of calculating *MINDIST* between q and a node set $\mathtt{N} = \{r, s, t\}$. As shown in the figure, $MINDIST(q, \mathtt{N})$ is the sum of $MINDIST(q, t)$ and $MINDIST(r, t)$. We can use the distance measure as a lower bound distance for pruning as shown by the following result.

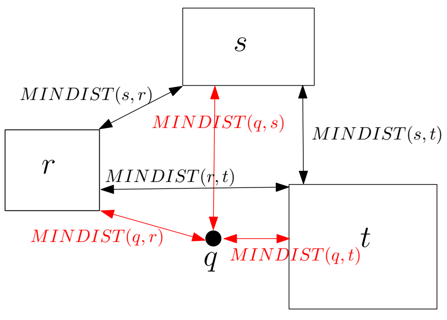
LEMMA 4.1. *Given a node set* $\mathtt{N} = \{n_1, n_2, \ldots, n_m\}$ *and a query point* q*, for any candidate* $g = \{o_1, o_2, \ldots, o_m\}$ *where* $o_i \in n_i$*, we have*

$$MINDIST(q,\mathtt{N}) \leq \| q,g \| \tag{8}$$

PROOF.

$$\begin{aligned} \| q,g \| &= \max_{o_i,o_j \in g} \| o_i,o_j \| + \min_{o_i \in g} \| q,o_i \| \\ &\geq \max_{n_i,n_j \in \mathtt{N}} MINDIST(n_i,n_j) + \min_{n_i \in \mathtt{N}} MINDIST(q,n_i) \\ &= MINDIST(q,\mathtt{N}) \end{aligned}$$

□



$$MINDIST(q,\{r,s,t\}) = MINDIST(q,t) + MINDIST(r,t)$$

Figure 4: ***MINDIST*** **between** q **and a node set** $\{r,s,t\}$

Second, we modify the way to break a tie when multiple node sets in the priority queue are associated with the same distance to q. If two node sets $\mathtt{N}_1$ and $\mathtt{N}_2$ have the same *MINDIST*, we prefer the one with smaller area value, which is calculated by summing all the areas of node MBRs in $\mathtt{N}$. A more compact node set provides better performance in the worst case. In other words, its upper bound distance is smaller.

The detailed baseline algorithm for *k*NG query is depicted in Algorithm 1. The input includes a query point q and a set of R-trees built for the m data sources. The algorithm returns top-k nearest groups $\mathbb{G}$ as well as the distance of k-th result, denoted by δ. Initially, in line 1, we create a priority queue *PQ* in which the node sets are sorted in increasing order of their *MINDIST* to q. A tie is resolved by comparing the total area of the node MBRs. Smaller nodes are preferred. We initialize a node set containing all the root nodes from the R-trees and enqueue it into *PQ* (lines 2-3). In addition, we initialize δ to a relatively small value by examining the combination of leaf nodes around q (lines 4). This step takes $m \cdot log(N)$ node accesses, where N is the tree size. Thereafter, we dequeue the best candidate $\mathtt{N}$ from the priority queue and check whether all the nodes in $\mathtt{N}$ are leaf nodes. If we have reached the bottom level in all R-trees, as shown in lines 10-12, we retrieve the objects stored in each leaf node and exhaustively calculate the distance of all the possible combinations of objects from different trees using the procedure `ExhaustiveSearch` (details are omitted). If a better result is found, $\mathbb{G}$ and δ are updated. If $\mathtt{N}$ contains internal nodes, we call function `EnumerateNodeSet` to expand $\mathtt{N}$ to a collection of new node sets composed of their child nodes. The new candidates are inserted into *PQ* for further checking (lines 14-17). Finally, the algorithm terminates when the priority queue becomes empty or the distance of the best candidate in *PQ* is larger than δ (lines 8-9). The remaining node sets do not need to be examined because they are their distance to q must be larger than δ and can be pruned by Lemma 4.1.

Algorithm 1 Baseline Algorithm for *k*NG Query

```
Input: A query point $q$ and $m$ R-trees
Output: $k$ nearest groups $\mathbb{G}$ and the $k$-th distance $\delta$
 1. create a new priority queue $PQ$
 2. initialize a node set $\mathtt{N}$ containing root nodes from $m$ R-trees
 3. initialize $\mathbb{G} \leftarrow \{\}$
 4. $\delta \leftarrow$ InitDist()
 5. $PQ$.Enqueue($\mathtt{N}$)
 6. while $PQ$ is not empty do
 7.   $\mathtt{N} \leftarrow PQ$.pop()
 8.   if $MINDIST(q,\mathtt{N}) \geq \delta$ then
 9.     break
10.   if all the nodes in $\mathtt{N}$ are leaf nodes then
11.     ExhaustiveSearch($\mathtt{N}$)
12.     update $\mathbb{G}$ and $\delta$
13.   else
14.     create an empty node set $\mathtt{N}'$
15.     $S \leftarrow$ EnumerateNodeSet($\delta$,$\mathtt{N}$, $m$, $\mathtt{N}'$)
16.     for $s \in S$ do
17.       $PQ$.Enqueue($s$)
18. return $\mathbb{G}$ and $\delta$
```

Algorithm 2 shows how to enumerate new node sets from the child nodes. The algorithm is implemented in a recursive manner. The *pos* starts from m and decreases by 1 in each recursion. If *pos* becomes 0, it means we have found a new candidate $\mathtt{N}'$. It contains m nodes and is inserted into the result set S (lines 1-2). Otherwise, we check whether the *pos*-th node in $\mathtt{N}$ is a leaf node. If it is a leaf node, we insert it into $\mathtt{N}'$ and calculate $MINDIST(q,\mathtt{N}')$ to see if $\mathtt{N}'$ is a valid node set. Note that although $\mathtt{N}'$ contains fewer than m nodes, we can still apply Equation 7 to calculate the distance. If the result is smaller than δ, we consider it valid and continue to expand the remaining nodes in $\mathtt{N}$ (lines 8-10). Otherwise, $\mathtt{N}'$ can be pruned because for any candidate containing nodes in $\mathtt{N}'$, its distance must be larger than δ. If the *pos*-th node in $\mathtt{N}$ is not a leaf node, we iterate its child nodes in the corresponding tree and examine the validity in the same way (lines 9-12). Finally, when all the child nodes in all the trees have been checked, the algorithm terminates and S is returned.

