Distributed Continuous Range Query Processing on Moving Objects*

Haojun Wang, Roger Zimmermann, and Wei-Shinn Ku

University of Southern California, Computer Science Department,
Los Angeles, CA, USA
{haojunwa, rzimmerm, wku}@usc.edu

Abstract. Recent work on continuous queries has focused on processing queries in very large, mobile environments. In this paper, we propose a system leveraging the computing capacities of mobile devices for continuous range query processing. In our design, continuous range queries are mainly processed on the mobile device side, which is able to achieve real-time updates with minimum server load. Our work distinguish itself from previous work with several important contributions. First, we introduce a distributed server infrastructure to partition the entire service region into a set of service zones and cooperatively handle requests of continuous range queries. This feature improves the robustness and flexibility of the system by adapting to a time-varying set of servers. Second, we propose a novel query indexing structure, which records the difference of the query distribution on a grid model. This approach significantly reduce the size and complexity of the index so that in-memory indexing can be achieved on mobile objects with constrained memory size. We report on the rigorous evaluation of our design, which shows substantial improvement in the efficiency of continuous range query processing in mobile environments.

1 Introduction

With the growing popularity of GPS-enabled mobile devices and the advances in wireless technology, the efficient processing of *continuous range queries*, which is defined as retrieving the information of moving objects inside a user-defined region and continuously monitoring the change of query results in this region over a certain time period, has been of increasing interest. Continuous range query processing is very important due to its broad application base. For instance, the Department of Transportation may want to monitor the traffic change on a freeway section to develop a traffic control plan. In a natural disaster, it is highly desirable to locate all fire engines within a certain area for emergency response. Continuous range queries pose new challenges to the research community because the movement of objects causes the query results to change correspondingly. Applying a central server processing solution where moving objects periodically update their locations is obviously not scalable. On the other hand, the growing computing capabilities of mobile devices has enabled approaches such as

* This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), CMS-0219463 (ITR), IIS-0534761 and equipment gifts from the Intel Corporation, Hewlett-Packard, Sun Microsystems and Raptor Networks Technology.

S. Bressan, J. Küng, and R. Wagner (Eds.): DEXA 2006, LNCS 4080, pp. 655–665, 2006.

$MobiEyes$ [2] and MQM [1] that use mobile devices to answer continuous range queries, where a centralized server acts as a mediator. However, these solutions suffer from some limitations. First, a centralized server is not robust enough under certain situations. In the mentioned example of natural disasters, some servers might be down or only provide limited computational capacity. Therefore, it is highly desirable to have a fault resilient infrastructure. Second, the communication between the server and moving objects should be minimized in order to manage data in large mobile environments. Finally, the memory and computing capabilities of mobile devices are limited so that the implementation of in-memory processing on moving objects needs to be carefully considered.

In this paper, we address the problem of processing real-time continuous range queries by proposing a robust and scalable infrastructure. The goal is to build a system that supports a large number of moving objects with limited server and communication resources. In our design, continuous range queries are mainly processed by mobile devices. Our work distinguishes itself from previous work with two contributions. First, we propose a distributed server infrastructure. We introduce the feature of *service zones*. A service zone is a subspace being recursively binary partitioned from the entire service region. Each server controls a service zone. Our system is able to adaptively allocate and merge service zones as servers join or leave. In addition, we propose a novel *grid index* on continuous range queries. Instead of recording the distribution of queries, our design of grid index preserves the change of the query distribution and is more compact than other grid index structures. Our experimental results show that our design is very efficient to support numerous continuous range queries with a very large number of moving objects under the mobile environments.

The rest of this paper is organized as follows. The related work is described in Section 2. In Section 3 we introduce the design of service zones, grid index, and the support of continuous range query processing. The experimental validation of our design is presented in Section 4. Finally, we discuss the conclusions and future work in Section 5.

2 Related Work

A number of studies have addressed continuous spatial queries. Related work, such as presented in [10], [11], and [14], addressed the processing of continuous spatial queries on the server. For the efficient processing of a large number of continuous queries at the same time, Prabhakar et al. [7] addressed the issue of stationary continuous queries in a centralized environment. In addition, Mokbel et al. [5] proposed SINA that supports moving queries over moving objects in server-based processing. By contrast, MQM [1], and MobiEyes [2] assume a distributed environment, where the mobile hosts have the computing capability to process continuous queries. A centralized server is introduced by both approaches to work as a mediator coordinating the query processing. In MQM, the concept of *resident domain* is introduced as a subspace surrounding the moving object to process continuous queries. Continuous queries are partitioned into a set of *monitor regions*, where only the monitor regions covered by the resident domain will be sent to the moving object. However, partitioning continuous queries is inefficient because it increases the number of queries in the system.

