Mobile Information Systems 8 (2012) 107–126
DOI 10.3233/MIS-2012-0134
IOS Press

A distributed approach to continuous monitoring of constrained k-nearest neighbor queries in road networks

Hyung-Ju Cho*, Seung-Kwon Choe and Tae-Sun Chung
Department of Computer Engineering, Ajou University, Gyeonggi-Do, South Korea

Abstract. Given two positive parameters k and r, a constrained k-nearest neighbor (CkNN) query returns the k closest objects within a network distance r of the query location in road networks. In terms of the scalability of monitoring these CkNN queries, existing solutions based on central processing at a server suffer from a sudden and sharp rise in server load as well as messaging cost as the number of queries increases. In this paper, we propose a distributed and scalable scheme called DAEMON for the continuous monitoring of CkNN queries in road networks. Our query processing is distributed among clients (query objects) and server. Specifically, the server evaluates CkNN queries issued at intersections of road segments, retrieves the objects on the road segments between neighboring intersections, and sends responses to the query objects. Finally, each client makes its own query result using this server response. As a result, our distributed scheme achieves close-to-optimal communication costs and scales well to large numbers of monitoring queries. Exhaustive experimental results demonstrate that our scheme substantially outperforms its competitor in terms of query processing time and messaging cost.

Keywords: Distributed algorithm, continuous monitoring, constrained k-NN query, road network

1. Introduction

The increased popularity of mobile communication devices with embedded positioning capabilities (e.g., GPS) has triggered the development of many location-based applications. The ability to support location-based queries from mobile clients on a road network is essential for this class of mobile applications.

In this work, we investigate the continuous monitoring of k nearest neighbor queries over static objects within a query distance r of the current location where the k and r values are provided by the clients. A location based query that reports the k nearest neighbors within a query distance r of a moving query point q is called a constrained $k-$nearest neighbor (CkNN) query [8,9,12]. These queries are often useful to find specific objects (e.g., gas stations reachable in 10 minutes driving) within some specific region. Consider the example of a family traveling by car. They may want to continuously monitor the three closest restaurants within 5 km of their current location so that they can choose a restaurant that serves their favorite food.

This paper assumes that (1) query points move freely on the road network and any mobility pattern (e.g., speed, direction, route, etc.) on them is not given, (2) objects (e.g., gas stations, restaurants) of

*Corresponding author: Department of Computer Engineering, Ajou University Woncheon-dong, Suwon Si Yeongtong-gu, Gyeonggi-Do, 443-749, South Korea. Tel.: +82 31 219 2535; Fax: +82 31 219 1834; E-mail: hjcho@ajou.ac.kr.

interest to the clients do not change their location on the road network, and (3) a central server and each client talk to each other using a wireless channel such as cellular services or Wi-Fi. Last, like most on-line monitoring systems, main-memory evaluation is exploited.

Continuous monitoring of k-nearest neighbor queries and range queries on the road network has been well studied [5,14,18,21,23,26,27,29]. In real-life scenarios where (1) a large and rapidly growing number of monitoring queries are handled and (2) complex queries are processed, existing solutions which rely on central processing at the server suffer from high server loads and messaging costs [3,10, 22]. We address this issue by proposing a *d*istributed and scalable scheme called DAEMON which is tailored for the continuous *mon*itoring of CkNN queries in the road network.

Our distributed processing of CkNN queries consists of server side processing and client side processing. The server is dedicated to efficiently processing CkNN queries issued at intersections and quickly retrieving the objects in the road segments between neighboring intersections. The clients are devoted to making their own query result using the information provided by the server. Our proposed scheme is distinct from its rivals which largely depend on the capabilities of a centralized server. First, in our scheme, clients do not register their continuous queries to the server. Second, the clients do not send their location information to the server, which helps ease privacy issues regarding the locations of users [1, 19]. Last, the server burden is dramatically decreased by shipping part of the query processing to the clients [3,10,22].

At the server side, a shared execution technique is employed to decrease the processing time of queries issued at intersections of road segments. Among all nodes, a small portion of them, which we call condensing nodes, are carefully selected using query results. Each condensing node keeps information on the objects within an observation distance R. Finally, using the server response, clients generate the query results for the road segments which they are currently at.

Our principal contributions are summarized as follows:

- We present a distributed and scalable scheme called DAEMON for continuous monitoring of CkNN queries in road networks.
- Our scheme minimizes server load and messaging cost by shipping part of the query processing to the clients. The server neither exploits nor stores movement information (e.g., route or location updates) of the clients. This enables the server to run this service on-demand and mitigates location privacy issues.
- We employ the shared execution technique called condensing nodes to reduce the processing time at the server. This leads to a significant reduction in the evaluation time of queries issued at intersections of road segments.
- We conduct an exhaustive performance evaluation to show that DAEMON is markedly superior to its competitor under various conditions while achieving close-to-optimal communication costs.

The rest of the paper is structured as follows: Section 2 briefly reviews related work on k NN algorithms and query monitoring. Section 3 presents preliminaries and formulates the problem for this work. Sections 4 to 6 elaborate on our distributed approach to continuous monitoring of CkNN queries on the road network. Thorough experimental results are given in Section 7. Finally, Section 8 concludes the paper.

2. Related work

For location-based services, processing k-nearest neighbor (kNN) queries on the road network has been well studied (e.g. [1,5,6,8,9,12,14,18,20,27,28]). For a good survey on location based query processing, refer to [14]. Papadias et al. proposed the incremental network expansion (INE) algorithm for kNN queries. The basic idea of this algorithm is to incrementally search the road segments from the query point until the k-nearest objects of the query point are found [20]. The performance of the INE algorithm is closely related to the density of objects in the network. Hence, the INE suffers dramatic degradation in performance when objects are sparsely distributed. To overcome this limitation, many kNN query processing algorithms which utilize the pre-computed network distance to optimize the query processing (e.g. [1,6,27]) have been proposed. Bao et al. introduced a new type of query called a k-range nearest neighbor (kRNN) query to find the k closest objects of every point on the road segments inside a given query region based on the network distance [1]. They presented an algorithm which exploits a shared execution paradigm to eliminate the redundant searching overhead among adjacent nodes. Ghadiri et al. presented two fuzzy clustering methods for group nearest neighbor queries based on fuzzy logic distance model [11].

There is a large body of research work on continuous monitoring of spatial queries in Euclidean spaces. Cheema et al. present a safe zone based approach to continuous monitoring of distance-based range queries that return the objects within a distance r of the query location [4]. Mokbel et al. proposed an algorithm called SINA for evaluating a set of concurrent spatial queries in order to reduce the overall computational cost using shared execution and incremental evaluation [15]. Distributed approaches have also been investigated to monitor continuous range queries [3,10] and continuous kNN queries [22]. The main idea of these distributed schemes is to shift some of the load from the server to mobile clients. Morvan et al. proposed a mobile relational algebra to decentralize the control of dynamic query optimization processes [17]. Hasan et al. presented algorithms which are applicable to any arbitrarily-shaped constrained region for continuous monitoring of CkNN queries [12] and Gao et al. presented algorithms for processing CkNN searches on the trajectories of moving objects [9]. However, the techniques based on Euclidean distance are not applicable to our problem concerning network-constrained mobile clients and network distance-based queries.

There is also a large body of research on continuous monitoring of spatial queries for road networks. Chen et al. dealt with path k-NN queries that return kNNs with respect to the shortest path connecting the destination and the user's current location [5]. Stojanovic et al. proposed a technique for continuous monitoring of range queries over moving objects [21]. The range of the query may be defined by a user selected area, a map window, a polygon, a circle or a part of a road segment. Recently, Xuan et al. proposed several Voronoi-based algorithms for continuous range queries [23–25] as well as continuous k NN queries [27]. Their algorithms may suffer from high computational cost at the server and high communication cost as the number of monitoring clients increases rapidly.

Mouratidis et al. presented two algorithms, IMA and GMA, for the continuous nearest neighbor monitoring problem in road networks. In this problem, both queries and objects of interest move freely [18]. GMA integrates IMA with the shared execution paradigm. Specifically, GMA groups together the queries falling on the path between two consecutive intersections in the network and monitors the k-NNs of the intersections. It then utilizes query results at the intersections in order to facilitate evaluation of the queries on the path. It has been shown that GMA is typically better than IMA. Therefore, we focus mainly on GMA in the paper.

Even if GMA and IMA can be extended to continuously monitor CkNN queries over static objects, they are not very efficient because they are not specifically designed for monitoring these CkNN queries. GMA demonstrates the following disadvantages in terms of scalability. (1) The query processing of GMA is centralized at the server, which is in sharp contrast to our distributed scheme. Hence, in GMA, the server burden grows rapidly with the number of monitoring queries. (2) GMA requires that clients install their continuous queries on the server and report their changed location periodically in order for the server to capture the status (e.g., location and query conditions) of each client. None of both are demanded by our scheme. These requirements may raise location privacy problem. (3) Frequent location updates greatly increase the server load. Whenever clients move and their movements are reported to the server, the queries are re-evaluated at the server. If query results are updated, the updated results then need to be delivered to the clients through a wireless network. In addition, the server may not be able to cope with the high location report rate which is necessary to ensure accurate answers.