4.2 Answering kNG Join Using R-tree

To process *k*NG Join for a set of query points Q, a naive solution is to apply *k*NG query for each point in Q resulting in $|Q|$ independent *k*NG query evaluations. Although the method is simple, it incurs considerably high computational cost when $|Q|$ is large. A better solution is to take advantage of the spatial proximity of query points so that points that are close to one another are grouped together to reduce the distance calculation cost. Hence, we partition Q into a set of disjoint groups such that $Q = \{G_1, G_2, \ldots, G_g\}$ where $G_i \cap G_j = \emptyset$ and $\cup_{G_i} = Q$. We also define the minimum distance between G_i and a node set $\mathtt{N}$ as follows:

$$MINDIST(G_i,\mathtt{N}) = \max_{n_i,n_j \in \mathtt{N}} MINDIST(n_i,n_j) + \min_{n_i \in \mathtt{N}} MINDIST(G_i,n_i) \tag{9}$$

Algorithm 2 **EnumerateNodeSet**(δ, N, *pos*, N′)

Input: The k-th distance δ, a node set N to expand, a position variable *pos* and an empty node set N′
Output: A collection of new node sets S

```
1.  if $pos = 0$ then
2.    $S$.insert(N′)
3.  else
4.    if N[$pos$] is a leaf node then
5.      N′[$pos$] $\leftarrow$ N[$pos$]
6.      if $MINDIST(q, N') < \delta$ then
7.        EnumerateNodeSet($\delta$, N, $pos - 1$, N′)
8.    else
9.      for child nodes $c_i \in$ N[$pos$] do
10.       N′[$pos$] $\leftarrow c_i$
11.       if $MINDIST(q, N') < \delta$ then
12.         EnumerateNodeSet($\delta$, N, $pos - 1$, N′)
13. return $S$
```

where $MINDIST(G_i, n)$ is calculated by approximating G_i as an MBR. Similar to Lemma 4.1, we can use the following result for pruning.

LEMMA 4.2. *Given a node set* $\mathtt{N} = \{n_1, n_2, \ldots, n_m\}$ *and a group of query points* G_i*, for any point* $q \in G_i$ *and any candidate* $g = \{o_1, o_2, \ldots, o_m\}$ *where* $o_i \in n_i$*, we have*

$$MINDIST(G, \mathtt{N}) \leq \| q, g \| \tag{10}$$

PROOF.

$$\begin{aligned} MINDIST(G_i, \mathtt{N}) &= \max_{n_i, n_j \in \mathtt{N}} MINDIST(n_i, n_j) + \min_{n_i \in \mathtt{N}} MINDIST(G_i, n_i) \\ &\leq \max_{o_i, o_j \in \mathtt{g}} \| o_i, o_j \| + \min_{o_i \in \mathtt{g}} MINDIST(G_i, o_i) \\ &\leq \max_{o_i, o_j \in \mathtt{g}} \| o_i, o_j \| + \min_{o_i \in \mathtt{g}} \| q, o_i \| \\ &= \| q, g \| \end{aligned}$$

□

In the query processing stage, we traverse the groups in Q and each group is visited once. Suppose G_i is the active group we are examining, we trigger a kNG query to find nearest groups for all the points in G_i at one time. The search scheme is similar to Algorithm 1 except that q is replaced by a group of query points. Thus, we need to maintain a separate buffer for each query point to store its top-k results and k-th distance. The `ExhaustiveSearch` function checks all the possible combinations for all the points in G_i. For the pruning part in the algorithm, the *MINDIST* calculation is replaced by Equation 9. In addition, we need to define a different upper bound distance δ_{G_i} to replace δ in Algorithm 1 as follows:

$$\delta_{G_i} = \max_{q \in G_i} \delta_q \tag{11}$$

Any node set whose distance to G_i is larger than δ_{G_i} can be pruned. Figure 5 illustrates an example to show why the number of distance calculations can be reduced when query points that are close together are considered as a group. In this example, q_1, q_2 and q_3 form a group G with its δ_G value also depicted in the figure; S_1, S_2, R_1, and R_2 represent the MBRs of four index nodes. Here, R_2 and S_2 can be pruned because their *MINDIST* to G is larger than δ_G. In this way, there is no need to check the distance between $\{q_1, q_2, q_3\}$ and $\{R_2, S_2\}$ individually. More generally, for a group $G = \{q_1, q_2, \ldots, q_g\}$, the distance from q_i to the nodes far away from G is calculated only once, compared to g times in the naive solution. Thus, a considerable amount of computational cost can be saved.

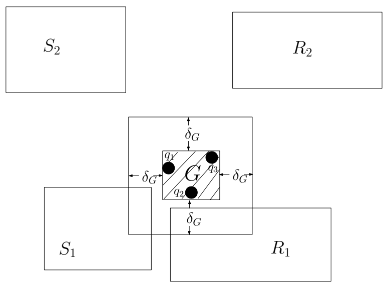


Figure 5: An example of group pruning

The remaining issue is how to partition Q into groups. In [41], Zhang et al. proposed a selection criteria with two threshold values for a group G. One is *max_area* to limit the MBR area of G and the other is *max_num* to limit the number of points in G. In this paper, we propose a simpler version with only one parameter. We build an R-tree on Q and treat each leaf node as a group. In this way, we can use the node capacity as a parameter to control both the group size and area because the points are usually well clustered in the leaf nodes. In our experiments, we will study the effect of this parameter on kNG Join processing.