On the issue of moving object indexing, the *TPR-Tree* [9] and its variants have been proposed to index moving objects with trajectories. However, the support of continuous queries by these methods is very inefficient. Kalashnikov et al. [4] evaluated the efficiency of indexing moving objects and concluded that using a grid approach for query indexing results in the best performance. Other methods to process continuous queries without a specific index can be found such as the usage of *validity regions* [11], *safe regions* [3], *safe periods* [5], and *No-Action regions* [13]. These approaches have in common that they return a valid time or region of the answer. Once the result becomes invalid, the client submits the query for reevaluation.

Our work distinguishes itself from the above approaches, by specifically addressing the scalability and robustness of the system. We adaptively organize servers to cooperatively work in the entire service space. Furthermore, we propose a grid index that is able to be implemented as an in-memory data structure on mobile devices. There is no restriction on the movement of objects and the system is extremely efficient to support continuous range queries with a very large number of moving objects.

3 System Design and Components

3.1 System Infrastructure and Assumptions

Figure 1 illustrates the system infrastructure of our design. We are considering mobile hosts with abundant power capacity, such as vehicles, that are equipped with a Global Positioning System (GPS) for obtaining continuous position information. We assume that the mobile host has some memory and computing capacity to store the queries and process range query operations. In our paper, we use the term moving objects to refer to these mobile hosts participated in the query processing. On the base-station side, our design has two assumptions. First, the servers and moving objects communicate via cellular-based wireless network. Moreover, protocols such as $GeoCast$ [6] can be adopted for sending messages within a certain region. Second, the servers with spatial databases are connected via the wired Internet infrastructure. Each server is able to receive query requests from any user and forward them to the appropriate servers.

In our design, moving objects are represented as points and range queries are denoted as rectangular regions. Given a set of moving objects and continuous range queries, the challenge is to calculate which objects lie within which query regions at a certain time.

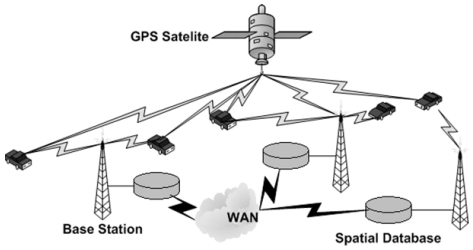


Fig. 1. The system infrastructure

In this paper, we focus on range queries, which are widely used in spatial applications and can be used as preprocessing tools for other queries, such as nearest neighbor queries. For simplicity, we use the term queries to refer to continuous range queries in the following sections.

3.2 Server Design

In this section, we describe our design of the server infrastructure. First, we describe how the system adaptively manages the service region by adapting to a time-varying set of servers through the concept of the service zone. Next, we present another important feature, the grid index. By using the grid index, our system avoids excessive query retrieval from the server and significantly reduces the communication overhead.

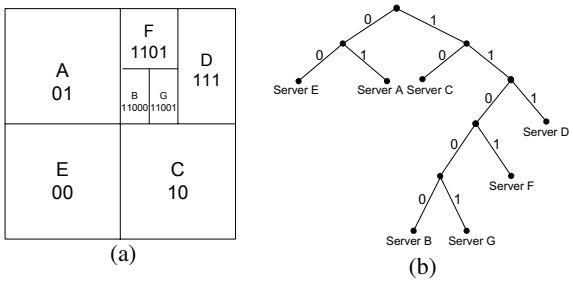


Fig. 2. An example of the system with 7 servers and their service zone identifier (SID) tree

Service Zones. We leverages the design of Content Addressable Network (CAN) [8] to dynamically partition the entire service region into a set of subspaces. Each subspace is controlled by a server. We define the term *service zone* as the subspace controlled by a server. Each service zone is addressed with a service zone identifier (SID), which is calculated from the location of the service zone. Figure 2a shows an example of the entire service region partitioned into 7 service zones. The service zone partitioning is a binary partition approach that always equally divides a larger service zone into two smaller child service zones. Hence the corresponding SID address for service zones can be represented with a binary tree structure as shown in Figure 2b. Each server maintains a routing table with tuples $\langle SID, address \rangle$ storing the routing information of its neighbor servers. By using the same routing mechanism as CAN, our system is able to allocate any service zone with complexity of $O(n\log n)$ in a system of n servers.