3. Preliminaries

In this section, we describe the road network, system assumptions, and terms used in the paper and present the formal definition of our problem.

3.1. Road network and system assumptions

We model the underlying road network as a weighted undirected graph $G = (N, E)$ where N is a finite set of nodes, E is a set of edges (i.e., road segments), and $E \in N \times N$. Each edge is given the length of its corresponding road segment as a weight.

In this work, we consider our system to be a mobile environment where the mobile clients can communicate with a central server through some wireless communication infrastructure (e.g., cellular services or Wi-Fi). The clients are equipped with positioning technology like GPS and can locate their positions. They also have sufficient capabilities to carry out computational tasks. All these assumptions are either widely agreed upon or are seen as common practice in most existing mobile systems in the context of the monitoring and tracking of moving objects [3,7,10,13,16,22].

3.2. Definitions of terms and problem

To clarify the meanings of the primary terms used in this paper, we present the terms and their formal definitions. Nodes can be categorized according to node degree. (1) If a node degree is larger than 2, the node is referred to as an intersection node. (2) If a node degree is 2, the node is referred to as an intermediate node. (3) If a node degree is 1, the node is referred to as a terminal node. A certain amount of intersection nodes are chosen and referred to as condensing nodes. Each condensing node is used to store the information (i.e., object's id, location, and distance to the condensing node) about the objects that lie within its observation distance R.

A sequence denotes a path between two nodes n_s and n_e, such that n_s (or n_e) is either an intersection node or a terminal node and the other nodes in the path are intermediate nodes. Two end nodes n_s and n_e of a sequence are called the boundary nodes. Particularly, the sequence where a query point q remains is called the (current) active sequence of q. Table 1 presents the primary symbols used in this paper and their definition.

Using the network in Fig. 1, we formulate our continuous monitoring problem. In this figure, objects a, b, c represent entities of interest and the query point q corresponds to a client that requests continuous monitoring of the k closest objects within a query distance r of its current location. The numbers above the road segments indicate the network distances of two adjacent objects or nodes. That is, $d(n_2, a) = d(n_6, c) = 1.5$ and $d(n_2, b) = d(b, n_5) = 1$. Suppose that $k = 2$ and $r = 1$ are given and q moves along sequence $SQ_{(n_2,n_5,n_6)}$. Then, while q runs between n_2 and n_5, the CkNN query result

Table 1
Primary symbols and their definition

Symbol	Definition
$SQ_{(n_i,n_{i+1},\ldots,n_j)}$	A sequence where n_s (or n_e) is the start (or end) boundary node and the others $n_{s+1},\ldots,n_{e-1}$ are intermediate nodes. Start and end boundary nodes are decided by the movement direction of query point
$d(a,b)$	The shortest path length between object a and object b
q	Query point on the road network (e.g., the current location of a vehicle)
Aq	The set of objects satisfying query conditions at location q
k	Number of nearest neighbors requested by CkNN query
r	Network distance requested by CkNN query
R	Observation distance of condensing node

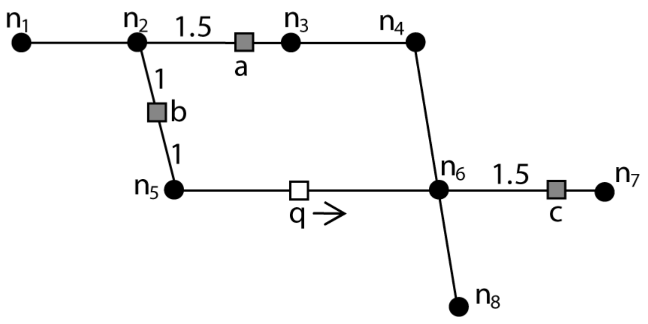
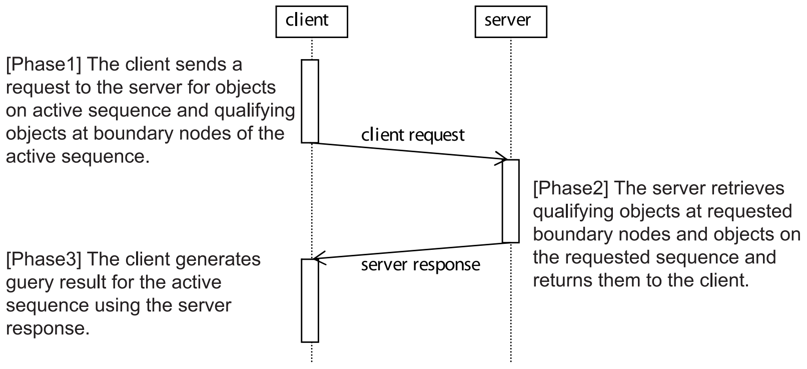


Fig. 1. Simple road network for problem definition.



[Phase1] The client sends a request to the server for objects on active sequence and qualifying objects at boundary nodes of the active sequence.

[Phase2] The server retrieves qualifying objects at requested boundary nodes and objects on the requested sequence and returns them to the client.

[Phase3] The client generates guery result for the active sequence using the server response.

Fig. 2. Interaction diagram between client and server.

is object b and while q runs between n_5 and n_6, the query result is empty. Since we do not assume any knowledge about the movement patterns of queries, the next active sequence is not revealed before the query point actually gets to it. On reaching n_6 which is the end of the current active sequence $SQ_{(n_2,n_5,n_6)}$, q may choose any of the next active sequence candidates $SQ_{(n_6,n_4,n_2)}$, $SQ_{(n_6,n_7)}$, $SQ_{(n_6,n_8)}$.

4. Continuous monitoring of constrained k NN queries

Section 4.1 gives an overview of our scheme which consists of three phases as illustrated in Fig. 2. Sections 4.2, 5, and 6 elaborate on the first, second, and third phase, respectively.

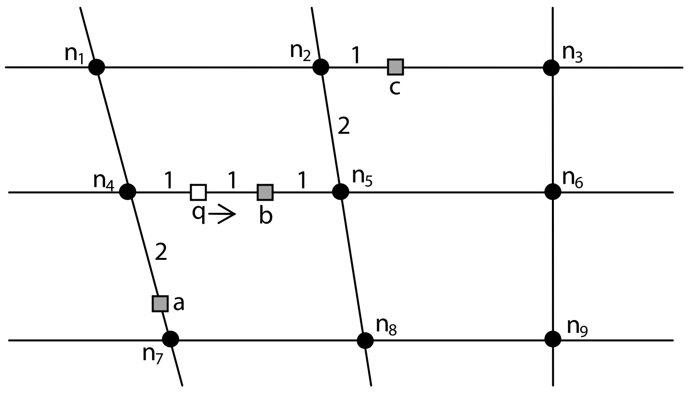


Fig. 3. Example road network and CkNN query q with $k = 2$ and $r = 3$.

4.1. Overview

Figure 2 depicts the interaction diagram between the client and server. It is important to note that while a client travels along the active sequence, an interaction which consists of the client request and server response is sufficient for the client to generate the query result for the active sequence. Naturally, this interaction is repeated whenever the client reaches a new active sequence. To simplify the presentation, we focus on the interaction between a client and the server while the client runs on the active sequence. The three phases of Fig. 2 are fully described in Sections 4.2, 5, and 6, respectively.

In the first phase, the client asks the server for the objects on the active sequence and the qualifying objects which meet query conditions at each boundary node of the active sequence. Note that the client does not provide its location information to the server.

In the second phase, the server processes the client request and returns the response through the wireless network. The request consists of the following two tasks: (1) evaluating CkNN queries issued at the boundary nodes of the active sequence and (2) retrieving the objects on the active sequence. After the processing at the server is complete, the processed result is returned to the client over the wireless channel.

In the final phase, using the server response, the client generates the query result for the active sequence. The query result consists of valid intervals and their corresponding qualifying objects where a valid interval indicates a part of the travel path where a set of qualifying objects remains unchanged.

The characteristics of our proposed architecture can be summarized as follows: First, the server is tailored to efficiently evaluate CkNN queries issued at boundary nodes and to rapidly retrieve the objects on the active sequences. The clients are devoted to quickly generating their query results using the server responses. Such distributed processing greatly reduces the server burden and communication costs. Second, each client sends only one request to the server while on a particular active sequence. This achieves satisfactory communication costs between the clients and the server in most cases. Last, the clients do not disclose their location information to the server, which alleviates privacy issues regarding location information.

4.2. Sending client request to server

In the rest of the paper, we utilize the example road network in Fig. 3 to explain the three phases of Fig. 2. In Fig. 3, query q with $k = 2$ and $r = 3$ moves along the active sequence $SQ_{(n_4,n_5)}$.

The numbers above the road segments represent the distances between two adjacent objects. That is, $d(n_4, a) = d(n_4, b) = d(n_2, n_5) = 2$ and $d(n_5, b) = d(n_2, c) = 1$.

Each client request is in a form $< qid, k, r, \mathit{boundary\_node}(s), \mathit{sequence} >$ where *qid* indicates query id (e.g., vehicle id), k and $r$indicate query conditions detailed in Table 1, $\mathit{boundary\_node}(s)$ represent boundary nodes of the active sequence, and the last attribute *sequence* represents the active sequence.