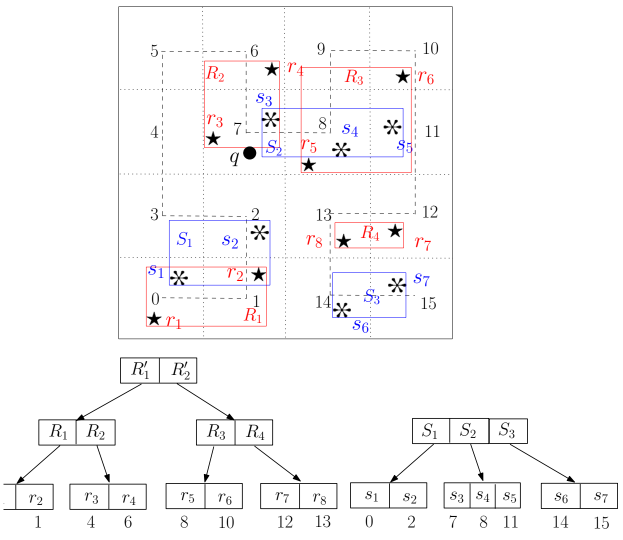
5. A HILBERT R-TREE-BASED SCHEME

The baseline algorithm adopts best-first search paradigm which is shown to be I/O optimal by accessing the minimum number of nodes when answering kNN queries [20]. However, kNG query is much more complex than kNN query as it needs to examine all the possible combinations of points in different data sources. As the number of data sources increases, the I/O cost increases linearly but the number of distance calculations grows exponentially, which causes CPU cost to become the bottleneck. The baseline algorithm starts from the root and traverses the trees in a top-down manner. Suppose δ is the k-th distance of the final top-k results. All the node sets whose minimum distance to the query point is smaller than δ have to be checked. For R-trees, since the MBRs of internal nodes at higher levels are larger and closer to each other, this results in an incredibly large number of node sets containing internal nodes.

To improve the query processing efficiency, we propose a new search algorithm using Hilbert R-tree [24]. Hilbert R-tree is a variant of R-tree that utilizes Hilbert curve, a space curving shown to preserve spatial locality most effectively, to guide data insertion. The tree nodes are sorted by the Hilbert value of the center point. In this way, each node has a well-defined set of sibling nodes so that we can use deferred splitting to improve node utilization. Figure 6 shows two Hilbert R-trees built on data sets R and S. The space is partitioned into 16 cells by a Hilbert curve with second order.

The structure is the same as R-tree and all the points are ordered based on the Hilbert value. In the following, we will present how to utilize the Hilbert R-tree to answer kNG query and kNG Join.

5.1 Answering kNG Query Using Hilbert R-tree



R_1' R_2'
R_1 R_2
R_3 R_4
S_1 S_2 S_3
r_2 r_3 r_4 r_5 r_6 r_7 r_8
s_1 s_2 s_3 s_4 s_5 s_6 s_7
q

Figure 6: Hilbert R-tree example

To overcome the drawback of generating too many node sets in the baseline algorithm, we propose to examine only the combinations of leaf nodes. The node set containing internal nodes will not be enumerated to save computational cost. In the optimal case, the leaf node set should be visited in increasing order of their distance to the query point. However, without global knowledge, it is unrealistic to enumerate all the combinations of leaf nodes and sort them by the distance. Our new search algorithm uses Hilbert value to guide the order of candidate enumeration so that promising node sets can be examined as early as possible. The algorithm is based on expand-and-prune. First, we obtain the Hilbert value of query point q, denoted by $H(q)$. We assume that the orders of our Hilbert curve for space partitioning are high enough so that each point is associated with a unique Hilbert value. For each Hilbert R-tree, we locate the leaf node whose Hilbert value is closest to $H(q)$. These leaf nodes are called pivot nodes. Given m trees, we can get m pivot nodes. Then, the search algorithm starts from these pivots and gradually expands to check the nodes in both the left and right directions so that the node sets close to q can be visited earlier. The algorithm terminates when it has expanded to the end on both directions in all the R-trees.

The details of the query processing are shown in Algorithm 3. Lines 1-7 depict the initialization work. In the first step, we locate the pivot leaf node for each Hilbert R-tree. These pivot nodes form a node set $\mathtt{N}$ serving as our initial search space. $\mathtt{N}$ is expected to be the most promising candidate because the nodes in $\mathtt{N}$ have Hilbert value closest to $H(q)$. This means that the pivot nodes are close to both q as well as one another. Hence, `ExhaustiveSearch` is called to get a reasonably good top-k results used for pruning. Given such an initial δ, we conduct a range query for each Hilbert R-tree to find the leaf nodes whose distance to q is smaller than δ. These leaf nodes are considered as active. We create two data structures, $\mathscr{L}_i$ and $\mathscr{R}_i$, to store active leaf nodes in tree T_i from the left side and right side respectively. The nodes in $\mathscr{L}_i$ and $\mathscr{R}_i$ are sorted in non-descending order of their Hilbert value distance to $H(q)$.

To effectively enumerate the node sets containing leaf nodes in $\mathscr{L}_i$ and $\mathscr{R}_i$, we associate with each leaf node n a variable $I(n)$, indicating its **interval distance** to the pivot node. In other words, $I(n)$ corresponds to the number of nodes between n and its pivot node. For instance, $I(n)$ for a pivot node is 0. For an immediate neighboring node n of a pivot node, $I(n)$ is set 1, although its Hilbert value distance to $H(q)$ could be much larger. We initialize a global variable $I = 1$ in line 8 as the expansion interval and gradually increase I to check nodes far from the pivots. In each expansion iteration, the node sets enumerated should satisfy the following condition to avoid duplicate enumeration:

$$\sum_{n \in \mathtt{N}} I(n) = I \tag{12}$$

The candidates generated in each iteration are inserted into a priority queue as in the baseline algorithm. `ExhaustiveSearch` is called to calculate the real distance for these candidates. $\mathbb{G}$ and δ are updated whenever a better result is found (lines 11-19). After each expansion, a pruning method is invoked to reduce the number of active nodes. The leaf nodes that have not been visited and with a distance to q larger than the new δ will be discarded (lines 20-26). The algorithm terminates when all the possible node sets in $\mathscr{L}_i$ and $\mathscr{R}_i$ have been enumerated (lines 27-29).