When a new query q is submitted, the system first forwards it to all servers covered by its query region through the M-CAN multicast algorithm from the design of CAN. When a server receives the query, it is inserted into the query repository. Consequently, the grid index on the server is updated. We will describe the details of the grid index in the next section. Finally, the server broadcasts a message $GridIndexUpdate(G_{Index})$ to all moving objects associated with it, where G_{Index} is the updated grid index.

When a query q is about to be deleted, the server searches through its repository to delete the corresponding entry. Consequently, the server updates the grid index.

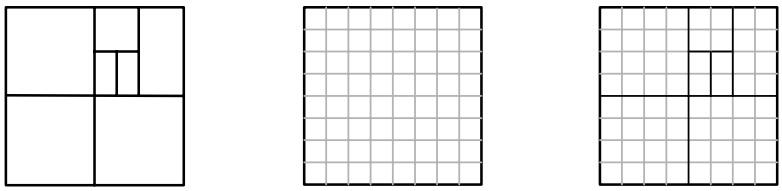
When a new server joins the system, several steps must be taken to allocate a service zone for it. First, the new server must find a bootstrap server, which is already a member of the system. Second, the bootstrap server broadcasts a message that a new server is about to join the system. Other servers in the system reply back with the information of its current system load and service zone. Our goal is the balance the system load among servers. Hence the server with the highest system load (for instance, average used disk space, average memory usage, or other user identified resources) will be performed a partition to divide the corresponding service zone into halves. Next, the bootstrap server sends a message to the partitioned server to forward queries overlapping the new server's service zone. The partitioned server also broadcasts the updated service zone information to moving objects associated with it. Moving objects register with the new server if their current locations are controlled by the new server. After the new server receives queries forwarded from the partitioned server, it creates and maintains the grid index correspondingly. Finally, the neighbors of the partitioned server will be notified to update their routing tables.

When a server leaves the system, we need to ensure that the corresponding service zone is taken over by the remaining servers. The departing server explicitly hands over its repository of moving objects and queries to one of its neighbors whose service zone can be merged with the departing servers zone to produce a valid single service zone.

Grid Index. The memory capacity of moving objects is limited. On the other hand, it is highly desirable to have an index structure that helps moving objects to retrieve queries from the server only when they are very close to the queries. Therefore, the index also needs to be compact in terms of the size to be used on moving objects. Here we present a grid index structure fulfilling these requirements.

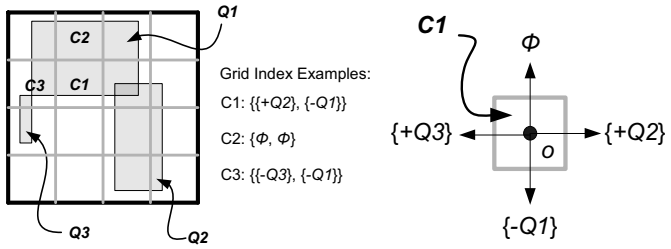
Previous work of grid-based indexing on continuous queries, such as [4] and [2], aims at random data access by recording the distribution of queries. However, objects move continuously along a trajectory in mobile environments, therefore queries in large parts of the service region can be pruned and no random access is needed. Based on this observation, our grid index preserves the difference of the query distribution that can be efficiently used for continuous query processing.

The basis of our grid index is a set of cells. Each cell is a region of space obtained by partitioning the entire service region using a uniform order. Figure 3a demonstrates a system with 7 servers. Figure 3b shows the entire service region divided into 64 grid cells. Figure 3c shows how these grid cells are distributed on the example servers. By



(a) A 7 service zone example (b) A 64 grid cell example (c) Service zones and grid cells overlapping

Fig. 3. Service zones and grid cells



(a) (b)

Fig. 4. The grid index

using a uniform grid order to partition the service region into grid cells, given the coordinates of an object, it is easy to calculate the cell in which the object resides.

