Which is Better for kNN Query Processing in the Cloud: Sequential or Parallel[*]

Chong Zhang[1], Xiaoying Chen[2], Bin Ge[3], Weidong Xiao[4]

[*]Science and Technology on Information Systems Engineering Laboratory
National University of Defense Technology, Changsha 410073, China
[+]Collaborative Innovation Center of Geospatial Technology, China

{[1]leocheung8286, [2]chenxiaoying1991}@yahoo.com
[3]gebin1978@gmail.com, [4]wilsonshaw@vip.sina.com

ABSTRACT

With the development of various Cloud system, providing powerful kNN query capability to DaaS (Database as a Service) is an essential requirement for many applications. In this paper, we are interested in two opposite approaches for processing kNN query in Cloud system, parallel processing and sequential processing, and we want to explore the answer of which one performs better. For addressing such a question, we devise a new distributed indexing structure VIHCO, which is characterized by fast locating Cloud nodes capability. Then parallel and sequential processing methods are designed upon the structure. For parallel one, we take differential cells between two consecutive range queries into consideration, and for sequential one, we elaborately design an accurate message delivery algorithm. We verify our ideas through experiments, which is conducted on both synthetic and real dataset, and the results show that VIHCO outperforms a previous work RT-CAN, and the sequential method is more efficient under small k query condition and small system size, while parallel one suits for large k and large scale of computing nodes.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - *query processing*

General Terms

Algorithms, Measurement, Performance

Keywords

kNN query, Cloud, histogram, parallel, sequential

[*]This work is supported by NSF of China grant 61303062 and 71331008.

1. INTRODUCTION

With the development of Cloud computing, various layers of computing resources are used in terms of *pay-as-you-go*, such as IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service). Nowadays, Database as a Service (DaaS)[2][13] is a hot topic for database community in Cloud computing era. For DaaS users, it is not necessary to focus on the location of database instance, nor the physical storage mechanism of schema or tables, not even data partition fashion or query processing, in one word, the inner of database is transparent to the users. They just define the structure of table, and insertion, query or other operations seem similar to use a centralized local database. However, it is possible for one table, data are spread over many computing nodes, and querying processing needs the collaboration of these nodes. And as data volume increases, the database should be adaptive to the new scale and new query requirement, i.e., it should be elastic.

In this paper, we focus on kNN query in DaaS, which is an essential function for spatial database. Given a point in the space, kNN query aims to find k nearest objects to the query point. This topic is addressed well in some previous works[13][9][14], however, there are two opposite ideas to solve the problem in the state-of-the-art, namely, parallel processing and sequential processing. Parallel method exploits the parallelism of Cloud nodes, and make them work simultaneously, while sequential one uses the vicinity relationship between query point and Cloud nodes to accurately deliver query messages. Nevertheless, which is better for DaaS is not studied before. Hence, in this paper, we extend our work in [14] to acquire the answer.

For comparing the two approaches, we use a previous work RT-CAN as a baseline, and propose a new structure, called VIHCO (VIcinity-based Hilbert Cloud Overlay), to index spatial data in Cloud system and to process kNN query. The feature of VIHCO is not only leveraged on fast look up routing table (finger table), but also highlighted on vicinity neighbors to quickly locate the nearby Cloud nodes. Based on such structure, we present the designs of parallel and sequential processing algorithms. Experiments on both synthetic and real dataset show that VIHCO outperforms RT-CAN, in efficiency and scalability, and the sequential method is more proper under small k query condition and small system size, while parallel one suits for large k and large scale of computing nodes.

To summarize, we make the following contributions:

- We propose the problem of comparing performances between parallel and sequential processing for kNN query in Cloud computing architecture.
- We design an indexing structure VIHCO capable to fast locate Cloud nodes.
- We devise parallel and sequential processing algorithms based on VIHCO, and conduct experiments to compare their performances under different parameters.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 formally defines the problem and presents VIHCO structure. Algorithms for parallel and sequential processing are presented in section 4 and 5, respectively. And we experimentally evaluate VIHCO and acquire comparison answers in section 6. Finally, section 7 concludes the paper with directions for future works.

2. RELATED WORKS

A similar work RT-CAN solving multidimensional queries in Cloud system is proposed in [13]. Each peer firstly builds R-tree to index multidimensional data locally, and then selects the nodes in the level above the leaf level of R-tree to publish to CAN by interacting with overlay node. RT-CAN addresses point query, range query and kNN query for multidimensional data in Cloud system. Later, the authors proposed a framework for supporting DBMS-like indexes in the Cloud[2], enabling users to define their own indexes without knowing the complicated structure of the underlying network, and three conventional indexes, namely hash indexes, B^+-tree-like indexes and multi-dimensional indexes are implemented. The results of work [14] prove that our proposed structure DRISTIX is better than RT-CAN, and in this paper, we extend our last work to exclusively study kNN query processing.