Algorithm 3 *k*NG Search Using Hilbert R-tree

```
Input: A query point $q$ and $m$ Hilbert R-trees
Output: $k$ nearest groups $\mathbb{G}$ and the $k$-th distance $\delta$
1.  for each Hilbert R-tree $T_i$ do
2.      find the pivot node $P_i$ closest to $H(q)$ in $T_i$
3.  Initialize a node set N containing all the $m$ pivot nodes
4.  ExhaustiveSearch(N)
5.  update $\mathbb{G}$ and $\delta$
6.  for each Hilbert R-tree $T_i$ do
7.      initialize $\mathscr{L}_i$ and $\mathscr{R}_i$
8.  $I \leftarrow 1$
9.  while TRUE do
10.     $S \leftarrow$ EnumerateNodeSet( $\delta, I, m$, N$'$)
11.     create a new priority queue $PQ$
12.     for $s \in S$ do
13.         $PQ$.Enqueue($s$)
14.     while $PQ$ is not empty do
15.         N $\leftarrow PQ$.pop()
16.         if $MINDIST(q, \mathtt{N}) \geq \delta$ then
17.             break
18.         ExhaustiveSearch(N)
19.         update $\mathbb{G}$ and $\delta$
20.     for $(i \leftarrow 1; i \leq m; i \leftarrow i+1)$ do
21.         for $(j \leftarrow I; j \leq |\mathscr{L}_i|; j \leftarrow j+1)$ do
22.             if $MINDIST(q, \mathscr{L}_i[j]) \geq \delta$ then
23.                 remove $\mathscr{L}_i[j]$
24.         for $(j \leftarrow I; j \leq |\mathscr{R}_i|; j \leftarrow j+1)$ do
25.             if $MINDIST(q, \mathscr{R}_i[j]) \geq \delta$ then
26.                 remove $\mathscr{R}_i[j]$
27.     $maxI \leftarrow \sum_{1 \leq i \leq m} \max(|\mathscr{L}_i|, |\mathscr{R}_i|)$
28.     if $maxI < I$ then
29.         break
30.     $I \leftarrow I + 1$
31. return $\mathbb{G}$ and $\delta$
```

Given an expansion interval I, Algorithm 4 is a recursive procedure to enumerate all the candidate node sets satisfying the constraint in Equation 12. In Table 1, we present a running example of our search algorithm based on interval expansion. It finds the nearest groups from R and S for the query point in Figure 6. Since $H(q) = 7$, the pivot nodes in R and S are R_2 and S_2 respectively. After calling `ExhaustiveSearch` on $\{R_2, S_2\}$, we get a small δ and invoke a range query to initialize the active leaf nodes in R and S. $\mathscr{L}_R$ and $\mathscr{R}_S$ are empty because the leaf nodes on the side of the pivot are far away and get pruned. $\mathscr{R}_R$ and $\mathscr{L}_S$ both contain one element. Then, we expand the search space to enumerate new node sets for $I = 1$ and $I = 2$. The algorithm terminates when I increases to 3 as we cannot find a node set satisfying Equation 12. The algorithm only checks four node sets, which is more efficient than the baseline method.

Algorithm 4 `EnumerateNodeSet`$(\delta, interval, pos, \mathtt{N}')$

```
Input: the k-th distance $\delta$, current expand interval $interval$, a position variable $pos$ and an empty node set $\mathtt{N}'$
Output: A collection of new node sets $S$
1.  if $pos = 0$ then
2.      $S$.insert($\mathtt{N}'$)
3.  else
4.      if $\sum_{1 \le i < pos} \max(|\mathscr{L}_i|, |\mathscr{R}_i|) \ge interval$ then
5.          $\mathtt{N}'[pos] \leftarrow P_i$
6.          EnumerateNodeSet($\delta$, $interval$, $pos - 1$, $\mathtt{N}'$)
7.      for $i \leftarrow 1; i \le interval; i \leftarrow i + 1$ do
8.          if $pos = 1$ && $i \neq interval$ then
9.              continue
10.         if $i \le |\mathscr{L}_i|$ then
11.             $\mathtt{N}'[pos] \leftarrow \mathscr{L}_i[i]$
12.             EnumerateNodeSet($\delta$, $interval - i$, $pos - 1$, $\mathtt{N}'$)
13.         if $i \le |\mathscr{R}_i|$ then
14.             $\mathtt{N}'[pos] \leftarrow \mathscr{R}_i[i]$
15.             EnumerateNodeSet($\delta$, $interval - i$, $pos - 1$, $\mathtt{N}'$)
16. return $S$
```

Table 1: Running example of interval expansion

I	Priority Queue	$\mathscr{L}_R$	$\mathscr{R}_R$	$\mathscr{L}_S$	$\mathscr{R}_S$
$I = 0$	$\{R_2, S_2\}$	$\{\emptyset\}$	$\{R_3\}$	$\{S_1\}$	$\{\emptyset\}$
$I = 1$	$\{R_3, S_2\}, \{R_2, S_1\}$	$\{\emptyset\}$	$\{R_3\}$	$\{S_1\}$	$\{\emptyset\}$
$I = 2$	$\{R_3, S_1\}$	$\{\emptyset\}$	$\{R_3\}$	$\{S_1\}$	$\{\emptyset\}$