Recall that the exemplary CkNN query q of Fig. 3 requests continuous monitoring of two NNs within three (kilometers) of its location. The first request $< q, 2, 3, \{n_4, n_5\}, SQ_{(n_4,n_5)} >$ is generated and transferred to the server. Soon, the client receives the response from the server and makes its own query result for $SQ_{(n_4,n_5)}$. In the case where the client turns left at boundary node n_5, the second request $< q, 2, 3, \{n_2\}, SQ_{(n_5,n_2)} >$ is transferred to the server. Observe that n_5 is not included in the fourth attribute (i.e., $\{n_2\}$) of the second request. The reason for this is that the response for the previous active sequence $SQ_{(n_4,n_5)}$ is cached at the client and recycled. Specifically, the information (i.e., object's id, location, and distance to n_5) on the qualifying objects at n_5 is re-used by the client.

5. Processing client request at the server

Section 5.1 describes our CkNN search algorithm which exploits condensing nodes and sequences. Section 5.2 discusses the updates of objects such as insertion and deletion of objects.

5.1. CkNN search algorithm

Upon receiving the client request, the server starts to process the request. Recall that the client request consists of two tasks. One is to evaluate queries issued at the boundary nodes of the active sequence and the other is to retrieve the objects on the active sequence. To process the client request effectively, we introduces condensing nodes and sequences both of which significantly reduce the computation time of the server. Condensing nodes and sequences are stored in the condensing node table CT and the sequence table ST, respectively. Both of CT and ST are hash tables on node id and sequence id, respectively.

Condensing nodes are generated and updated using query results at runtime. Each condensing node stores information about the objects within its observation distance R. This results in reducing the network expansion from the condensing node. As a result, the condensing nodes enable the server to quickly evaluate queries issued at boundary nodes. Each entry in condensing node table CT is in a form <*nid*, *loc*, R, O, *adj_seq*> where *nid* is a node id for a condensing node n_c, *loc* indicates the position of n_c, R denotes the observation distance of n_c, *adj_seq* denotes the set of adjacent sequences of n_c and O denotes the set of pairs $< \mathit{obj}, d(n_c, \mathit{obj}) >$ where *obj* is an object such that $d(n_c, \mathit{obj}) \leqslant n_c.R$.

Each sequence has the information on the objects on the sequence. We note that the total number of sequences is typically smaller than that of edges in the network since a sequence may include multiple edges. Each entry in sequence table ST is in a form $< \mathit{seqid}, (n_s, n_{s+1}, \ldots n_e), O_{(n_s,n_{s+1},\ldots n_e)}, \mathit{len} >$ where *seqid* is a sequence id for $SQ_{(n_s,n_{s+1},\ldots,n_e)}$, n_s and n_e represent the boundary nodes, $(n_s, n_{s+1}, \ldots n_e)$ is a list of ids of nodes contained in the sequence, $O_{(n_s,n_{s+1},\ldots n_e)}$ represents the set of pairs $< \mathit{obj}, d(n_s, \mathit{obj}) >$ where *obj* is an object on $SQ_{(n_s,n_{s+1},\ldots,n_e)}$, and *len* is the sequence length which is the total weight of edges in the sequence. The server can quickly retrieve the objects on the requested sequence by using ST. In the rest of this section, we describe our CkNN search algorithm to rapidly evaluate queries issued at boundary nodes.

The CkNN search algorithm does two jobs. One is to evaluate a CkNN query and the other is to update with query result the information of the boundary node n_q which corresponds to query location

```
Algorithm 1: C$k$NN _search ($q$,$q.k$,$q.r$)
Input: $q$: a query point, $q.k$: the number of NNs, $q.r$: query distance
Output: a set of objects satisfying query predicate and their distance to $q$
1:  $kth\_dist \leftarrow q.r$    /* $kth\_dist$ is initialized to the $q.r$ value */
2:  $q.A \leftarrow \phi$    /* answer set $q.A$ keeps the objects satisfying query predicate */
3:  $PQ.push(q, 0)$    /* $PQ$ is a priority queue. note that $d(q, q) = 0$ */
4:  while $PQ$ is not empty do
5:      $(n_{top}, d(q, n_{top})) \leftarrow PQ$.pop()
6:      Mark $n_{top}$ as visited in order to avoid multiple visits.
7:      if $d(q, n_{top}) \leqslant kth\_dist$ then
8:          search_boundary_node ($n_{top}, d(q, n_{top}), kth\_dist$)
9:      else /* it means that $d(q, n_{top}) > kth\_dist$ */
10:         exit while loop
11: /* recall that $n_q$ is the boundary node which corresponds to $q$ */
12: if $n_q$ is an intersection node and $kth\_dist > n_q.R$ then
13:     if $kth\_dist \leqslant R_{\max}$ then    $n_q.O \leftarrow q.A$,    $n_q.R \leftarrow kth\_dist$
14:     else /* it means that $kth\_dist > R_{\max}$ */
15:         $n_q.R \leftarrow R_{\max}$    /* note that $n_q.R$ is set to $R_{\max}$ */
16:         for each object $o \in q.A$ do
17:             if $d(n_q, o) \leqslant R_{\max}$ then    $n_q.O \leftarrow n_q.O \cup \{(o, d(n_q, o))\}$
18: return $q.A$
```

if necessary. Note that query evaluation is described in lines 1 to 10 of the algorithm and boundary node update is described in lines 11 to 17.

The distance *kth_dist* between q and its kth NN o_{kth} is initialized to the query distance $q.r$ value. First, the tuple $(q, 0)$ is pushed to the priority queue *PQ* for a network expansion. Note that a tuple of *PQ* consists of a node to be searched and its distance to q. If $PQ = \phi$ or $d(q, n_{top}) > kth\_dist$, the algorithm exits the while loop. Otherwise, n_{top} and its adjacent sequences are searched iteratively.

If the boundary node n_q is an intersection node and $n_q.R < kth\_dist$, $n_q.R$ and $n_q.O$ are updated using *kth_dist* and query answer $q.A$, respectively. Note that terminal nodes and intermediate nodes cannot be condensing nodes. Observation distance of each boundary node is initialized to 0 and cannot exceed the maximum observation distance $R_{\max}$ value which is given as a system parameter. If $n_q.R < kth\_dist \leqslant R_{\max}$, $n_q.R$ and $n_q.O$ are updated to *kth_dist* and $q.A$, respectively. If $kth\_dist > R_{\max}$, $n_q.R$ is set to the R_{max} value and $n_q.O$ is updated to the set of objects o such that for each object $o \in q.A$, $d(q, o) \leqslant R_{\max}$. Condensing nodes are kept in memory and sorted in a least recently used (LRU) list. The last entry is removed if there is no space available to accommodate a new condensing node.

The Search_boundary_node algorithm explores the objects within the observation distance of n_{top} and its adjacent sequences with an offset $d(q, n_{top})$. If r is a condensing node, the algorithm first investigates each object o in $n_{top}.O$. If $d(q, n_{top}) + d(n_{top}, o) \leqslant kth\_dist$, a tuple $(o, d(q, o))$ is added to answer set $q.A$ and *kth_dist* is updated to the value of $d(q, o_{kth})$ if $o_{kth} \in q.A$. Next, each adjacent sequence of r is explored. If n_{top} is a condensing node and its observation distance is not smaller than the length of an adjacent sequence of n_{top}, the adjacent sequence is skipped since the objects on the sequence have already been investigated. Otherwise, the adjacent sequence should be explored. Finally, among adjacent boundary nodes of r, the nodes which have not been explored are identified and are pushed to *PQ*.

The Search_sequence algorithm examines the objects on an adjacent sequence of n_{top}. For each object o on the adjacent sequence, we check if $d(q, o) \leqslant kth\_dist$. If so, a tuple $(o, d(q, o))$ is added to $q.A$ and *kth_dist* is updated to the value of $d(q, o_{kth})$. When the CkNN search algorithm ends, the qualifying objects in $q.A$ and their information (e.g., location and distance to q) are returned to the client.

```
Algorithm 2: Search_boundary_node ($n_{top}, d(q, n_{top})$, kth_dist)
Input: $n_{top}$: a node popped from PQ, $d(q, n_{top})$: an offset, kth_dist: distance from $q$ to its $k$th NN
Output: a set of objects satisfying query predicate at $n_{top}$ and its adjacent sequences
1:      /* let $n_{top}.O$ be the set of objects within observation distance of $n_{top}$ */
2:      if $n_{top}$ is a condensing node then      /* if $n_{top}.R > 0$, $n_{top}$ is a condensing node */
3:          for each object $o \in n_{top}.O$ do /* if $n_{top}$ is a condensing node, each object in $n_{top}.O$ is examined */
4:              if $d(q, n_{top}) + d(n_{top}, o) \leqslant$ kth_dist then
5:                  $q.A \leftarrow q.A \cup \{(o, d(q, n_{top}) + d(n_{top}, o)\}$
6:                  kth_dist $\leftarrow d(q, o_{kth})$ where $o_{kth} \in q.A$
7:      for each adjacent sequence $SQ_{(n_{top}, n_j)} \in n_{top}.adj\_seq$ do
8:          if $n_{top}$ is a condensing node and $SQ_{(n_{top}, n_j)}.len \leqslant n_{top}.R$ then
9:              skip the visit to $SQ_{(n_{top}, n_j)}$
10:         else
11:             Search_sequence ($SQ_{(n_{top}, n_j)}, d(q, n_{top})$, kth_dist)
12:         if $n_j$ is not marked as visited then    PQ.push $(n_j, d(q, n_{top}) + d(n_{top}, n_j))$
13:     return $q.A$
```

```
Algorithm 3: Search_sequence ($SQ_{(n_{top}, n_j)}, d(q, n_{top})$, kth_dist)
Input: $SQ_{(n_{top}, n_j)}$: an adjacent sequence of $n_{top}$, $d(q, n_{top})$ and kth_dist have the same meanings as before
Output: a set of objects satisfying query predicate on an adjacent sequence of $n_{top}$
1:      for each object $o \in O_{(n_{top}, n_j)}$ do
2:          if $d(q, n_{top}) + d(n_{top}, o) \leqslant$ kth_dist then
3:              $q.A \leftarrow q.A \cup \{(o, d(q, o))\}$
4:              kth_dist $\leftarrow d(q, o_{kth})$ where $o_{kth} \in q.A$
5:      return $q.A$
```