Several works from moving objects query and sensor network are also related to our work. DTI[6] is a distributed index for trajectory queries on moving objects. For each moving object, it builds a SkipList overlay to index trajectory. This method suffers from high overhead, because it maintains complicated overlay connection when the number of moving objects is large. Hua Lu et al.[7] address continuous queries for monitoring moving objects constrained in road network in a distributed environment. However, they adopt a central server to control the whole distributed queries, which is a bottle neck in real application. DIST[8] addresses tracking moving object in sensor network, it uses quad-tree to divide space recursively and builds distributed index hierarchically among sensors, and adopts efficient update to reduce maintenance cost. However, due to lack of scalability of tree structure, DIST suffers bottle neck problem.

A related work addresses multi-dimensional queries for PaaS, MD-HBase[9] uses k-d tree and quad-tree to partition space and adopts Z-curve to convert multidimensional data points into one dimension, and supports multi-dimensional range and nearest neighbor queries, which leverages a multidimensional index structure layered over HBase. However, this work is limited in HBase, it could not be generalized to many Cloud system.

3. PROBLEM DEFINITION AND OVERLAY STRUCTURE

3.1 Problem Definition

In this paper, only two-dimensional space is considered. A Cloud platform contains a set $\mathcal{C}$ of Cloud nodes, $\mathcal{C}$=$\{cn_i \mid 1 \leq i \leq n\}$, where cn_i denotes the i-th Cloud node with stand alone storage and computational capability. A set $\mathcal{O}$ of spatial objects is spread over Cloud nodes $\mathcal{C}$ according to the underlying Cloud storing policy. And each object obj is represented as $(x,\, y,\, shp)$, where $(x,\, y)$ denotes the location of obj, and shp is a set of points denoting the shape of obj. If obj is a point object, shp is null. Otherwise, i.e., obj is an object with spatial extent, an MBR (Minimal Bounding Rectangle) is used as the approximated shape of the obj, and $(x,\, y)$ means the center of the MBR.

A kNN query could be formally defined as: given a set $\mathcal{O}$ of spatial objects, a kNN query $(q$=$(x_q,\, y_q),\, k)$, aims to find a set $\mathcal{O}_q \subseteq \mathcal{O}$, such that $|\mathcal{O}_q|$=k, and $d(o,q) \leq d(o',q)$, $\forall o \in \mathcal{O}_q$, $o' \in \mathcal{O} \setminus \mathcal{O}_q$, where $d()$ is the Euclidean distance function.

3.2 Overlay Structure

We propose an overlay structure called VIHCO (Vicinity-based Hilbert Cloud Overlay), which takes advantage of both linearization of Hilbert curve[5] and space vicinity. The overview of the structure is presented in Figure 1.

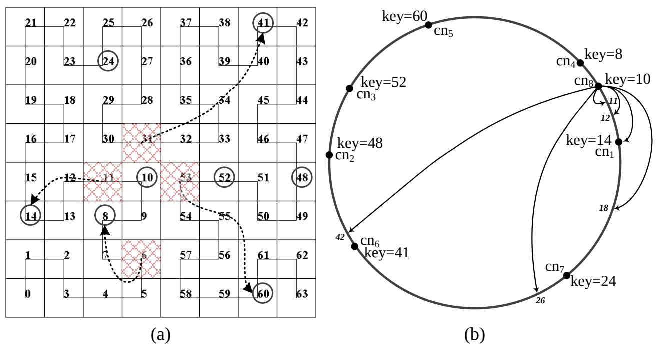


Figure 1: Overview of VIHCO

The whole space is partitioned by Hilbert curve, in particular, for a λ-order Hilbert, the space is divided into $2^{2\lambda}$ cells, and each cell is attached a Hilbert value varied from 0 to $2^{2\lambda} - 1$. Initially, Cloud nodes are formed into overlay network using Chord protocol[10], a distributed hash-based (key-value) P2P network structure. Formally, a set of 2^m consistent keys $[0,\, 2^m - 1]$ is maintained by a set of Cloud nodes, and each Cloud node is exclusively responsible for maintaining a segment $[key_s,\, key_e]$ $(0 \leq key_s \leq key_e \leq 2^m - 1)$, each key of which is associated with data items (value). According to the order of the segment maintained in the key domain, each Cloud node connects to its predecessor node and successor node, in addition, it maintains a routing table called finger table containing m entries where the i-th entry points the node maintaining the key key_e+2^{i-1}. For the Cloud nodes maintaining the first and the last segments of keys, they are also connected by each other as predecessor and successor, so the whole structure looks like a ring. Figure 1(b) illustrates an example, node

cn_8 maintains key segment [9, 10], with connecting predecessor cn_4 and successor cn_1, for maintaining figure table, cn_8 finds the nodes which maintain the key $10+2^0$, $10+2^1$, $10+2^2$, $10+2^3$, $10+2^4$, $10+2^5$, respectively, resulting cn_1, cn_7, cn_6, cn_2.

