Scalable Continuous Query Processing and Moving Object Indexing in Spatio-temporal Databases

Xiaopeng Xiong

Department of Computer Science, Purdue University
xxiong@cs.purdue.edu

Abstract. Spatio-temporal database systems aim to answer continuous spatio-temporal queries issued over moving objects. In many scenarios such as in a wide area, the number of outstanding queries and the number of moving objects are so large that a server fails to process queries promptly. In our work, we aim to develop scalable techniques for spatio-temporal database systems. We focus on two aspects of spatio-temporal database systems: 1) the query processing algorithms for a large set of concurrent queries, and 2) the underlying indexing structures for constantly moving objects. For continuous query processing, we explore the techniques of *Incremental Evaluation* and *Shared Execution*, especially to k-nearest-neighbor queries. For moving object indexing, we utilize *Update Memos* to support frequent updates efficiently in spatial indexes such as R-trees. In this paper, we first identify the challenges towards scalable spatio-temporal databases, then review the current contributions we have achieved so far and discuss future research directions.

1 Challenges and Motivations

The integration of position locators and mobile devices enables new pervasive location-aware computing environments [3, 32] where all objects of interest can determine their locations. In such environments, moving objects move continuously and send location updates periodically to spatio-temporal databases. Spatio-temporal database servers index the locations of moving objects and process outstanding continuous queries. Characterized by a large number of moving objects and a large number of continuous spatio-temporal queries, spatio-temporal databases are required to exhibit high scalability in terms of the number of moving objects and the number of continuous queries.

To increase the scalability of spatio-temporal databases, there exist two main challenges. The first challenge is to support a large set of continuous queries concurrently. With the ubiquity and pervasiveness of location-aware devices and services, a set of continuous queries execute simultaneously in a spatio-temporal database server. In the case that the number of queries is too large, the performance of the database degrades and queries suffer long response time. Because of the real-timeliness of the location-aware applications, long delay makes the

query answers obsolete. Therefore, new query processing algorithms addressing both efficiency and scalability are required for answering a set of concurrent spatio-temporal queries.

The second challenge for building scalable spatio-temporal databases is to index moving objects efficiently. Building indexes on moving objects can facilitate significantly query processing in spatio-temporal databases. However, due to the dynamic property of moving objects, the underlying indexing structures will receive numerous updates during a short period of time. Given the fact that update processing is costly, traditional spatial indexes may not be applied directly to spatio-temporal databases. This situation calls for new indexing techniques supporting frequent updates.

The above two challenges motivate us to develop scalable techniques for both continuous query processing and moving object indexing in spatio-temporal databases. Specifically, we propose the *SEA-CNN* algorithm for evaluating a large set of continuous k-Nearest-Neighbor queries. While SEA-CNN addresses continuous k-nearest-neighbor queries, it has potential to extend to other types of queries. Meanwhile, we propose the *RUM-tree* for indexing moving objects by enhancing the standard R-trees with *Update Memos*. The update scheme utilized in the RUM-tree can be applied to other indexes to improve their update performance.

In the rest of the paper, we review our research works conducted so far and discuss future Ph.D. research directions.

2 Current PhD Contributions

In this section, we review the contributions we have achieved to build highly scalable spatio-temporal database management systems. The efforts focus on two aspects: (1) Continuous query processing, especially, k-Nearest-Neighbor query processing and (2) Moving object indexing. For each aspect, we first summarize the related works and then generalize our current work. In the following discussion, we assume a two-dimensional environment where objects move continuously and their locations are sampled to the server from time to time. However, the proposed techniques can be applied to higher dimensional environments as well.

2.1 SEA-CNN: Shared Execution Algorithm for Continuous k-Nearest Neighbor Queries

Related Work. The scalability in spatio-temporal queries has been addressed recently in [9, 19, 23, 34, 39, 54]. The main idea is to provide the ability to evaluate concurrently a set of continuous spatio-temporal queries. Specifically, these algorithms work for stationary range queries [9, 39], distributed systems [19], or continuous range queries [34, 54]. Utilizing a *shared-execution* paradigm as a means to achieve scalability has been used successfully in many applications, e.g., in NiagaraCQ [14] for web queries, in PSoup [12, 13] for streaming queries, and in SINA [34] for continuous spatio-temporal range query. However, to our

best knowledge, there has no former work that addresses the scalability issue of k-Nearest-Neighbor queries.