5.2. Discussion on updates of objects

Finally, we discuss the updates in the set of objects even if the updates occur rarely. There are three types of updates to a set of objects: an object addition (e.g., opening of a gas station), an object deletion (e.g., closing of a gas station), and an object movement. Object movement can be treated as the deletion and addition of the object. Such changes are easily reflected to the CT and ST. Consider that a new object o_{new} is added. It is necessary that the sequence and condensing nodes which are affected by the addition of o_{new} are identified. To do this, we simply find the sequence including o_{new} and the condensing nodes within the maximum observation distance R_{max} from o_{new}. Finally, new entries for the object o_{new} are appended to the affected sequence and condensing nodes. The object deletion is handled in a similar way to the object addition.

6. Generating query result for active sequence at the client

Section 6.1 describes the distance graph between a moving query point and an object. Section 6.2 elaborates on decision of valid intervals and their query answer.

6.1. Distance graph between moving query point and qualifying object

Using the response from the server, the client generates valid intervals and their answer for the active sequence. We continue to use the example of Fig. 3. Now the client has query results (i.e., A_{n_4} and A_{n_5}) at both boundary nodes of the active sequence and the information on the objects (i.e., $O_{(n4,n5)}$) on the active sequence. Specifically, $A_{n_4} = \{< obj, d(n_4, obj) > | q(obj, n_4) \leqslant 3\} = \{< a, 2 >,$

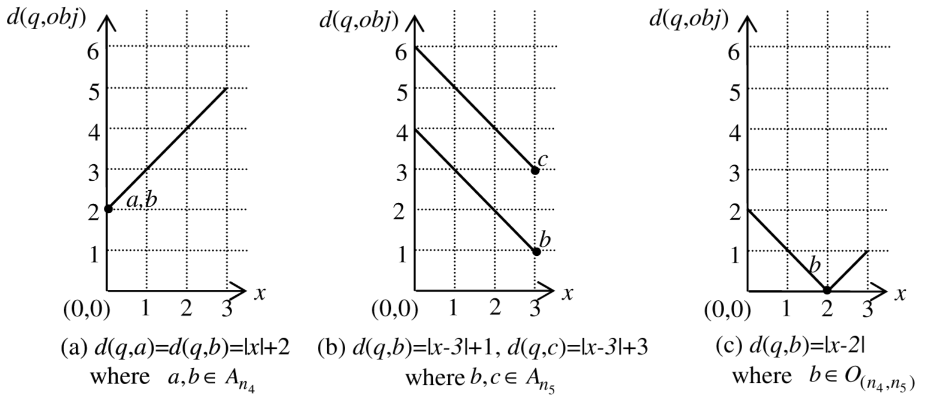


Fig. 4. Distance graph between q and each of a, b, c.

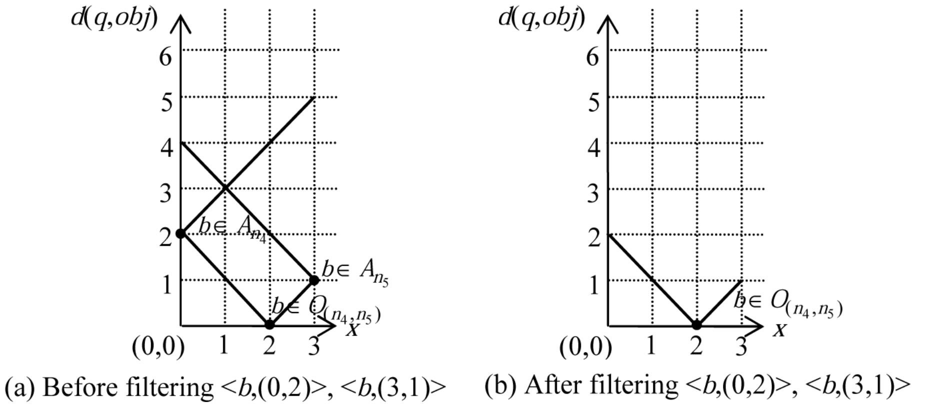


Fig. 5. Filtering redundant tuples $< b, (0, 2) >, < b, (3, 1) >$ for object b.

$< b, 2 >\}$, $A_{n_5} = \{< obj, d(n_5, obj) > |d(obj, n_5) \leqslant 3\} = \{< b, 1 >, < c, 3 >\}$, and $O_{(n4,n5)} = \{< obj, d(n_4, obj) > |obj \in SQ_{(n_4,n_5)}\} = \{< b, 2 >\}$.

Figure 4 depicts the change of the network distance between query point $y_2 \geqslant |x_2 - x_1| + y_1$ and each of objects a, b, c while q moves on $SQ_{(n_4,n_5)}$. In these graphs, the x and y values indicate $d(q, n_4)$ and $d(q, obj)$, respectively, where $obj \in A_{n_4} \cup A_{n_5} \cup O_{(n_4,n_5)}$. For instance, $x = 2$ means that q reaches object b. In Figs 4 to 7, each bold dot (e.g., (0,2), (3,3), (3,1), (2,0) in Fig. 4) represents an apex point of a piecewise-linear graph. At the apex point, the distance between q and an object is smallest. In other words, as q is far from the apex point of an object, the distance of q to the object increases. For simplicity, we consider the edges to be bidirectional, but our methods can be easily applied to networks with unidirectional edges (e.g., one way roads or roads where the weight is not same for different directions).

From Fig. 4, one can see that while q moves on a sequence $SQ_{(n_s,n_{s+1},\ldots n_e)}$, the distance between q and each object in A_{n_s}, A_{n_e}, or $O_{(n_s,n_{s+1},\ldots,n_e)}$ is represented by a piecewise-linear graph. Therefore, it is necessary to determine the coordinate of the apex point of the piecewise-linear graph. The coordinate (x_{apex}, y_{apex}) of the apex point can be automatically determined according to the conditions in Table 2. Let R_{μ_e} be the set of $< o, (x_{apex}, y_{apex}) >$ tuples where object $o \in A_{n_s} \cup A_{n_e} \cup O_{(n_s,n_{s+1},\ldots,n_e)}$ and (x_{apex}, y_{apex}) represents the coordinate of the apex point for the object o. Then, for objects a, b, c in

Table 2
Coordinate of the apex point for object $o \in (A_{n_s} \cup A_{n_e} \cup O_{(n_s,n_{s+1},\dots n_e)})$

Condition	x_{apex}	y_{apex}
$o \in O_(n_s, n_{s+1}, \dots n_e)$	$d(n_s, o)$	0
$o \in A_{n_s}$(e.g., $a, b \in A_{n_4}$)	0	$d(n_s, o)$
$o \in A_{n_e}$(e.g., $b, c \in A_{n_5}$)	sequence length	$d(n_e, o)$

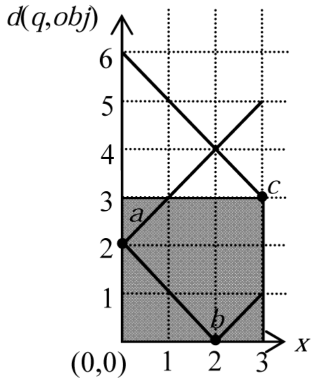


Fig. 6. Constrained query region and qualifying objects a, b, c.

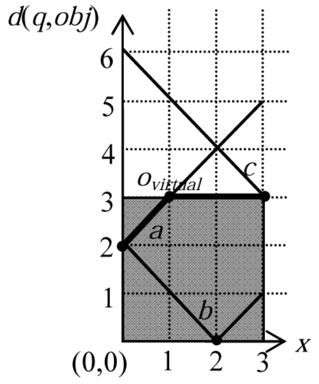


Fig. 7. Marginal objects a, $o_{virtual}$, c for [0,1], (1,3), [3,3], respectively.

A_{n_4}, A_{n_5}, or $O_{(n_4,n_5)}$, $\Phi = \{< a, (0, 2) >, < b, (0, 2) >, < b, (3, 1) >, < c, (3, 3) >, < b, (2, 0) >\}$.