For simplicity, let $2\lambda = m$, which means that each cell in the space is mapped to a key on the ring, i.e., each Cloud node cn_i is responsible for maintaining a set of cells which are continuous in Hilbert value domain, and we call these cells as cn_i's *charge-cell*s and cn_i is these cells' *charge-node*. The term *charge* here means, if an object *obj* spatially intersects with cell c_j, *obj* is maintained (stored and indexed) by c_j's *charge-node*. Figure 1 illustrates an example of VI-HCO, the whole space is partitioned by a 3-order Hilbert (Figure 1(a)), and there are 8 Cloud nodes, namely, cn_1 to cn_8, which connect to each other, forming a Chord (Figure 1(b)). Hence, for cn_8, whose key is 10, its *charge-cell*s are cell 9 and 10, similarly, cn_2 (key=48) is responsible from cell 42 to 48.

Hilbert curve merely tries best to preserve locality of original space, hence, some spatial relation is lost after linearization, e.g., in Figure 1, cell 10 and cell 53 are adjacent to each other in original space, but they are distinguished by 43 in linear space, which consequently increases complexity of query processing. So we propose a vicinity-based approach to improve performance. Each Cloud node not only maintains finger table, but also connects to its vicinity neighbors, in particular, for each Cloud node cn_i, assuming the Chord key of it is key_i, then the vicinity neighbors are *charge-node*s of cell key_i's left, lower, right, upper cells, if the *charge-node* of left (lower, right, upper) cell is cn_i itself, then continue searching the *charge-node* of left (lower, right, upper) of left (lower, right, upper) cell, until the *charge-node* is not cn_i itself. Take cn_8 as an example in Figure 1(a), its Chord key is 10, and left, lower, right, upper cells of cn_8 are 11, 9, 53, 31, but cell 9 is cn_8's *charge-cell*, so cell 6 is selected, and the vicinity neighbors are cn_1, cn_4, cn_3 and cn_6.

4. PARALLEL PROCESSING

In this section, parallel processing for kNN query is presented. The main workflow is leveraging a series of range queries to accomplish kNN query. In particular, firstly, an initial radius r is calculated by k and estimation of spatial objects distribution[1][11], then a range query (q, r) is formed, i.e., to find objects located in the circular range centered at q, with the radius r.

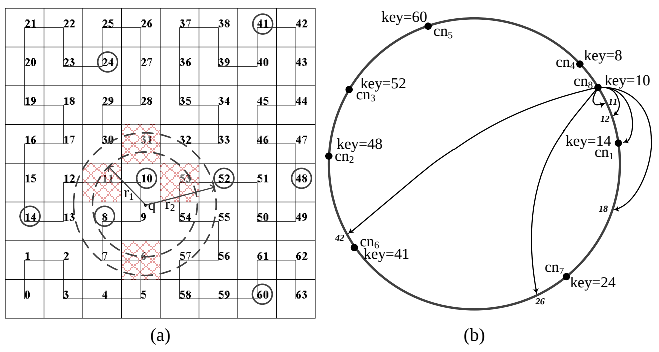


Figure 2: Parallel Processing

For processing such range query, first, the spatially nearest Cloud node to q, say cn_q, is found, and then according to global Hilbert partition policy, cn_q finds the set SC_q of cells intersecting with the circle, after that, the cells with contiguous Hilbert values are formed into intervals, i.e., SC_q={[c_{s_i}, c_{e_i}] | $s_i \le e_i$}, and then cn_q uses its vicinity neighbors to search these cells, in particular, for each interval [c_{s_i}, c_{e_i}], if there is a cell c_k being cn_q's left, lower, right or upper cell, then cn_q sends the interval to the corresponding vicinity neighbor, say cn_v, to search the results, otherwise, i.e., there is no cell being cn_q's neighbor cell, cn_q uses its predecessor, successor and/or finger table to find the proper *charge-node*s responsible for these cells to retrieve the results. Figure 2 illustrates an example, assuming cn_8 (key=10) is the nearest Cloud node to query point q, after obtaining radius r_1, cn_8 finds SC_q={[6, 8], 11, [30, 32], [53, 54], 57}, note that cell 9 and 10 are cn_8's *charge-cell*s, so they are not contained in SC_q. Then, for each interval (or value) in SC_q, say [6, 8], cn_8 examines that cell 6 is vicinity cell, so [6, 8] is sent to cn_4, similarly, 11, [30, 32] and [53, 54] are sent to cn_1, cn_6 and cn_5, respectively, and then cell 57 is routed through looking up finger table and is forwarded to cn_5.