5.2 Answering kNG Join Using Hilbert R-tree

For kNG Join, we adopt the same grouping strategy as in the baseline algorithm. We build an R-tree for the query point set Q and traverse its leaf nodes to search the nearest groups. Since a query leaf node contains more than one point, there exists multiple pivot nodes in each Hilbert R-tree. Algorithm 3 is extended to provide a new node set enumeration order to save computational cost. Figure 7 shows an example with two query points $\{q_1, q_2\}$ in a group, leading to two pivot node sets $\{R_2, S_2\}$ and $\{R_4, S_3\}$. It is obvious that these two pivot node sets should be examined first so that we can get a small δ_G for pruning. In the interval expansion stage, we iteratively enumerate the node sets near each pivot set. In this way, δ_G can be reduced as much as possible. To avoid duplicate invocation of `ExhaustiveSearch`, we maintain an additional hash map to check whether a node set has been previously examined. Since there are only a relatively small number of active leaf nodes in each R-tree, the hash map will not cause a burden to the memory resource. The remaining part of the algorithm is similar to Algorithm 3 and its details are omitted due to space constraint.

6. QUERY COST MODEL

In this section, we analyze the cost model of an optimal approach, denoted by kNG-opt, to evaluate kNG query based on the distance definitions given by Equations 4– 6. To simplify the model, we make the following assumptions:

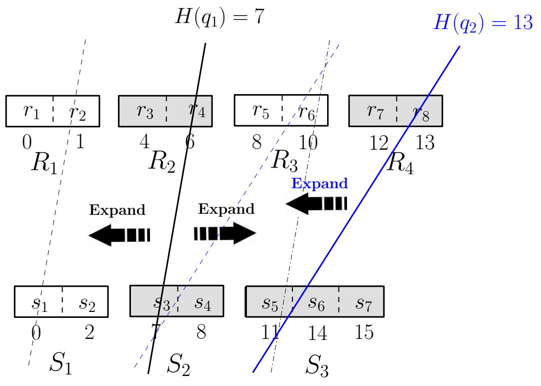


Figure 7: Incremental expansion for kNG Join

1. There are m data sources $D_1, D_2, \ldots, D_m$. Each D_i has exactly N spatial objects.
2. The spatial objects are two-dimensional points.
3. The spatial points in each D_i are uniformly distributed in space $[0,1)^2$.
4. The query points are distributed uniformly as well.
5. We set $k = 1$ to find the closest group with respect to q.
6. R-tree index is used and the MBRs of the data pages are squares with length σ. Each node contains an average number of C_{avg} objects.

Since kNG query is expensive in terms of both CPU and I/O cost, we provide estimation for both aspects in our model. Similar to the cost model for NN query in [31], we estimate the I/O cost by the number of leaf nodes accessed. For the CPU cost, we estimate it by the number of candidates generated in the optimal solution because the distance calculation between q and these candidates is the main CPU bottleneck. Suppose δ is the best distance, we first prove that only the data points with distance no greater than δ will be accessed by the optimal solution.

LEMMA 6.1. *If δ is the distance of the best result, the candidate search space includes the points p satisfying $\| q, p \| \le \delta$*

PROOF. If p is a point and $\| q, p \| > \delta$, we can prove that for any candidate g containing p,

$$\| q, g \| \ge \| q, p \| > \delta$$

, based on our distance constraints given by Equations 4, 5 and 6. Then, the group can be pruned. On the other hand, for a point p with $\| q, p \| \le \delta$, it can not be pruned. The point will be accessed in the search stage because it is possible that the inner-group distance is zero, resulting in a better result. □

Based on the work in [11], given a query window with size $q' \times q'$ and center at q , the number of leaf nodes accessed, denoted by $L(q)$, can be estimated as follows:

$$L(q) = \frac{N}{C_{avg}} \cdot (\sigma + q')^2 \qquad (13)$$

Similarly, since our data model contains m R-trees, we can estimate the I/O cost as the total number of leaf nodes accessed by the optimal solution:

$$I/O_{cost} \approx \frac{m \cdot N}{C_{avg}} \cdot (\sigma + \delta)^2 \tag{14}$$

To estimate the CPU cost, since all the points in the circle cannot be pruned, all of their combinations have to be enumerated in the optimal solution. By applying the formula in [1], the CPU cost can be estimated by

$$\begin{aligned} CPU &\approx (nb(\delta, circle))^m & (15)\\ &= \pi^m \cdot (N-1)^m \cdot \delta^{2m} & (16) \end{aligned}$$

Finally, we need to estimate the value of δ, which is an unknown variable in both CPU and IO cost model. We know that the distance of the nearest neighbor in a data source D_i can be calculated with the following formula [31]:

$$nn(q, D_i) = \frac{1}{\sqrt{\pi} \cdot \sqrt{N-1}} \tag{17}$$

In this case, if we draw a circle with radius $nn(q, D_i)$ and center at q, there exists m nearest points from different sources on the circle. Based on our distance definition, the inner-group distance falls within $[\,0,\ 2 \cdot nn(q, D_i)]$. Hence, we have

$$\frac{1}{\sqrt{\pi} \cdot \sqrt{N-1}} \leq \delta \leq \frac{2}{\sqrt{\pi} \cdot \sqrt{N-1}} \tag{18}$$

From the above cost functions, we can observe that the I/O cost grows linearly with m while the CPU cost grows exponentially with m. Thus, when the number of data sources is large, kNG query and kNG Join are CPU bound.

7. EXPERIMENT STUDY

In this section, we study the performance of the baseline algorithm and the Hilbert R-tree solution in answering kNG query and kNG Join. We also compare our methods with the IR-tree approach proposed for the collective spatial keyword search scenario in [6]. All the experiments were conducted on a server with Quad-Core AMD Opteron(tm) Processor 8356, 64GB memory, running Centos 5.6.