Observe that object b appears three times in Φ. Two redundant tuples <b,(0,2)> and <b,(3,1)> should be removed as shown in Fig. 5 since <b,(0,2)>, <b,(3,1)>, <b,(2,0)> represent different network distances between q and b and the smallest distance becomes the length of the shortest path between them. After removing the two redundant tuples, $\Phi = \{< a, (0, 2) >, < b, (2, 0) >, < c, (3, 3) >\}$. When R_{μ_e} includes multiple tuples for the same object, these tuples should be carefully handled using the *cover* relationship [6]. For instance, consider that $\Phi = \{< o_{same}, (0, 5) >, < o_{same}, (3, 5) >\}$ has two tuples for the same object o_{same} and sequence length is 3. In this case, none of the two tuples is removed.

6.2. Decision of valid intervals and their query answer

Figure 6 shows the relationship between qualifying objects and the constrained query region. In this figure, the gray rectangle represents the constrained query region. The query distance and sequence length correspond to the height and width of the rectangle, respectively.

We now elaborate on how to decide the valid intervals and query answer for each valid interval. First, at the start boundary node n_s, the k-th NN o_{kth} of q is chosen. Unless o_{kth} is found due to a shortage of qualifying objects in Φ, the farthest object $o_{farthest}$ of q at n_s is chosen and is regarded as o_{kth}. A virtual object $o_{virtual}$ is introduced to represent the constrained query region and its distance to q is equal to query distance $q.r$. A marginal object is chosen between o_{kth} and $o_{virtual}$. A valid interval has the same marginal object whose distance to q is not smaller than the distance of q to every qualifying object in the valid interval. In other words, If $d(q, o_{kth}) \leqslant d(q, o_{virtual})$, o_{kth} becomes the marginal object. Otherwise, $o_{virtual}$ becomes the marginal object. To determine a split point where query results of contiguous valid intervals become different, we simply keep an eye on the marginal object and note when it changes.

Table 3
Experiment parameter settings

Parameter	Range
Number of objects (N_{obj})	0.5 k $\sim$ 8 k
Distribution of objects (D_{obj})	Skewed/uniform
Number of moving queries (N_{qry})	1 k $\sim$ 100 k
Number of requested nearest neighbors (k)	8 $\sim$ 64
Query distance (r)	1 km $\sim$ 8 km
Maximum observation distance (R_{max})	1 km $\sim$ 5 km
Query speed (Q_{sp})	60 km/hr
Server cache size/Client cache size	128MB/32 KB

Figure 7 identifies marginal objects among objects a, b, c, and $o_{virtual}$. For the interval [0,1], object a becomes the marginal object since object a is o_{kth} for the interval and $d(q, a) \leqslant d(q, o_{virtual})$. However, for interval (1,2], $o_{virtual}$ becomes the marginal object since object a is o_{kth} for interval (1,2] but $d(q, o_{virtual}) < d(q, a)$. Similarly, for interval (2,3), $o_{virtual}$ becomes the marginal object since $d(q, o_{virtual}) < d(q, c)$. Finally, for interval [3,3], object c becomes the marginal object. The bold line segments represent valid intervals [0,1], (1,3), and [3,3] of the marginal objects a, $o_{virtual}$, and c, respectively.

To produce query result for a valid interval, it is sufficient to find qualifying objects o such that for each object $o \in \Phi$, $d(q, o) \leqslant d(q, o_{marginal})$ where $o_{marginal}$ denotes the marginal object for the interval. Consequently, the query answer $q.A$ for $SQ_{(n_4,n_5)}$ is given as follows: $q.A = \{(I, A_I) | ([0,1], \{a, b\}), ((1,3), \{b\}), ([3,3], \{b, c\})\}$ where the first attribute I is a valid interval and the second attribute A_I is the query answer for the valid interval I.

It is worth noting that the CkNN query can be transformed to either a range query or k-NN query depending on the query conditions. In an extreme case, the CkNN query becomes a range query when the k value is set to infinity. In contrast, the CkNN query becomes a k-NN query when the r value is set to infinity.

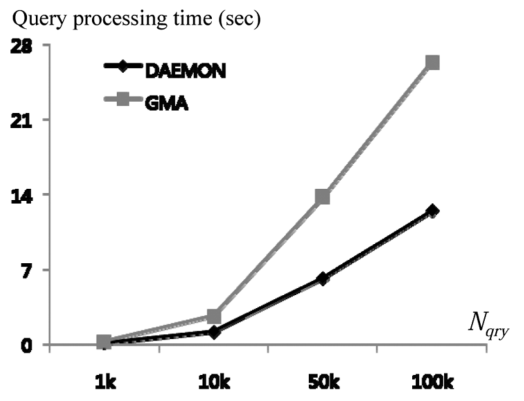
7. Performance study

In this section, we evaluate the performance of DAEMON and GMA [18] under a variety of conditions. Section 7.1 describes experimental settings and Section 7.2 presents experimental results.

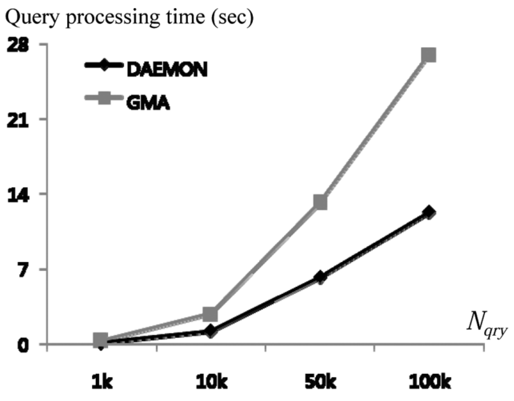
7.1. Experimental settings

The road map of Stanton County, Kansas in the United States, which contains 8,512 nodes and 9,597 road segments, is used as the road network in our experiment [30]. The road map size is approximately 1,617 km^2 ($=$ 49 km $\times$ 33 km). The number of intersection nodes is 1,736 and the number of sequences is 3,153.

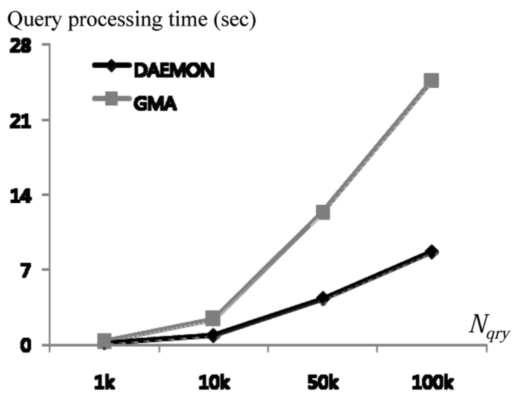
Each workload consists of from 1 k to 100 k CkNN queries which correspond to mobile clients moving at 60 km/hr for 10 minutes. We simulate moving clients (mobile queries) by using the spatio-temporal data generator described in [2]. In GMA, mobile clients inform the server of their changed locations every second. The initial locations of the queries are randomly distributed around the road network and their next road segments are selected with even probability. The distribution of the objects is either uniform or skewed and the cardinality of the objects varies between 0.5 k and 8 k. In other words, when the object cardinalities are 0.5 k and 8 k, average object densities become 0.3/km^2 and 4.9/km^2,