After retrieving objects, cn_q examines the number of results, if it is not less than k, then cn_q sorts these objects according to the distance to q and returns results, otherwise, i.e., there are less than k objects, cn_q increases r by δ, a new range query (q, r+δ) is formed, and the intersecting cells are also calculated and sent to corresponding *charge-node*s. Then the *charge-node*s only search for the new coming cells and return results to cn_q, and again cn_q examines the number, and the above procedure is repeated until k objects are obtained. Continuing the example in Figure 2, when cn_8 finds that the number of results is less than k, it issues a new range query with radius r_2, and the corresponding cells are sent to *charge-node*s until k objects are retrieved, the iteration is terminated.

Algorithm 1 kNN Query Parallel Processing

```
Input:
    $q$=($x_q$, $y_q$), $k$
Output:
    $Qlist$   //result list
 1: $r = estimateR(k)$
 2: $cn_q = findNearestCN(q)$
 3: while true do
 4:     $SC_q = cn_q.findIntersectingCells(q, r)$
 5:     for each $inter_i \in SC_q$ do
 6:         if ($cn_v$=$cn_q.vNeighbor(inter_i)$)$\neq null$ then
 7:             $cn_q$.sendTo($cn_v$, $inter_i$)
 8:         else
 9:             $cn_q$.forward($inter_i$)
10:         end if
11:     end for
12:     $ret$=$cn_q.receiveResults()$
13:     if $|ret| \geq k$ then
14:         return $k$ nearest neighbors to $q$
15:     else
16:         $r = r + \delta$
17:     end if
18: end while
```

Algorithm 1 presents our parallel processing for kNN query. Line 1 calculates the radius r, and then the nearest Cloud node cn_q to q is found in line 2. From line 3 to the end, range queries are issued, and the returned results are examined until k objects are obtained. Node cn_q first finds the intersecting cells SC_q by (q, r) (line 4), and then for each interval in SC_q, cn_q uses vicinity neighbor (line 6 and 7) or finger table (line 8 and 9) to deliver the query messages, and then checks the number of returned results, until gets k objects.

5. SEQUENTIAL PROCESSING

In this section, we present the sequential processing for kNN query, which is a progressive procedure to obtain k objects. The difference is that, parallel processing is composed of a series of range queries, with constantly extending radius of the circular range to retrieve results, while for sequential processing, the query is progressively delivered to the corresponding Cloud nodes, i.e., the Cloud nodes receive the query in an order according to the distance to the query point. Although for parallel processing, query messages are almost delivered to different Cloud nodes at the same time, and then nodes process local search simultaneously, the critical point in parallel processing is the estimation of range radius r, i.e., with a large r, k results would be retrieved at one time, but some useless objects beyond k objects are also obtained, which promises waste of communication, while with a small r, some query messages would be repeatedly delivered to the same Cloud nodes, which also lower down the query performance. The advantage of sequential processing is that, without prior knowledge of k, the results are incrementally obtained according to sorting function, without wasting communication cost. For achieving sequential process, we propose to use histogram.

5.1 Design of Histogram

We use histogram to assist sequential processing. Each Cloud node maintains its own histogram, which is composed of several buckets. Each bucket is formatted with $\langle bid, range, num\rangle$, where *bid* identifies a bucket, and *range* denotes an interval marked with a starting value and an ending value of the Chord key domain, and *num* denotes the number of items in *range*. Different buckets' *range*s are non-overlapping with each other and all buckets' *range*s constitute the key domain.

For each Hilbert cell, its *charge-node* calculates the MBR (Minimum Bounding Rectangle) of objects intersecting with the cell, hence for each Cloud node, there is a set of MBRs corresponding to its *charge-cell*s. Relying on continuous maintaining routing tables of Cloud node[4], the MBRs are able to be disseminated to other Cloud nodes, i.e., the MBR information is piggy-back on the routing table maintenance message. Such a design would reduce the communication cost, thus the whole performance would not suffer. After constant message exchanging, each Cloud node is able to know the MBR for each Hilbert cell, and then for each bucket in the histogram, a bucket MBR (b-MBR) is built, which summarizes all objects in the bucket.