K-nearest-neighbor queries are well-studied in traditional databases (e.g., see [21, 26, 36, 41]). The main idea is to traverse a static R-tree-like structure [20] using *"branch and bound"* algorithms. For spatio-temporal databases, a direct extension of traditional techniques is to use branch and bound techniques for TPR-tree-like structures [7, 29]. The TPR-tree family (e.g., [42, 43, 49]) indexes moving objects given their future trajectory movements. Continuous k-nearest-neighbor queries (CkNN) are first addressed in [44] from the modeling and query language perspectives. Recently, three approaches have been proposed to address CkNN queries [22, 46, 48]. Mainly, these approaches are based on: (1) Sampling [46]. Snapshot queries are reevaluated with each location change of the moving query. At each evaluation time, the query may get benefit from the previous result of the last evaluation. (2) Trajectory [22, 48]. Snapshot queries are evaluated based on the knowledge of the future trajectory. Once the trajectory information is changed, the query needs to be reevaluated. However, the scalability issue of k-Nearest-Neighbor query has not been addressed by the above works yet.

Orthogonal but related to our work, are the recently proposed k-NN join algorithms [8, 51]. The k-nearest-neighbor join operation combines each point of one data set with its k-nearest-neighbors in another data set. The main idea is to use either an R-tree [8] or the so-called *G-ordering* [51] for indexing static objects from both data sets. Then, both R-trees or G-ordered sorted data from the two data sets are joined either with an R-tree join or a nested-loops join algorithm, respectively. The CkNN problem is similar in spirit to that of [8, 51]. However, we focus on spatio-temporal applications where both objects and queries are highly dynamic and continuously change their locations.

Our Contributions. In [53], we propose, SEA-CNN, a *Shared Execution Algorithm* for evaluating a large set of CkNN queries continuously. SEA-CNN introduces a general framework for processing large numbers of simultaneous CkNN queries. SEA-CNN is applicable to all mutability combinations of objects and queries, namely, SEA-CNN can deal with: (1) Stationary queries issued on moving objects (e.g., "Continuously find the three nearest taxis to my hotel"). (2) Moving queries issued on stationary objects (e.g., "Continuously report the 5 nearest gas stations while I am driving"). (3) Moving queries issued on moving objects (e.g., "Continuously find the nearest tank in the battlefield until I reach my destination"). In contrast to former work, SEA-CNN does not make any assumptions about the movement of objects, e.g., the objects' velocities and shapes of trajectories.

Unlike traditional snapshot queries, the most important issue in processing continuous queries is to maintain the query answer continuously rather than to obtain the initial answer. The cost of evaluating an initial query answer is amortized by the long running time of continuous queries. Thus, our objective in SEA-CNN is not to propose another kNN algorithm. In fact, any existing

algorithm for kNN queries can be utilized by SEA-CNN to initialize the answer of a CkNN query. In contrast, SEA-CNN focuses on maintaining the query answer continuously during the motion of objects/queries.

SEA-CNN is designed with two distinguishing features: (1) Incremental evaluation based on former query answers, and (2) Scalability in terms of the number of moving objects and the number of CkNN queries. Incremental evaluation entails that only queries whose answers are affected by the motion of objects or queries are reevaluated. SEA-CNN associates a searching region with each CkNN query. The searching region narrows the scope of a CkNN's reevaluation. The scalability of SEA-CNN is achieved by employing a *shared execution* paradigm on concurrently running queries. Shared execution entails that all the concurrent CkNNs along with their associated searching regions are grouped into a common query table. Thus, the problem of evaluating numerous CkNN queries reduces to performing a spatial join operation between the query table and the set of moving objects (the object table).

During the course of execution, SEA-CNN groups CkNN queries in a query table. Each entry stores the information of the corresponding query along with its searching region. Instead of processing the incoming update information as soon as they arrive, SEA-CNN buffers the updates and periodically flushes them into a disk-based structure. During the flushing of updates, SEA-CNN associates a searching region with each query entry. Then, SEA-CNN performs a spatial join between the moving objects table and the moving queries table.