7.1 Experiment Setup

We implemented three types of disk-based indexes in C++, including R-tree, Hilbert R-tree and IR-tree. The page size was set to 4KB. In the construction of the R-tree and IR-tree, we chose relatively small node size based on findings from the parameter tuning experiments. To build the Hilbert curve, we split the cell small enough such that each point has a unique cell id. It incurs higher index construction cost but facilitates the query processing. In the implementation of IR-tree, we set the node capacity to 100 as reported in [6]. In addition, we used a more efficient way to improve the performance of IR-tree. We created a fixed-size bitmap, with each bit representing one keyword to indicate whether a keyword appears in a node. If a query keyword occurs in that node, its corresponding bit is set to 1. Since the total number of unique keywords in our datasets is not large, the bitmap checking is more efficient than using an inverted index.

In the query processing stage, we report both running time and I/O cost for kNG query to show the kNG query processing is CPU-bound. The I/O cost grows linearly with the number of query keywords m while the running time increases exponentially. Hence, for kNG Join, we only report the running time as the I/O cost is minor and can be ignored.

7.2 Datasets

To ensure a comprehensive performance study, we conducted our experiments on both synthetic and real databases:

1. **Uniform**. The data sets are two dimensional points and generated in uniform distribution in the space $[0,1)^2$.

2. **Twitter** For the Twitter dataset, we extracted 1 million tweets with geographical information represented in the form of latitude and longitude. The infrequent keywords were removed from the sample database. Finally, the dataset contains 959 unique and frequent keywords and each tweet has an average of 4 keywords.

3. **USGS** [26]. The dataset contains 105,725 points of interest under 63 categories in California. Each category represents one data source. The dataset is highly skew in terms of data size and spatial distribution for each category. For instance, "school" has 11,186 occurrences while there are only 40 points associated with "forest".

4. **FourSquare** [1]. This dataset contains 206,416 venues under 355 categories in New York City.

7.3 Parameter Tuning

The node capacity is an important parameter of R-tree in answering kNG queries. When this parameter gets smaller, the tree height increases and more nodes will be accessed, incurring high I/O cost. On the other hand, as the node capacity increases, the nodes become larger and closer to each other. The node sets at higher levels of the tree are not easy to be pruned, leading to more node set generation. Figure 8 shows the running time of the baseline algorithm in answering kNG query with respect to different internal node capacity and leaf node capacity. We increase the node capacity from 5 to 65 and report the performance for each pair choice. We can see that when the node capacity is set to 15, the index achieves the best performance. Thus, in the following experiments, we use 15 as the default node capacity for the baseline algorithm. The Hilbert R-tree also exhibits a similar performance pattern and it performs best when the node capacity is set to 10; the detailed results are omitted due to space constraint.

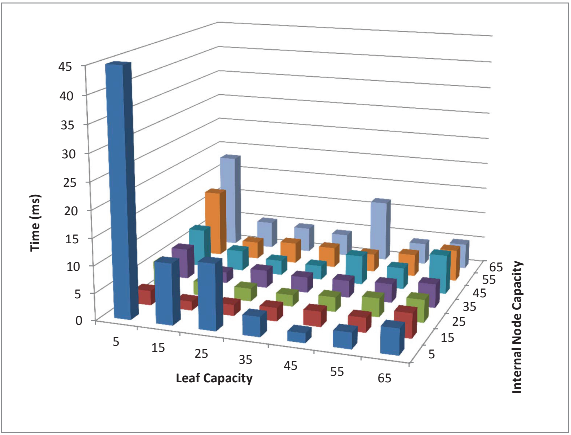


Figure 8: Parameter tuning for tree node capacity

[1]http://www-users.cs.umn.edu/~baojie/Research.htm

7.4 Experiments on kNG Query

In this subsection, we compare the performance of *k*NG query processing with respect to different parameter settings. More specifically, we study the effect of the number of data sources *m*, the dataset cardinality $|D_i|$ and the number of results *k*.

7.4.1 Effect of m on kNG Query

In this experiment, we run *k*NG queries on an increasing number of data sources in both synthetic and real datasets. In the synthetic dataset, we generated data sources with the same cardinality, i.e., $|D_i|$ is set to 100K for each data source. We compared the Hilbert R-tree solution with the baseline algorithm. In each experiment, 100 randomly generated queries were executed and the average running time and I/O accesses to retrieve top-10 results are reported in Figures 9(a) and 9(b). We can see that as *m* increases, the running time of both solutions grows exponentially. Both algorithms enumerate exponential number of node sets, incurring very high CPU cost when *m* is large. However, the I/O cost grows linearly with *m*. It shows that the *k*NG query is CPU-bound for large *m*. The results are consistent with our cost model in Section 6.

For the three real datasets, we compared our methods with IR-tree. The CPU and I/O performance comparison for the Twitter dataset are shown in Figures 9(c) and 9(d), respectively. Note that we have removed the infrequent keywords and this dataset can be used to test the performance in the worst case where the search space is extremely large. The results show that the performance of IR-tree is orders of magnitude worse than our solutions in terms of both CPU and I/O cost. Even for a *k*NG query on two data sources, its running time is 1 second while our methods take only 1 millisecond. Hence, a single-index-based solution is not efficient for *k*NG query. If a keyword is frequent, it appears in a large number of tree nodes, leading to expensive access cost. On the other hand, the effective area of a keyword in a tree node could be much smaller than the node MBR. This makes the query keywords closer to each other and it becomes more difficult for the node set pruning.