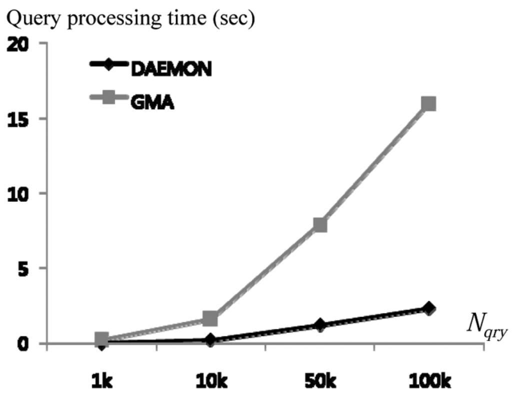
(a) N_{obj}=0.5k, D_{obj}=skewed



(b) N_{obj}=0.5k, D_{obj}=uniform



(c) N_{obj}=8k, D_{obj}=skewed



(d) N_{obj}=8k, D_{obj}=uniform

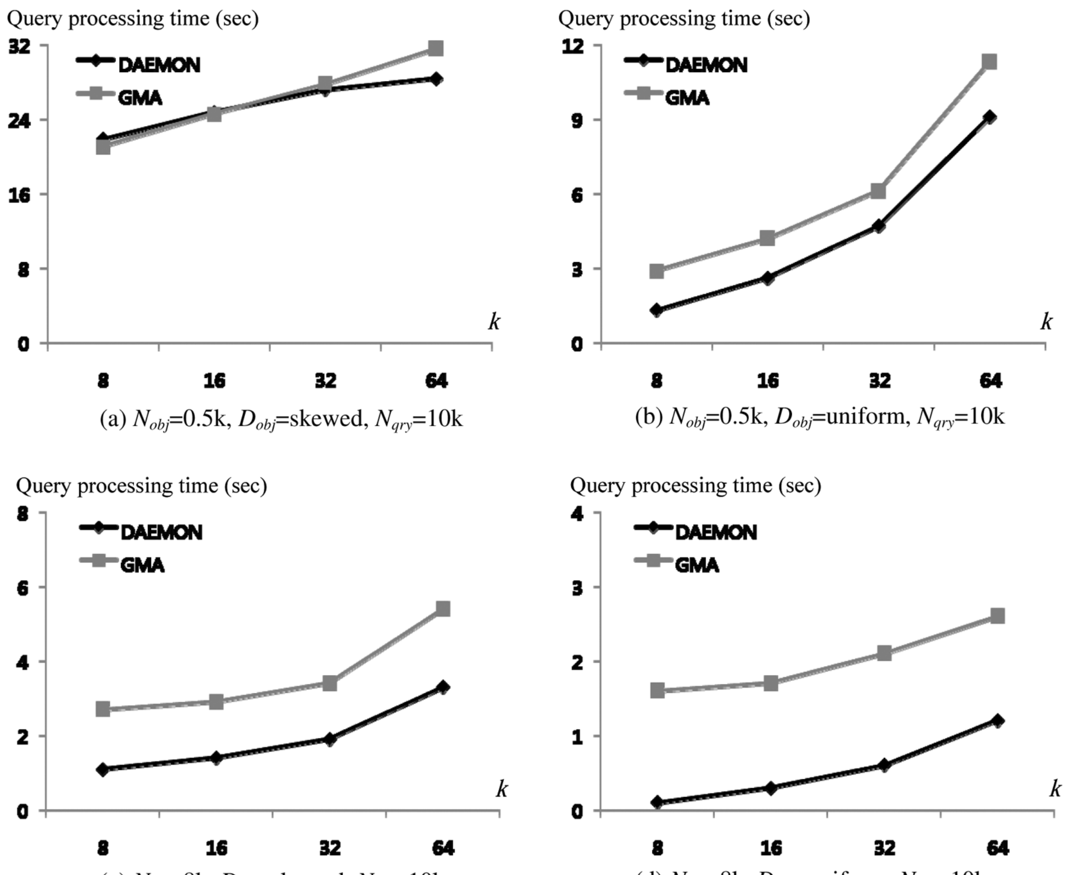
Fig. 8. DAEMON vs. GMA in terms of query processing time (k = [8,16], r = [3 km, 7 km]).

respectively. The server cache size is set to 128 MB and the client cache size is set to 32 KB. The former is used to maintain information on condensing nodes while the latter is used to cache server responses for current and previous active sequences. Table 3 summarizes the parameters used in the experiment. All the experiments are conducted on a Pentium 2.8 GHz CPU with 4 GBytes of memory.

We investigate the performance of DAEMON and GMA with two major measures: (1) query processing time and (2) messaging cost. The query processing time is the average of the sum of query processing times at the server and clients while the clients travel along their active sequence. The messaging cost is measured by the size and the number of messages transferred between the clients and server while the clients travel along their active sequence.

7.2. Experimental results

Figure 8 shows a performance comparison of DAEMON and GMA using query workloads where the k value of each query is randomly chosen between 8 and 16 and the r value is randomly chosen between 3 km and 7 km. The result trend is so similar that we only show a small selection of these results. On average, the performance of DAEMON is 3.2 times better than that of GMA. The performance gap increases with the N_{qry} value. This is expected because in GMA, the frequent location updates of clients



(a) N_{obj}=0.5k, D_{obj}=skewed, N_{qry}=10k

(b) N_{obj}=0.5k, D_{obj}=uniform, N_{qry}=10k

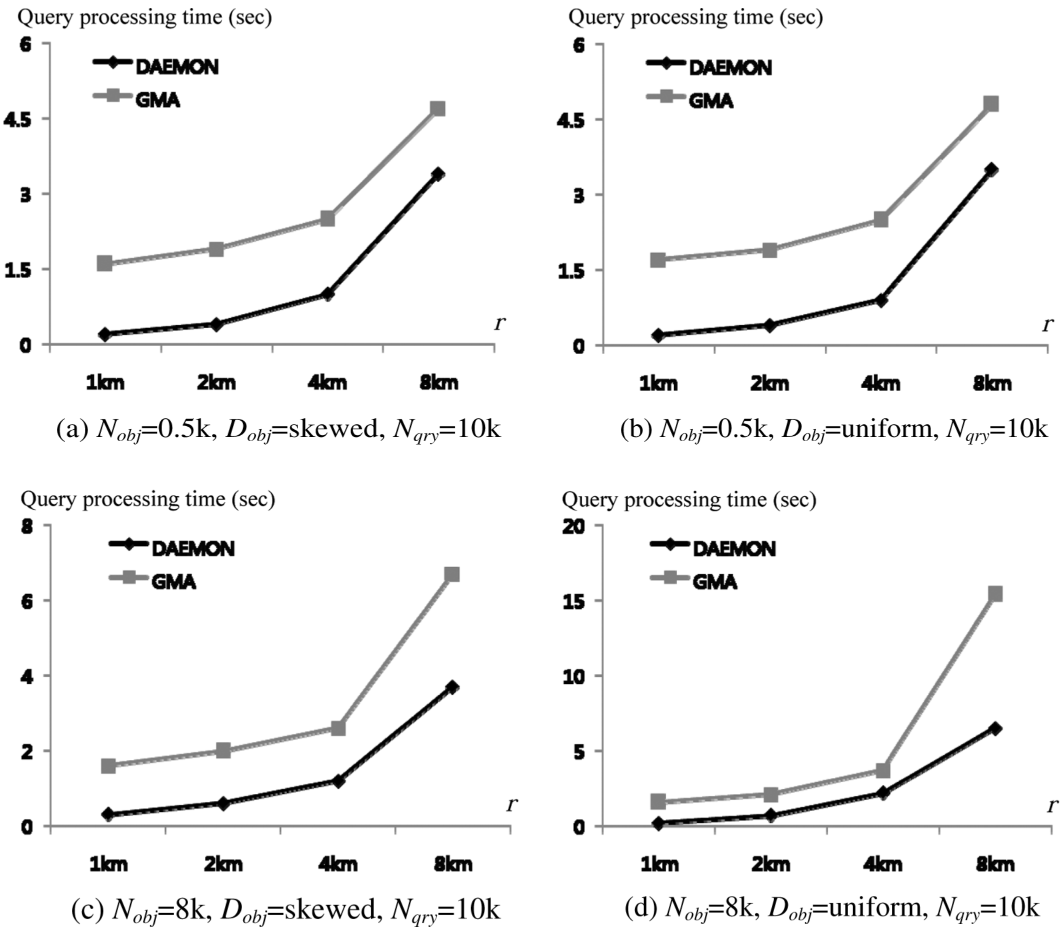
(c) N_{obj}=8k, D_{obj}=skewed, N_{qry}=10k

(d) N_{obj}=8k, D_{obj}=uniform, N_{qry}=10k

Fig. 9. DAEMON vs. GMA in terms of query processing time ($r = \infty$, $k =$ [8,64]).

incur the high wireless communication cost and trigger query reevaluation. In GMA, the client sends location update messages periodically (e.g., every second) to the server. The server re-evaluates the CkNN query whenever the movement of the client occurs. The server should provide the updated result to the client if the movement invalidates current query result. Such a situation incurs high computational costs and messaging costs particularly when N_{obj} (e.g., $N_{obj} = 8$ k) is large and N_{qry} (e.g., $N_{qry} = 100$ k) is also large as seen in Figs 8(c) and 8(d).

In DAEMON, the clients do not report their locations and simply ask for the query results at boundary nodes and for objects on the sequence. The query processing is distributed among the clients and the server. Therefore, in DAEMON, large N_{qry} values do not cause sharp performance degradation. It is interesting to observe that Figs 8(a) and 8(b) show very similar results even if the distributions (i.e., uniform and skewed distributions) are quite different. The reason for this is that the query region is limited by query distance r and the object cardinality is very low (in this case $N_{obj} = 500$) so that there is little difference in the search space despite the difference in the distribution of objects. In summary, DAEMON also shows an increase in query processing time due to the increased server load as the N_{qry} value increases, but its performance improves relative to GMA during this increase in load. In both DAEMON and GMA, the processing time at the server accounts for the majority of the total query processing time.



(a) N_{obj}=0.5k, D_{obj}=skewed, N_{qry}=10k

(b) N_{obj}=0.5k, D_{obj}=uniform, N_{qry}=10k

(c) N_{obj}=8k, D_{obj}=skewed, N_{qry}=10k

(d) N_{obj}=8k, D_{obj}=uniform, N_{qry}=10k

Fig. 10. DAEMON vs. GMA in terms of query processing time ($k = \infty$, r = [1 km, 8 km]).

Figure 9 shows a performance comparison of DAEMON and GMA using workloads where the k value of each query varies from 8 to 64 and the r value is set to infinity. These workloads simulate continuous NN queries. The query performance of DAEMON is greatly affected by the density and distribution of objects since the r value is set to infinity. As seen in Fig. 9(a), when a small number of objects are distributed nonuniformly, the search space is so large that the condensing nodes are of little use. On average, the performance of DAEMON is 2.6 times better than that of GMA. The performance gap between DAEMON and GMA typically increases with the N_{obj} value. The number of valid intervals in an active sequence increases with the N_{obj} value. When the object cardinality is high as seen in Figs 9(c) and 9(d), the server of GMA has to evaluate queries whenever the movements of the clients are reported to the server. If necessary, updated results are sent to the clients. However, in DAEMON, each client sends only one request to the server per active sequence and the server evaluates the query only once for every client.