By combining incremental evaluation and shared execution, SEA-CNN achieves both efficiency and scalability. In [53], we provide theoretical analysis of SEA-CNN in terms of its execution cost and memory requirements, and the effects of other tunable parameters. We also provide a comprehensive set of experiments demonstrating that, in comparison to other R-tree-based CkNN techniques, SEA-CNN is highly scalable and is more efficient in terms of I/O and CPU costs.

2.2 RUM-tree: R-trees with Update Memos

Related Work. As one of the dominant choices for indexing spatial objects, the R-tree [20] and the R*-tree [6] exhibit superior search performance in spatial databases. However, R-trees were originally designed for static data where updates rarely happen. The R-tree is not directly applicable to dynamic location-aware environments due to their costly update operation. To facilitate the processing of continuous spatio-temporal queries, for the past decade, many research efforts focus on developing indexes on spatio-temporal objects (e.g., see [33] for a survey). There are two main categories for indexing spatio-temporal objects: (1) trajectory-based, and (2) sampling-based. For the object trajectory based indexing, four approaches have been investigated: (1) Duality transformation (e.g., see [1, 16, 27, 37]), (2) Quad-tree-based methods (e.g., see [50]), (3) R-tree-based index structures (e.g., see [38, 39, 42, 43, 49]), and (4) B-tree-based structures [24]. For the sampling-based indexing, the Lazy-update R-tree (LUR-tree) [28] modifies the original R-tree structure

to support frequent updates. A hash-based structure is used in [45, 47] where the space is partitioned into a set of overlapped zones. SETI [10] is a logical index structure that divides the space into non-overlapped zones. Grid-based structures have been used to maintain only the current locations of moving objects (e.g., see [19, 34, 53]). One common limitation of the above techniques is that the corresponding old entry has to be removed from the index when an update happens. On the contrary, one unique feature of our work is to allow old entries of an object co-exist with the latest entry.

Our Contributions. In [52], we propose the RUM-tree (stands for R-tree with Update Memo) that aims to minimize the update cost in R-trees. The main idea behind the RUM-tree is as follows. When an update happens, the old entry of the data item is not required to be removed. Instead, the old entry is allowed to co-exist with newer entries before it is removed later. In the RUM-tree, specially designed *Garbage Cleaners* are employed to periodically remove obsolete entries in bulks.

In the RUM-tree, each leaf entry is assigned a *stamp* when the entry is inserted into the tree. The stamp places a temporal relationship among leaf entries, i.e., an entry with a smaller stamp was inserted before an entry with a larger stamp. Accordingly, the leaf entry of the RUM-tree is extended to enclose the identifier of the stored object and the assigned stamp number.

The RUM-tree maintains an auxiliary structure, termed the *Update Memo* (UM, for short). The main purpose of UM is to distinguish the obsolete entries from the latest entries. UM contains entries of the form: $(oid, S_{latest}, N_{old})$, where *oid* is an object identifier, S_{latest} is the *stamp* of the *latest* entry of the object *oid*, and N_{old} is the maximum number of *obsolete* entries for the object *oid* in the RUM-tree. As an example, a UM entry $(O_{99}, 1000, 2)$ entails that in the RUM-tree there exist at most two *obsolete* entries for the object O_{99}, and that the *latest* entry of O_{99} bears the *stamp* of 1000. To accelerate searching, the update memo is hashed on the *oid* attribute.

The RUM-tree employs *Garbage Cleaners* to limit the number of obsolete entries in the tree and to limit the size of UM. The garbage cleaner deletes the obsolete entries *lazily* and in *batches*. Deleting *lazily* means that obsolete entries are not removed immediately; Deleting in *batches* means that multiple obsolete entries in the same leaf node are removed at the same time.

We explore two mechanisms of garbage cleaning in the RUM-tree. The first mechanism of garbage cleaning makes use of the notion of *cleaning tokens*. A *cleaning token* is a logical token that traverses all leaf nodes of the RUM-tree horizontally. The token is passed from one leaf node to the next every time when the RUM-tree receives a certain number of updates. The node holding a cleaning token inspects all entries in the node and cleans its obsolete entries, and then passes the token to the next leaf node after I updates. To locate the next leaf node quickly, the leaf nodes of the RUM-tree are doubly-linked in cycle. To speed up the cleaning process, multiple cleaning tokens may work in parallel in the garbage cleaner. In this case, each token serves a subset of the

leaf nodes. Besides the cleaning tokens, another *clean-upon-touch* mechanism of garbage cleaning is performed whenever a leaf node is accessed during an insert/update. As a side effect of insert/update, such clean-upon-touch process does not incur extra disk accesses. When working with the cleaning tokens, the clean-upon-touch reduces the garbage ratio and the size of UM dramatically.