5.2 Processing Description

The main tool in sequential processing is priority queue[3], which is controlled by query issuing node, the node constantly fetches top element from the queue, and parses the element into query, and deliver the query to corresponding *charge-node*s. To clearly specify such interaction, we call the issuing node as requester, and the nodes receiving the query then replying are called responders. Next issue is that we consider both point objects and extent objects (the objects with spatial shape), so for an extent object, it may cross more than one cells, which means one object might be stored in different Cloud nodes, thus when query is sent to different nodes, the results returned may contain duplicates, hence the requester should detect duplicates and eliminate them. Another issue is that, when a responder cn_j receives query message, it would be costly for cn_j returning all objects maintained by it, hence cn_j should just return the objects with distance to q not larger than that of the current top element in the queue, however, such a fashion would lost cn_j forever, i.e., the rest objects in cn_j have no chance to be scanned. So we propose a concept called *map object*, which means this map object is just attached to the current results returned to requester, but it is not a result for this time, it might be a result in the following iterations, and its function is to lead the requester to send query to cn_j if necessary. In the following, we present the description for sequential processing, assuming that, requester cn_q issues a kNN query (q, k), the processing in requester and responders are described below:

(1) *Processing in Requester.* A min-priority queue PQ is used here to store three kinds of elements: b-MBR (bM), object (*obj*) and map object (*mobj*). The *mobj* is a structure sent from responder, representing an *obj* stored in some responder, and is formatted as $\langle dist, \{CloudNodes\}, range\rangle$, where *dist* is the distance between q and the *obj* being represented, and $\{CloudNodes\}$ is a list of identifiers of Cloud nodes which are responsible for the key interval *range*. Elements in PQ are sorted by distance, i.e., $MINDIST$(q, bM), distance between q and *obj* or *mobj.dist*. In order to break the tie that two elements have the same distance, we make a convention that *obj* is preferred to *mobj*, and *mobj* is preferred to bM. Initially, all bMs of cn_q are enqueued into PQ, in addition, a counter c storing the number of globally retrieved *obj*s so far is set to zero. After that, the top element e of PQ is dequeued.

- If e is a bM, cn_q uses vicinity-connection or finger table to route bM-request $\langle q, ctopdist, interval\rangle$, to the *charge-node*s (responders) of the cells in *interval*, where *ctopdist* is distance between current top element and q, and *interval* is the cell interval covered by e. Then the thread for operating PQ is suspended until cn_q receives a reply from responders. The reply may contain both *obj*s and *mobj*, and *obj*s are added into result list, and c is increased by the number of *obj*s, and then cn_q examines that, if c is larger than or equal to k, meaning that k nearest objects are found, cn_q informs all responders to terminate, otherwise, i.e., c is less than k, *mobj* is enqueued into PQ.
- If e is an *mobj*, cn_q sends *mobj*-request $\langle q, e.dist, ctopdist, e.range\rangle$ to the responders $e.\{CloudNodes\}$. Similarly, PQ is blocked until cn_q receives a reply and follows process above.
- Otherwise, i.e., e is an *obj*, e is added to result list and c is increased by one. If c is equal to k, cn_q informs all responders to terminate the processing.

(2) *Processing in Responder.* When a responder cn_j receives a request, depending on the type of the request, two cases are discussed below:

- If the request is a bM-request $\langle q$, *ctopdist*, *interval*$\rangle$, for each *obj* stored in cn_j, which is bounded in *interval*, cn_j calculates the distance between q and the *obj*. Only the *obj*s with the distance not larger than *ctopdist* are added to the return list. To eliminate local duplicates, we use a simple but effective method based on the observation that, for a bM-request, the responders are always located in a contiguous key set and in the relationship of predecessor and successor. Assuming that, three responders cn_1, cn_2 and cn_3 are located for a bM-request with the query key range [key_1, key_2]. In particular, [key_1, key_a) is maintained by cn_1, [key_a, key_b) is for cn_2 and [key_b, key_2] is for cn_3. After they finds qualified *obj*s, cn_1 sends its results to its successor cn_2, and cn_2 sends the results found by itself to cn_3 as well as the ones from cn_1. Now, cn_3 is able to use the identifiers of *obj*s to filter the duplicates among their findings, then a refined result list is determined. After that, cn_3 sends a reply to the requester, containing the refined results. To determine *mobj*, cn_1, cn_2 and cn_3 follow the collaborative way above to find an object *nobj* (next object) which is the immediate one to the last object in current result in ascending order by distance. After that, cn_3 sends a *mobj*-reply to the requester, $mobj$=$\langle nobj$.distance, $\{cn_1, cn_2, cn_3\}$, *range*$\rangle$.

- If the request is a *mobj*-request $\langle q$, *dist*, *ctopdist*, *range*$\rangle$, the responders search the *obj*s with the distances between *dist* and *ctopdist*, whose associated keys are in *range*. Then similar to the above processing, the responders aggregate the results and one of them sends a *mobj*-reply to the requester.