The server maintains the grid index in its service zone. For each cell, the grid index structure consists of two lists identified as *right*, and *lower* that record the change of the query distribution from the right and lower neighbor cells, respectively. In the example shown in Figure 4a, a service zone is divided into 16 grid cells. Cell $C1$, $C2$, and $C3$ are partially covered by a query $Q1$. There are queries $Q2$ and $Q3$ covering the right and left neighbor cells of $C1$, repectively. As shown in Figure 4a, the grid index for cell $C1$, $C2$, and $C3$ is $\{\{+Q2\}, \{-Q1\}\}$, $\{\emptyset, \emptyset\}$, and $\{\{-Q3\}, \{-Q1\}\}$, respectively.

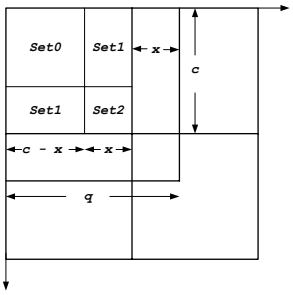


Fig. 5. Number of Grid Index Entries Analysis

Once a moving object is associated with a server, the server will forward the grid index of its service zone to the moving object. By using the grid index stored in its local memory the moving object is able to forecast the query locations with a refined granularity. As an example shown in Figure 4b, if there is a moving object o in the cell $C1$ is about to move across the right edge of $C1$, the right list of $C1$ is $\{+Q2\}$. Hence the object submits a request to retrieve the query $Q2$ from the server. If the object is about to cross the lower edge of $C1$, since the lower list of $C1$ is $\{-Q1\}$. The object could either to retain the information of query $Q1$ if there is enough memory or remove $Q1$ if more memory is needed for query processing. If the object is about to move across the upper edge of $C1$, the lower list of the upper neighbor cell will be retrieved and the values in the list will be inversed. In this example, the object retrieves the lower list of $C2$ and calculate the inverse value, which is $\emptyset$. This indicates that there is no query that needs to be retrieved from the server. When the object is about to move across the left edge of $C1$, a similar process will be performed on the right list of the left neighbor cell (i.e., $C3$). In this example, the inverse value of the list is $\{-Q3\}$. Therefore, the moving object submits a request to retrieve the query $Q3$ from the server.

To study the impact of our design on the index size, let us assume the shape of queries and grid cells are square and the length of each side of a query Q is q. Let c denote the side of each grid cell with $q > c$. Then q can be represented as $i \times c + x$, where $x \subset [0, c)$ and i is an integer. Without loss of generality let us consider the case where the top-left

corner of query q is located somewhere within the top-left grid cell of the system as shown in Fig 5. It can be verified that if the top-left corner of Q is inside $Set0$ it will result in $4(i+1)$ index entries. For $Set1$ the number of index entries is $2(i+1)+2(i+2)$, and for $Set2$ it is $4(i+2)$. Assuming uniform distribution of queries, on the average Q results in $4(q+c)/c$ index entries. On the other hand, recording the distribution of queries requires $(q+c)^2/c^2$ index entries on each Q [12]. For all $q/c \geq 3$, our approach requires less index entries than recording the distribution of queries on the grid.

3.3 Query Processing on Moving Objects

In this section, we describe the functionality of the moving objects. In our design, the following information is stored in the memory of moving objects for query processing:

- OID: the unique identifier of the moving object.
- $currentPos$: the current location of the moving object.
- G_{Index}: the grid index of the current service zone covering the moving object.
- $Queries$: the list of queries received from the server.

Table 1. Message types in query processing

Notation	Definition
$RegisterObject(OID)$	The message to register a moving object on a server.
$UnregisterObject(OID)$	The message to delete a moving object on a server.
$UpdateResult(OID, QID, Flag)$	The message to update a query result.
$RequestQueries(OID, Q_{List})$	The message to retrieve a set of queries.
$GridIndexUpdate(G_{Index})$	Updating the grid index broadcasted by the server.

In order to implement the query processing mechanism on the mobile object, a set of messages is defined as shown in Table 1.

A moving object is associated with a server at all times. When a moving object turns its power on, it broadcasts a message $RegisterObject(OID)$. The server monitoring the location of the object inserts it into the object repository and replies back with a $GridIndexUpdate(G_{Index})$ message. The server also sends the set of queries covering the current grid cell of the moving object.