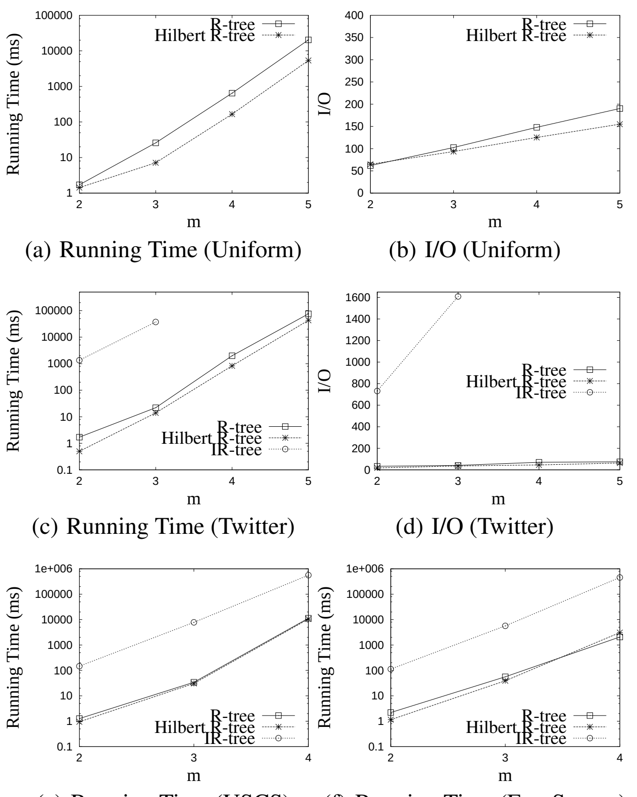
The running time performance for the UCSC and FourSquare datasets are shown in Figures 9(e) and 9(f), respectively. ..... Observe that the baseline algorithm performs slightly better than the Hilbert R-tree solution in the FourSquare dataset. The reason is this dataset is not as skew as the USGS in terms of the distribution of data source cardinality. Most of the data sources contain hundreds of points. Hence, it incurs a small cost for the baseline algorithm to enumerate the candidate node sets in a top-down and best-first manner. In the following experiments, we will use $m = 2$ as a default setting for the number of data sources to query.

7.4.2 Effect of $|D_i|$ on kNG Query

In this experiment, we fix $m = 2$ and increase the dataset size $|D_i|$ from 200K to 1M to examine the performance of Hilbert R-tree solution and the baseline algorithm. Again, 100 sample queries are executed and Figure 10 shows how the average running time and I/O access increase with $|D_i|$. As the dataset size grows, the height of the R-tree in the baseline algorithm becomes larger, leading to the enumeration of more node sets containing internal nodes. The Hilbert R-tree only checks the combination of leaf nodes. Hence, it demonstrates better performance than the baseline algorithm with an increasing dataset cardinality.

7.4.3 Effect of k on kNG Query

To compare the effect of *k* on the query processing performance, we increase it from 1 to 100. Figure 11 shows the performance results for two data sources with $|D_i| = 50$K. As *k* increases, it takes longer to answer a query as the distance used for pruning becomes larger. Thus, more nodes will become candidates, thereby incurring higher CPU and I/O cost. We can see that Hilbert R-tree scales better than the baseline algorithm. When *k* increases to 50, the performance of the baseline algorithm declines rapidly.

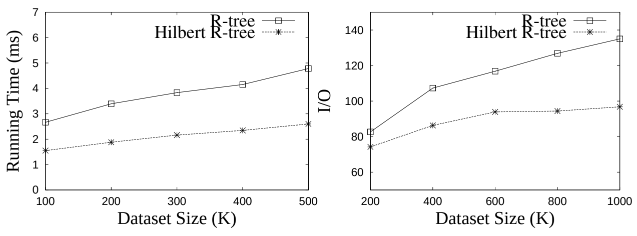


(a) Running Time (Uniform) (b) I/O (Uniform)

(c) Running Time (Twitter) (d) I/O (Twitter)

(e) Running Time (USCS) (f) Running Time (FourSquare)

Figure 9: Effect of *m* for *k*NG query



(a) Running Time (b) I/O

Figure 10: Effect of $|D_i|$ for *k*NG Query

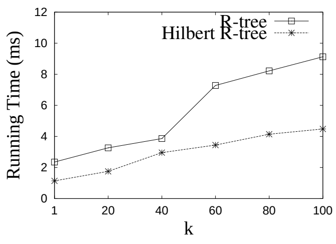


Figure 11: Effect of query set size

7.5 Experiments on kNQ Join

In this section, we compare the performance of the various methods for *k*NG Join. Note that all the indices are memory-based for these experiments and only the running time is reported. We report results for synthetic datasets in Sections 7.5.1 to 7.5.3, and results for real datasets in Section 7.5.4.

7.5.1 Effect of query grouping on kNG Join

In this experiment, we evaluate the effectivness of query grouping on R-tree index for *k*NG Join. Given a query set Q, we build an R-tree and treat each leaf node as a group. The points in the leaf node are considered to be well clustered. We can adjust the node capacity for appropriate group size selection. In this experiment, the query set size is set to 50K and we vary the dataset cardinality $|D_i|$ to evaluate the effect of grouping. We run experiments with $|D_i| = 10$K and $|D_i| = 50$K and the results are shown in Figure 12.

For the case in Figure 12(a), the query points are more dense than the data sources D_i. For example, this happens when there are a larger number of customers subscribing for "iPhone 5" and "accessory". We can see that the grouping does not take much effect for the baseline algorithm in Figure 12(a). Its performance slightly improves when the node capacity increases from 4 to 12. Then, it starts to decline when the node capacity continues to increase. However, in Figure 12(b), the grouping takes effect on the baseline solution. The performance improves dramatically when the node capacity increases to 8. We found that this was mainly caused by a reduction of node sets enumerated in the search stage. Compared to D_i with $10K$ points, the distance result in $50K$ is smaller and good for pruning. The Hilbert solution gains advantage from the grouping in both figures as it only explores the combination of leaf nodes. As long as query points close to each other are grouped, the performance can be improved.