We briefly describe sequential query processing in Algorithm 2. From line 1 to line 18, the pseudo-codes show processing in requester, and the processing in responder is presented between line 20 and line 30.

Figure 3 shows an example for kNN query sequential processing. Assume that cn_6 in Figure 1 issues a 5-NN query specified by a spatial query point q. We illustrate three nearest bMs, $b\text{M}_1$ for the bucket with range [30, 32], containing obj_3, obj_4, with distance 0 to q; $b\text{M}_2$ for [8, 11], containing obj_1, obj_5, obj_6, with distance 2 to q; and $b\text{M}_3$ for [52, 55], containing obj_1, obj_2, with distance 3 to q. And the distances for $obj_1 \sim obj_6$ to q are 3, 7, 1, 4, 5, 2. Initially, cn_6 enqueues the three bMs, resulting $\langle b\text{M}_1(0), b\text{M}_2(2), b\text{M}_3(3)\rangle$, then $b\text{M}_1$ is dequeued, because cn_6 is the *charge-node* of cell 30 to 32, it searches locally, and finds obj_3 as the first result, and the *mobj* is obj_4, due to obj_4 belonging to cn_6 itself, so obj_4 is directly added into PQ, now the queue is $\langle b\text{M}_2(2)$, $b\text{M}_3(3)$, $obj_4(4)\rangle$. Next $b\text{M}_2$ is dequeued, and cn_6 sends a bM-request with *ctopdist*=3 ($b\text{M}_3$'s $MINDIST$) to the responders cn_4, cn_8 and cn_1 for [8, 11], then the reply is obj_6, obj_1 and $mobj_5$ (obj_5 is not a result for this iteration, because in_5's distance(=5) is larger than *ctopdist*(=3), it is the immediate object following obj_1 according to the distance to q), so cn_6 adds obj_6 and obj_1 into result list, and enqueues $mobj_5$, and increases c by 2, resulting 3. Similarly, cn_6 sends bM-request to cn_3 and cn_5 which are responsible for [52, 55], after searching, cn_5 replies the result obj_1 and $mobj_2$, because obj_1 is a duplicate, it is discarded, and $mobj_2$ is enqueued. Then obj_4 is inserted into result list, c is set to 4. After that, cn_6 processes $mobj_5$ and sends a

Algorithm 2 kNN Query Sequential Processing

```
Input:
    $q$=($x_q$, $y_q$), $k$
Output:
    $Qlist$   //result list
1: Requester $cn_q$'s Process:
2: $PQ=\phi$, $c$=0
3: $cn_q.enqueue(b\text{Ms}, PQ)$
4: while $PQ \neq \phi$ do
5:     $e=cn_q.dequeue(PQ)$
6:     if $e$ is type of $b$M then
7:         $cn_q.deliver(\langle q, ctopdist, interval\rangle)$
8:     else if $e$ is type of $mobj$ then
9:         $cn_q.deliver(\langle q, e.dist, ctopdist, e.range\rangle)$
10:    else
11:        $Qlist \leftarrow e$
12:    end if
13:    $cn_q$ updates $Qlist$ and $c$ when receives a reply
14:    if $c \geq k$ then
15:        $cn_q$ informs all related responders to terminate
16:        return $Qlist$
17:    end if
18: end while
19:
20: Responder $cn_j$'s Process:
21: $msg=cn_j.receive()$
22: if $msg$ is type of $b$M-request then
23:     $\langle q, ctopdist, interval\rangle=parse(msg)$
24:     $objlist=cn_j.search\&Filter(q, ctopdist, interval)$
25: else
26:     $\langle q, dist, ctopdist, range\rangle=parse(msg)$
27:     $objlist=cn_j.search\&FilterMObj(q, dist, ctopdist, range)$
28: end if
29: $objlist'$=duplicateEliminate($objlist$)
30: $cn_j.return(objlist')$
```