Figure 10 shows a performance comparison between DAEMON and GMA using workloads where the r value of each query varies from 1 km to 8 km and the k value is set to infinity. These workloads simulate continuous range queries. The search space is limited by the query distance r independent of the density and distribution of objects. As a result, DAEMON substantially outperforms GMA due to

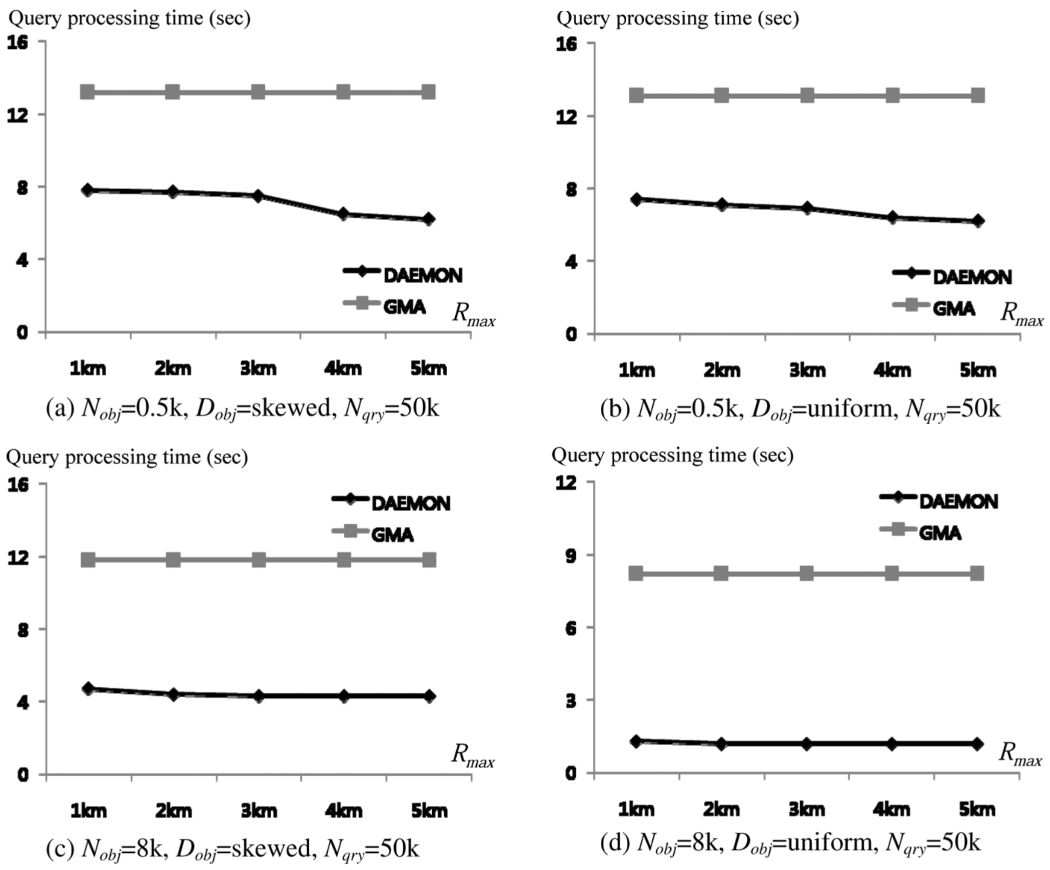


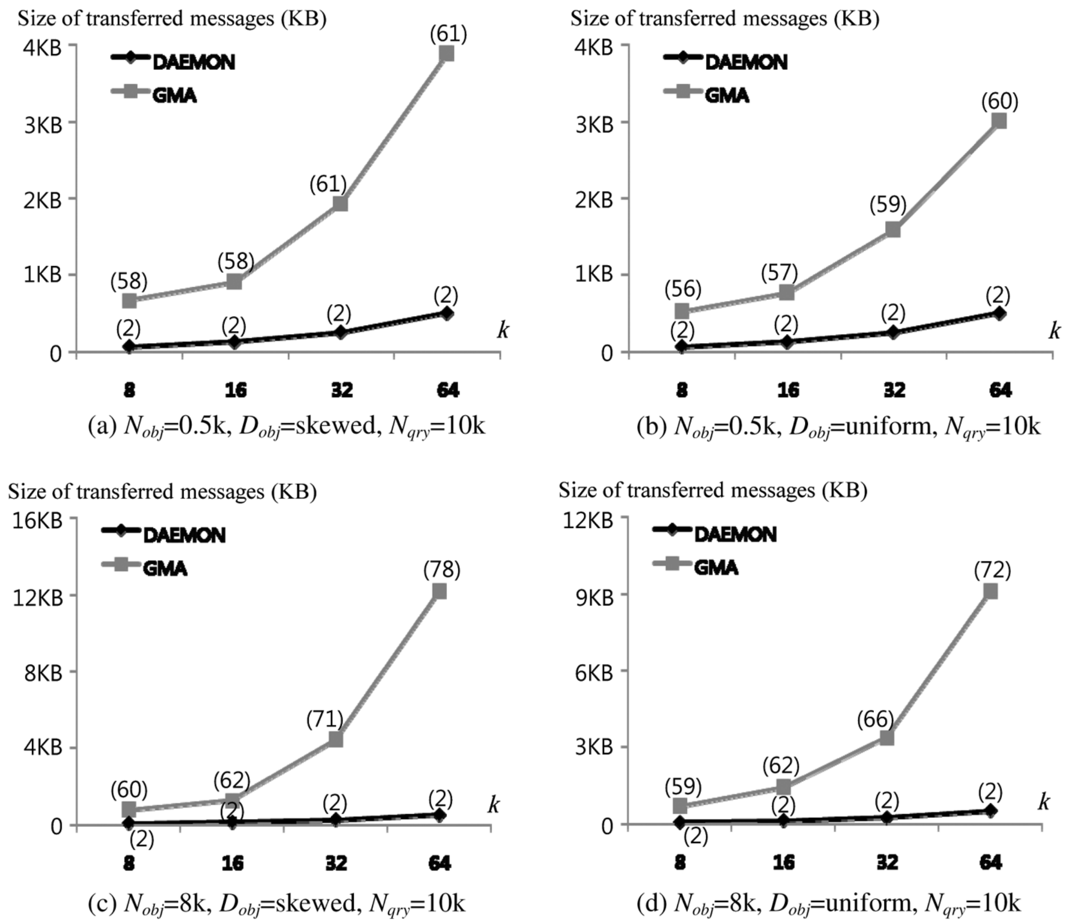
Fig. 11. Query processing time with respect to R_{max} (k = [8,16], r = [3 km, 7 km]).

the use of condensing nodes. On average, the performance of DAEMON is 4.3 times better than that of GMA.

Figure 11 shows the effect of maximum observation distance $R_{\max}$ of condensing nodes on the performance of DAEMON where the $R_{\max}$ value varies from 1 km to 5 km. As the maximum observation distance is larger, DAEMON shows better performance. The reason is that the observation distance of condensing nodes increases with the $R_{\max}$ value. When the cardinality of objects is low, a larger $R_{\max}$ value achieves better performance as seen in Figs 11(a) and 11(b). In Figs 11(c) and 11(d), since the cardinality of objects is high, the change in the $R_{\max}$ value rarely affects the query cost. DAEMON has a trade-off between its query cost and update cost both of which are controlled by the $R_{\max}$ value. As the $R_{\max}$ value increases, the query cost of DAEMON decreases while the update cost increases. To minimize the query cost for a given query distance r, it is sufficient for $R_{\max} \geqslant r$.

Figure 12 shows a comparison of messaging costs between DAEMON and GMA using workloads where the k value of each query varies between 8 and 64 and the r value is fixed to 5 km. The numbers in parentheses of these figures refer to the total number of messages transferred between the client and server while the client stays on the active sequence. It is important to note that DAEMON always requires two messages which are the client request and server response messages. In terms of messaging cost, DAEMON achieves a close-to-optimal communication cost.

In GMA, the client reports its location periodically (e.g., every second) and the server evaluates the



(a) N_{obj}=0.5k, D_{obj}=skewed, N_{qry}=10k (b) N_{obj}=0.5k, D_{obj}=uniform, N_{qry}=10k

(c) N_{obj}=8k, D_{obj}=skewed, N_{qry}=10k (d) N_{obj}=8k, D_{obj}=uniform, N_{qry}=10k

Fig. 12. DAEMON vs. GMA in terms of messaging cost (r = 5 km, k = [8,64]).

query based on the last updated location of the client and informs the client of the changed result if necessary. Specifically, in GMA, when the k values are set to 8, 16, 32, and 64, the ratio of the number of location update messages to the total number of messages are on average 95%, 92%, 87%, and 81%, respectively. For larger k values, the server provides more updated results to the clients. This is because the number of valid intervals in the active sequence increases with the k value. In our road map, the average length of a sequence is about 0.93 km. For example, if a client moves at 60 km/hr, it takes about 55.8 seconds for the client to go through the active sequence. This means that a GMA client sends an average of 55 location update messages to the server while on a particular active sequence.