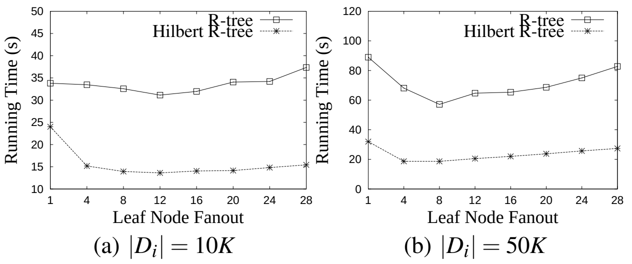


Figure 12: Effect of grouping in answering *k*NG Join

7.5.2 Effect of |Q| for kNG Join

In this experiment, we fix $|D_1|$ and $|D_2|$ to be $50K$ and vary $|Q|$ from 50K to 250K to examine the effect of query set size for processing *k*NG Join. We can see from Figure 13 that the running time of both methods grows linearly with $|Q|$. In addition, the Hilbert R-tree solution scales much better than the baseline algorithm. Since the query points and the data points are ordered by the Hilbert value, the method takes good advantage of L2 cache in accessing the data and demonstrates better pruning effect [24].

7.5.3 Effect of m and k on kNG Join

Similar to *k*NG query, we also report the effect of m and k on *k*NG Join. The experiments were conducted with $|Q| = 50$K and $|D_i| = 10$K. The results shown in Figure 14 exhibit similar performance trends as those for *k*NG query.

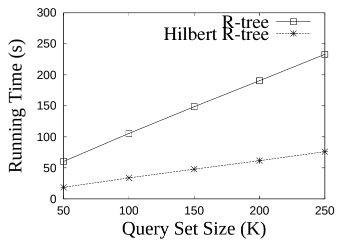


Figure 13: Effect of query set size

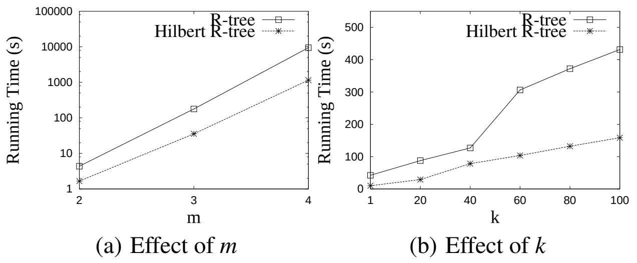


Figure 14: Effect of k and m in *k*NG Join

Table 2: *k*NG Join Test Queries on Twitter Dataset

sn	Query	$\|Q\|$	Group Keywords	$\|D_i\|$
Q_1	lake	37419	hotel,restaurant	77406, 54360
Q_2	hotel	77406	beer,rock	64813, 60003
Q_3	highschool	4595	tutor,swimming	17999, 2554
Q_4	hostel	3468	club,bar	89325, 89768

7.5.4 kNG Join on real datasets

In this section, we evaluate the performance for *k*NG Join on the Twitter dataset using four test queries shown in Table 2. For example, Q_1 finds the k nearest pairs of hotel and restaurant for all the lakes to benefit the visitors with a travel plan to that location. The running time of *k*NG Join for each query is shown in Figure 15. We can see that Hilbert R-tree solution outperforms the baseline algorithm by orders of magnitude. It validates the effectiveness of utilizing the Hilbert value to guide the order of node set enumeration.

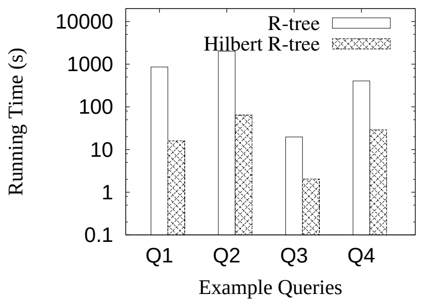


Figure 15: kNG Join performance on Twitter dataset

8. CONCLUSION

In this paper, we proposed two types of new queries named *k*NG query and *k*NG Join to find nearest groups. Such queries have wide applications in local service recommendation, spatial social network and publish/subscribe systems. We proved that the problem of processing each of these two query types is NP-hard and proposed two index-based solutions using R-tree and Hilbert R-tree respectively. We also presented a cost model to estimate the CPU and IO cost for for *k*NG query. Finally, our extensive experimental study on both synthetic and real datasets showed that the scheme based on Hilbert R-tree can process these new queries efficiently.

9. ACKNOWLEDGEMENT

This work is funded by the NExT Search Centre (grant R-252-300-001-490), which is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

10. REFERENCES

[1] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 299–310. Morgan Kaufmann, 1995.

[2] S. Berchtold, C. Böhm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, pages 577–588, 2000.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree : An index structure for high-dimensional data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.

[4] C. Böhm and F. Krebs. The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD Conference*, pages 237–246, 1993.

[6] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD Conference*, pages 373–384, 2011.

[7] Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *ICDE*, pages 1056–1065, 2007.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[9] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[11] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *PODS*, pages 4–13, 1994.

[12] C. Fan, W. Ju, J. Luo, and B. Zhu. On some geometric problems of color-spanning sets. In *FAW-AAIM*, pages 113–124, 2011.

[13] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[14] R. Fleischer and X. Xu. Computing minimum diameter color-spanning sets. In *FAW*, pages 285–292, 2010.

[15] R. Fleischer and X. Xu. Computing minimum diameter color-spanning sets is hard. *Inf. Process. Lett.*, 111(21-22):1054–1056, 2011.

[16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[17] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[19] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[20] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[21] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 396–405. Morgan Kaufmann, 1997.

[22] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.

[23] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[24] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, 1994.

[25] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.

[26] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[27] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.

[28] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.

[29] N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4):424–475, 2001.

[30] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *PODS*, pages 44–55, 1999.

[31] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in r-trees. In *ICDT*, pages 394–408, 1997.

[32] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD Conference*, pages 10–18, 1981.

[33] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 71–79. ACM Press, 1995.

[34] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD Conference*, pages 563–576, 2009.

[35] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

[36] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *VLDB*, pages 756–767, 2004.

[37] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE*, pages 4–15, 2010.

[38] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2012.

[39] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[40] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.

[41] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, pages 297–306, 2004.