When a moving object is about to leave its current service zone, it sends a message $UnregisterObject(OID)$ to the server. The server deletes the moving object from its repository and sends back a set of tuples$\langle SID, address\rangle$ from its routing table identifying adjacent service zones. The moving object sends a $RegisterObject(OID)$ message to the server controlling the zone it is entering.

When a moving object is about to move into a new grid cell, it consults the grid index as described in the previous section. If there are queries in the grid index needing to be retrieved, the moving object sends a message $RequestQueries(OID, Q_{List})$ to the server, where Q_{List} is a list of queries with query identifier QID. Once the server receives the message, it will send the corresponding queries to the moving object.

At all times, the moving object checks its current location $currentPos$ against the queries in the $Queries$ list. If the object moves into or moves out of a query, it sends

a message $UpdateResult(OID, QID, Flag)$ to the server, where QID is the query identifier and $Flag$ indicates whether the object resides in the query region.

Query processing on moving objects enables real time updates to the query result while reducing the cost of server processing substantially. We study the impact of our techniques in experiments and show that the results match our analytical expectation.

4 Experimental Evaluation

In this section we describe the experimental verification of our design. There are three metrics of interest extensively studied in our simulations. First, the *number of grid index entries* is measured as the average number of index entries generated on a server and forwarded to moving objects associated with it. This measure indicates the efficiency of our grid index design and whether the grid index can be used for in-memory processing on moving objects. Second, the *server communication cost* is measured as the average number of messages transmitted from servers to the moving objects. More specifically, the server communication cost consists of the registration messages, which are generated when a moving object enters or leaves a service zone, and the query retrieval messages, which are generated when a server receives a $RequestQueries$ message from a moving object. This metric implies whether the server may become a bottleneck in the system. Finally, the *mobile communication cost* is measured as the total number of messages transmitted from moving objects to servers. The mobile communication cost also consists of registration messages and query retrieval messages. Additionally, query update messages are generated by moving objects when they enter or leave a query region. This measure reflects the prime query processing cost and hence is important to demonstrate the scalability of our system.

4.1 Simulator Implementation

We implemented a prototype simulator that is structured into three main components: the *service zone generator*, the *object and query loader*, and the *performance monitor*.

The service zone generator creates a virtual square space with a 100km $\times$ 100km dimension. In the experiments, we partition the space into 64 service zones. Each service zone is identified by a SID representing a server.

In the next step, the object and query loader generates moving objects and imports continuous range queries into the system. We use the random walk model to simulate the movement of objects. Initially 10,000 objects are uniformly distributed in the space. Each of them moves with a constant velocity, which is randomly selected in the range from 10m to 20m per second, for a duration that is exponentially distributed with mean value equal to 100 seconds. We also generated two sets of rectangular regions as continuous range queries that are uniformly distributed in the space with an average area size of 1% and 10% of the plane size, respectively.

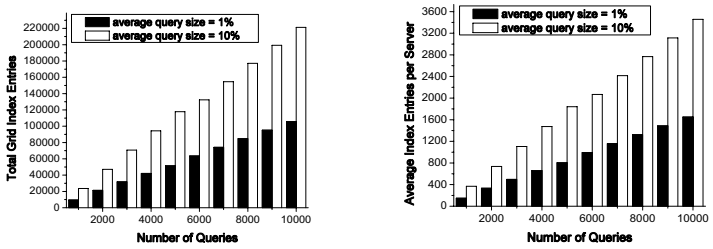
After the objects and queries are loaded into the system, the performance monitor generates the grid index for each server with 256 grid cells partitioning the entire service space. Each simulation runs for 5,000 seconds and the performance monitor reports the number of grid index entries, the server communication cost, and the mobile communication cost. Currently, our simulation is focused on the system performance in the

steady state, i.e., we do not add any more queries when the objects start to move. We plan to implement a dynamic simulation environment in the future.