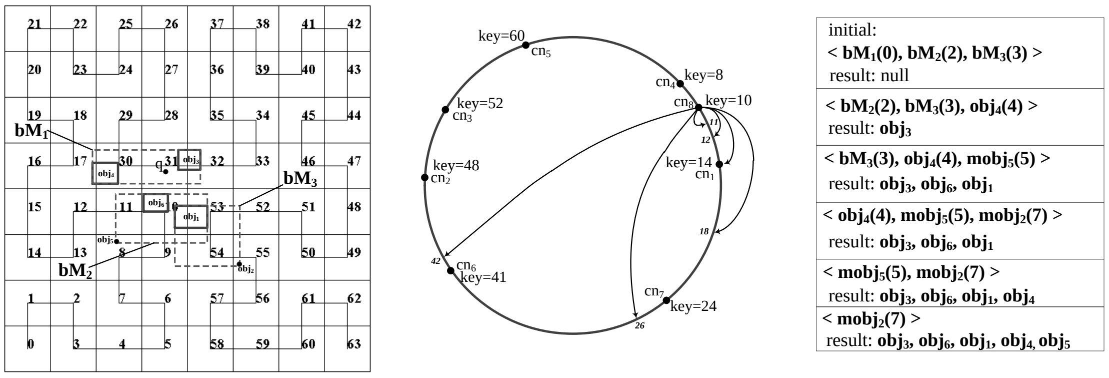
mobj-request to cn_4 with *dist*=5, *ctopdist*=7, and cn_4 finds obj_5 and replies to cn_6, then c is set to 5, which means the processing is terminated.

6. EXPERIMENTAL EVALUATION

We use two different datasets to evaluate our approaches. The first one is a synthetic dataset generated by GSTD[12] in space domain $[0, 1]^2$ and time domain $[0, 1]$. In particular, 500,000 objects, each specified by a two-dimensional rectangle, are initialized at t=0, with center coordinates in Gaussian distribution (mean=0.5 and variance=0.1) and size uniformly distributed in $[0, 0.1]^2$. As t evolves uniformly, the objects move from central part to the border of the space as well as resizing themselves, which totally results in almost 100 million records. Figure 4 visualizes the synthetic dataset (for the sake of legibility, only 5,000 objects are visualized). The second dataset is a real one[1] which records trajectories of taxis in Beijing from Nov. 1st to 3rd, 2013. In particular, each record in the dataset contains vehicle ID, geo-location, recording time stamp, etc. The amount of the records in the dataset is about 60 million.

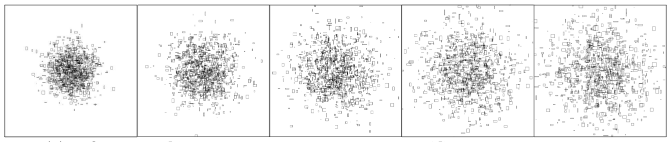
We implement VIHCO in Java and run it on a set of computing units. Each unit (Cloud node) is configured with

[1]http://activity.datatang.com/20130830/description



bM_1
bM_2
bM_3
q
obj_1
obj_2
obj_3
obj_4
obj_5
obj_6
key=60
cn_5
key=8
cn_4
cn_8
key=10
key=52
cn_3
key=48
cn_2
key=14
cn_1
cn_6
key=41
cn_7
key=24
11
12
18
26
42
initial:
$< bM_1(0), bM_2(2), bM_3(3) >$
result: null
$< bM_2(2), bM_3(3), obj_4(4) >$
result: obj_3
$< bM_3(3), obj_4(4), mobj_5(5) >$
result: obj_3, obj_6, obj_1
$< obj_4(4), mobj_5(5), mobj_2(7) >$
result: obj_3, obj_6, obj_1
$< mobj_5(5), mobj_2(7) >$
result: $obj_3, obj_6, obj_1, obj_4$
$< mobj_2(7) >$
result: $obj_3, obj_6, obj_1, obj_4, obj_5$

Figure 3: Sequential Processing



(a) t=0 (b) t=0.25 (c) t=0.5 (d) t=0.75 (e) t=1

Figure 4: Visualization of Synthetic Dataset

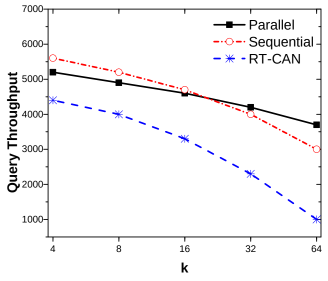
Intel(R) Core2 Quad 2.5GHz processor, 2GB memory and 500GB disk. The number of nodes varies from 4 to 16. And they are connected via 1Gbps network links. We use RT-CAN for comparison and query throughput (number of queries processed per second) as the metric in the experiments. And we vary k and network size to measure the performance, which are 16 and 8 as default, respectively.

6.1 Results on Synthetic Dataset

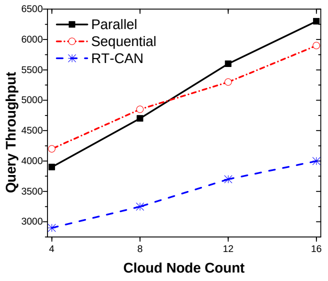
Figure 5 shows the performances of three query processing methods on synthetic dataset. When we increase k from 4 to 64, both performances degrade, which can be explained that a larger k involves more Cloud nodes to inspect distances from objects to the query point, hence messages and comparisons are raised. However, we can see from Figure 5(a), RT-CAN degrades faster than our two processing methods, this is due to: first, VIHCO uses not only long-links (finger table) to find *charge-nodes*, but also vicinity neighbors to quickly locate nearby nodes, second, our parallel processing method takes differential cells between two consecutive range queries into consideration, which reduces communication cost, on the other hand, our sequential method is able to elaborately forward query to the related Cloud nodes, while RT-CAN does not take these details into consideration.