Figure 13 shows the difference of the size of transferred messages in DAEMON when the client cache is turned on and off. The size of transferred messages is explored under the same conditions as Fig. 8. When the client cache is turned on and the server response for the previous active sequence is cached, the size of transferred messages is about 1.76 times smaller than when the client cache is turned off. This is expected because when using the client cache, the client typically asks for qualifying objects at a boundary node of the active sequence rather than those at both boundary nodes. It is interesting to observe that despite the difference in the cardinality of objects, there is little difference in the size of transferred messages between when N_{obj} = 8 k and D_{obj} = skewed and when N_{obj} = 0.5 k and D_{obj} = uniform. When N_{obj} = 8 k and D_{obj} = skewed, in many cases, the server response includes no

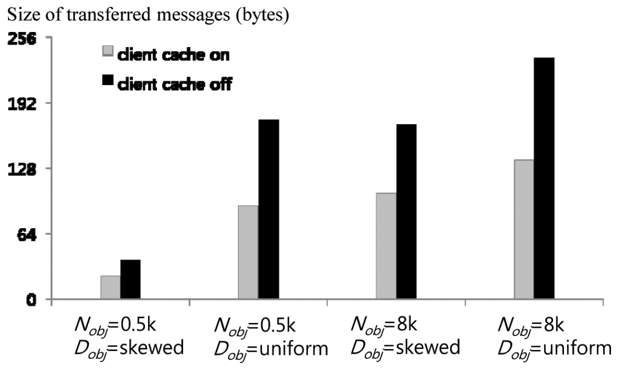


Fig. 13. Client cache on vs. client cache off (k = [8,16], r = [3 km, 7 km]).

qualifying objects or a smaller number of qualifying objects than the requested size due to the skewed distribution of objects.

8. Conclusions

We propose a distributed and scalable strategy, DAEMON, for the continuous monitoring of CkNN queries on road networks. In DAEMON, the server efficiently evaluates CkNN queries issued at boundary nodes and quickly retrieves the objects on the sequence using dedicated data structures: the condensing nodes and sequences. Each client sends only one request to the server while traveling on consecutive road segments between adjacent intersections and then generates query result for these road segments using the server response. Unlike the centralized solutions, in DAEMON, the clients do not send location update messages to the server, which reduces the communication cost and alleviates concerns of location privacy. Our distributed processing of CkNN queries significantly decreases the computation cost of the server and the communication cost between clients and the server while placing only a small processing burden on the mobile clients. Our extensive experimental results confirm that DAEMON clearly outperforms its rival in terms of query processing time while achieving close-to-optimal communication costs.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0013487).

References

[1] J. Bao, C. Chow, M. Mokbel and W. Ku, Efficient evaluation of k-Range nearest neighbor queries in road networks, *Mobile Data Management* (2010), 115–124.
[2] T. Brinkhoff, A framework for generating network-based moving objects, *GeoInformatica* **6**(2) (2002), 153–180.
[3] Y. Cai, K. Hua and G. Cao, Processing range-monitoring queries on heterogeneous mobile objects, *Mobile Data Management* (2004), 27–38.
[4] M. Cheema, L. Brankovic, X. Lin, W. Zhang and W. Wang, Continuous monitoring of distance-based range queries, *IEEE Trans Knowl Data Eng* **23**(8) (2011), 1182–1199.

[5] Z. Chen, H. Shen, X. Zhou and J. Yu, Monitoring path nearest neighbor in road networks, *SIGMOD Conference* (2009), 591–602.
[6] H. Cho and C. Chung, An efficient scalable approach to CNN Queries in a road network, *VLDB* (2005), 865–876.
[7] T. Delot, S. Ilarri, N. Cenerario and T. Hien, Event sharing in vehicular networks using geographic vectors and maps, *Mobile Information Systems* **7**(1) (2011), 21–44.
[8] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal and A. Abbadi, Constrained nearest neighbor queries, *SSTD* (2001), 257–278.
[9] Y. Gao, G. Chen, Q. Li, C. Li and C. Chen, Constrained k-Nearest neighbor query processing over moving object trajectories, *DASFAA* (2008), 635–643.
[10] B. Gedik and L. Liu, MobiEyes: A Distributed location monitoring service using moving location queries, *IEEE Trans Mob Comput* **5**(10) (2006), 1384–1402.
[11] N. Ghadiri, A. Dastjerdi, N. Aghaee and M. Nematbakhsh, Optimizing the performance and robustness of type-2 fuzzy group nearest-neighbor queries, *Mobile Information Systems* **7**(2) (2011), 123–145.
[12] M. Hasan, M. Cheema, W. Qu and X. Lin, Efficient algorithms to monitor continuous constrained k nearest neighbor queries, *DASFAA* (1) (2010), 233–249.
[13] H. Hu, J. Xu and D. Lee, PAM: An efficient privacy-aware monitoring framework for continuously moving objects, *IEEE Trans Knowl Data Eng* **22**(3) (2010), 404–419.
[14] S. Ilarri, E. Mena and A. Illarramendi, Location-dependent query processing: Where we are where we are heading, *ACM Comput Surv* **42**(3) (2010).
[15] M. Mokbel, X. Xiong and W. Aref, SINA: Scalable incremental processing of continuous queries in spatio-temporal databases, *SIGMOD Conference* (2004), 623–634.
[16] D. Moon, B. Park, Y. Chung and J. Park, Recovery of flash memories for reliable mobile storages, *Mobile Information Systems* **6**(2) (2010), 177–191.
[17] F. Morvan and A. Hameurlain, A mobile relational algebra, *Mobile Information Systems* **7**(1) (2011), 1–20.
[18] K. Mouratidis, M. Yiu, D. Papadias and N. Mamoulis, Continuous nearest neighbor monitoring in road networks, *VLDB* (2006), 43–54.
[19] K. Mouratidis and M. Yiu, Anonymous query processing in road networks, *IEEE Trans Knowl Data Eng* **22**(1) (2010), 2–15.
[20] D. Papadias, J. Zhang, N. Mamoulis and Y. Tao, Query processing in spatial network databases, *VLDB* (2003), 802–813.
[21] D. Stojanovic, A. Papadopoulos, B. Predic, S. Kajan and A. Nanopoulos, Continuous range monitoring of mobile objects in road networks, *Data Knowl Eng* **64**(1) (2008), 77–100.
[22] W. Wu, W. Guo and K. Tan, Distributed processing of moving K-Nearest-Neighbor query on moving objects, *ICDE* (2007), 1116–1125.
[23] K. Xuan, G. Zhao, D. Taniar, J. Rahayu, M. Safar and B. Srinivasan, Voronoi-based range and continuous range query processing in mobile databases, *J Comput Syst Sci* **77**(4) (2011), 637–651.
[24] K. Xuan, G. Zhao, D. Taniar, M. Safar and B. Srinivasan, Constrained range search query processing on road networks. Concurrency and Computation, *Practice and Experience* **23**(5) (2011), 491–504.
[25] K. Xuan, G. Zhao, D. Taniar, M. Safar and B. Srinivasan, Voronoi-based multi-level range search in mobile navigation, *Multimedia Tools Appl* **53**(2) (2011), 459–479.
[26] K. Xuan, G. Zhao, D. Taniar and B. Srinivasan, Continuous Range search query processing in mobile navigation, *ICPADS* (2008), 361–368.
[27] G. Zhao, K. Xuan, W. Rahayu, D. Taniar, M. Safar, M. Gavrilova and B. Srinivasan, Voronoi-based continuous k nearest neighbor search in mobile navigation, *IEEE Transactions on Industrial Electronics* **58**(6) (2011), 2247–2257.
[28] G. Zhao, K. Xuan and D. Taniar, Path kNN query processing in Mobile Systems, *IEEE Transactions on Industrial Electronics*, to appear (DOI: 10.1109/TIE.2011.2167113).
[29] G. Zhao, K. Xuan, D. Taniar and B. Srinivasan, LookAhead continuous KNN mobile query processing, *Comput Syst Sci Eng* **25**(3) (2010), 205–217.
[30] U.S. Census Bureau – TIGER/Line http://www.census.gov/geo/www/tiger/.

Hyung-Ju Cho received the B.S. and M.S. degrees in Computer Engineering from Seoul National University in February 1997 and in February 1999, respectively, and the Ph.D. degree in Computer Science from KAIST, in August 2005. He is currently a research assistant professor at the Dept. of Computer Engineering, Ajou University, South Korea. His current research interests include moving objects databases and query processing in mobile peer-to-peer networks.

Seung-Kwon Choe got his BS degree in Computer Engineering at Ajou University, South Korea in February 2010. Currently, he pursues the MS degree in Computer Engineering at Ajou University. His research area includes flash memory storages and moving objects databases.

Tae-Sun Chung received the B.S. degree in Computer Science from KAIST, in February 1995, and the M.S. and Ph.D. degree in Computer Science from Seoul National University, in February 1997 and August 2002, respectively. He is currently an associate professor at School of Information and Computer Engineering at Ajou University. His current research interests include flash memory storages, XML databases, and database systems.