4.2 Simulation Results

We were first interested in the efficiency of our grid index in terms of the size. Figure 6a plots the total number of grid index entries in the system as a function of the number of queries. The results clearly show that the total number of grid index entries increases linearly with the number of queries. Additionally, our grid index structure performs more efficiently with a larger average query size. With an average query size equal to 10% of the entire space, our grid index only doubles the number of entries compared with the case when the average query size equals 1% of the space. This behavior corroborates our analytical results described in Section 3. Furthurmore, the absolute size of the grid index is very small. If we use 16 bytes to identify a query, it only takes 3.35 MB to represent 10,000 queries with an average size of 10% of the space. Figure 6b shows the benefit of using a distributed infrastructure on the server side that further reduces the size of the grid index on each server. In the case of 10,000 queries with an average size of 10% of the space, on average the size of index entries is 54 KB on each server. This substantially reduces the requirement of memory on moving objects.

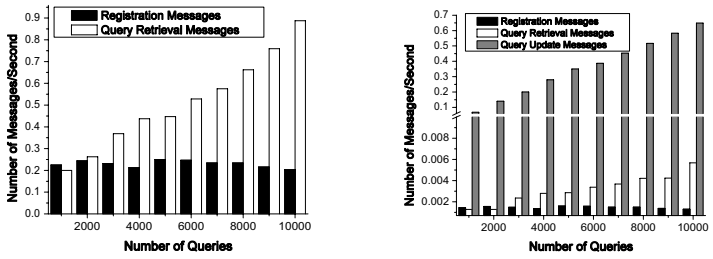
Figures 7a illustrates the average communication cost on each server with the set of queries with an average area size of 10% of the plane. As a general trend, the number



(a) The total number index entries in the system

(b) The average number of index entries on each server

Fig. 6. The number of grid index entries as a function of the number of queries



(a) The server communication cost

(b) The mobile communication cost

Fig. 7. The server and mobile communication cost as functions of the number of queries

of query retrieval messages increases with the number of queries. Intuitively, with a larger number of queries, the possibility for objects to retrieve query information from the server is larger. More importantly, the server communication cost is small in our simulation results. With 10,000 queries and 10,000 objects in the system, the server communication cost is about 1 message per second, which demonstrates that our server infrastructure is very scalable and suitable for mobile environments. Figure 7b demonstrates the mobile communication cost with respect to the number of queries. It shows that the query update messages are the primary cost of mobile communication cost. However, with 10,000 queries, the object query update message count on each object is about 0.7 per second. Assuming the size of query update message is 32 byte, the average message size transmitted from each object is about 22 bytes/second. Therefore, our design on the mobile object side is very scalable.

5 Conclusions and Future Directions

Continuous range queries have generated intense interest in the research community because the advances in GPS devices is enabling new applications. We have presented a novel system that utilizes the computing capability of moving objects for continuous range query processing. Our design of service zones and a grid index is able to provide accurate real time query results for a very large number of moving objects and queries.

In the future, we intend to study the communication costs so that the size of the grid can be optimized with regard to the query distribution. Moreover, a dynamic grid index retrieval from the server with respect to the memory capacity on moving objects is worth exploring.

References

1. Y. Cai, K. Hua, and G. Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *International Conference on Mobile Data Management*, pages 27–38, 2004.
2. B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *International Conference on Extending Database Technology*, pages 67–87, 2004.
3. H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD*, pages 479–490, 2005.
4. D. V. Kalashnikov, S. Prabhakar, S. E. Hambrusch, and W. G. Aref. Efficient Evaluation of Continuous Range Queries on Moving Objects. In *International Conference on Database and Expert Systems Applications*, pages 731–740, 2002.
5. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, pages 479–490, 2004.
6. J. C. Navas and T. Imielinski. GeoCast - Geographic Addressing and Routing. In *International Conference on Mobile Computing and Networking*, pages 66–76, 1997.
7. S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. In *IEEE Transaction on Computers*, 2002.
8. S. Ratnasamy. A Scalable Content-Addressable Network. In *Ph.D. Dissertation University of California Berkeley*, 2002.

9. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, 2000.
10. Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD*, pages 79–96, 2001.
11. Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, pages 287–298, 2002.
12. H. Wang, R. Zimmermann, and W.-S. Ku. ASPEN: An Adaptive Spatial Peer-to-Peer Network. In *ACM GIS*, pages 230–239, 2005.
13. X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable Spatiotemporal Continuous Query Processing for Location-aware services. In *International Conference on Scientific and Statistical Database Management*, page 317, 2004.
14. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries. In *SIGMOD*, pages 443–454, 2003.