For comparing parallel and sequential method, we can see from Figure 5(a), when k is small, sequential processing outperforms parallel one, while for a large k, the result is opposite. This can be explained that when k is small, there are fewer objects to be scanned, and parallel processing does not exploit parallelism of Cloud nodes sufficiently, on the other hand, sequential method is able to deliver messages accurately, any no-related node would not be contacted, hence the communication cost is reduced. While with a large k, the disadvantage of sequential method increases, i.e., Cloud nodes receive query messages one by one, and for parallel method, the query range is enlarged, more nodes are able to search results simultaneously, the performance is raised.

Next, we can see the comparison results from Figure 5(b) under different system sizes. A small size gives better performance to sequential processing and a large size gives to parallel one. The reason is similar to the previous one.



(a) Effect of k

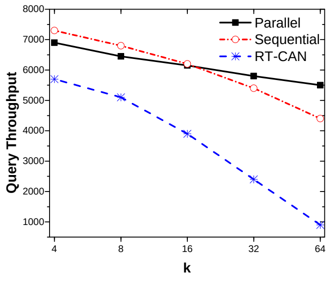


(b) Effect of system size

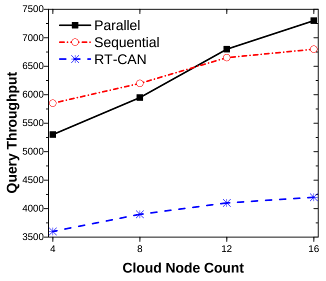
Figure 5: Results on Synthetic Dataset

6.2 Results on Real Dataset

Figure 6 shows the results on real dataset. The advantage of VIHCO and our query processing methods are definitely revealed in the results, where RT-CAN deteriorates more seriously with k increased. A detail observation is that the throughput of RT-CAN is only about 1,000 when k is equal to 64, while our methods are above 4,000. We can see VIHCO is more suitable for the skewed dataset than RT-CAN for kNN query.



(a) Effect of k



(b) Effect of system size

Figure 6: Results on Real Dataset

7. CONCLUSIONS

With the increasing of DaaS (Database as a Service) requirement, kNN query processing methods would be paid more attention by both academic community and industrial circles. In this paper, we propose an interesting problem, which is with better performance for Cloud computing, parallel method or sequential one. For answering such a question, we first devise an distributed indexing structure VIHCO for constructing overlay network, featured by vicinity-connection which is capable to fast locate the destinations. Upon VIHCO, we present the algorithms of parallel and sequential processing, and explore the answers through experiments both on synthetic and real dataset, and the results show that our VIHCO is better than RT-CAN, and the sequential method is more efficient under small k query condition and small system size, while parallel one suits for large k and large scale of computing nodes.

In the future, we plan to extend our work to in high dimensional space and road network space.

8. ACKNOWLEDGMENTS

This work is supported by NSF of China grant 61303062 and 71331008. We would like to thank Prof. Dai and Dr. Hu for helping with the proof.

9. REFERENCES

[1] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 78–86. ACM, 1997.

[2] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu. A framework for supporting dbms-like indexes in the cloud. *Proceedings of the VLDB Endowment*, 4(11):702–713, 2011.

[3] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[4] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[5] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. 1993.

[6] R. Lange, F. Dürr, and K. Rothermel. Scalable processing of trajectory-based queries in space-partitioned moving objects databases. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, page 31. ACM, 2008.

[7] H. Lu, Z. Huang, C. S. Jensen, and L. Xu. Distributed, concurrent range monitoring of spatial-network constrained mobile objects. In *Advances in Spatial and Temporal Databases*, pages 403–422. Springer, 2007.

[8] A. Meka and A. Singh. Dist: a distributed spatio-temporal index structure for sensor networks. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 139–146. ACM, 2005.

[9] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: a scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[11] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *Knowledge and Data Engineering, IEEE Transactions on*, 16(10):1169–1184, 2004.

[12] Y. Theodoridis, J. R. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *Advances in Spatial Databases*, pages 147–164. Springer, 1999.

[13] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 591–602. ACM, 2010.

[14] C. Zhang, X. Chen, B. Ge, and W. Xiao. Indexing historical spatio-temporal data in the cloud. In *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015.