With *garbage cleaners*, the size of UM is kept rather small and can practically fit in main memory of nowadays machines. To check whether an RUM-tree entry is an obsolete entry or not, we just need to compare the stamp number of entry with the S_{latest} of the corresponding UM entry. If the two values are equivalent, the RUM-tree entry is the latest entry for the object. Otherwise, the entry is an obsolete entry.

The Update Memo eliminates the need to delete the old data item from the index during an update. Therefore, the total cost for update processing is reduced dramatically. The RUM-tree has the following distinguishing advantages: (1) The RUM-tree achieves significantly lower update cost than other R-tree variants while offering similar search performance; (2) The update memo is much smaller than the secondary index used in other approaches, e.g., in [28, 30]. The *garbage cleaner* guarantees an upper-bound on the size of the Update Memo making it practically suitable for main memory; (3) The update performance of the RUM-tree is stable with respect to the changes between consecutive updates, to the extents of moving objects, and to the number of moving objects.

In [52], we present the RUM-tree along with the associated update, insert, delete and range search algorithms inside the RUM-tree. We design a garbage cleaner based on the concept of cleaning tokens to remove obsolete entries efficiently. Further, we theoretically analyze the update costs for the RUM-tree and for other R-tree variants employing top-down or bottom-up update approaches. We also derive an upper-bound on the size of the Update Memo. Furthermore, we conduct a comprehensive set of experiments. The experimental results indicate that the RUM-tree outperforms other R-tree variants, e.g., R*-tree [6] and FUR-tree [30], by up to a factor of eight in the presence of frequent updates.

3 Future Research Directions

There are still many research issues that can be extended from our current work. The future work can be generalized in the following two aspects.

3.1 Continuous Query Processing

Alternative Underlying Indexing Structure. In our current work, the SEA-CNN framework utilizes a grid-based structure to index the current locations of moving objects. In this case, auxiliary indexing structures are required to index the identifiers of both objects and queries. In this research direction, we aim to utilize a more efficient underlying index structure in the SEA-CNN to further boost the query processing. Specifically, we plan to incorporate the memo-based techniques as employed in the RUM-tree into the

grid-based structure to avoid the overhead of auxiliary indexes. In this way, we expect the performance of SEA-CNN can be further improved.

Historical and Predicative Queries. Currently, the *SEA-CNN* framework mainly supports NOW queries, namely, queries only ask for the current status of moving objects. In this research direction, we plan to extend the SEA-CNN framework to support queries that require historical information and future movement predication. To support historical query, SEA-CNN should be coped with efficient indexing structures applicable for historical search. To support future query, SEA-CNN needs to be extended from the sample-based model to the trajectory-based model, thus future movement can be predicted based on the trajectory information.

3.2 Moving Object Indexing

Crash recovery. In this direction, we address the issue of recovering the RUM-tree in the case of system failure. When the system crashes, the information in the update memo is lost. Therefore, our goal is to rebuild the update memo based on the tree on disk. Since the recovery problem is closely related to the logging problem, we aim to design different recovery algorithms based on various logging policies.

Concurrency control. Concurrency control in standard R-trees is provided by Dynamic Granular Locking (DGL) [11]. In this direction, we aim to extend the DGL to support concurrency accesses in the RUM-tree. We investigate the throughput of the RUM-tree under concurrency accesses and compare the performance with other R-tree variants.

Bulk updates. Bulk loading [18, 25, 31, 40] and bulk insertions [2, 5, 15, 17] in R-trees have been explored during the last decade. However, none of the previous works addresses the issue of updating indexed R-tree entries in bulk manners. The main reason is that for a set of updates that will go to the same R-tree node, the corresponding old entries are most likely to reside in different R-tree nodes. Identifying these R-tree nodes containing old entries causes high overhead. On the contrary, reducing an update operation to an insert operation enables the RUM-tree to support bulk updates efficiently. Since there are no deletions of old entries, bulk updates in the RUM-tree can be performed in a way similar to bulk insertions in ordinary R-trees. In this research direction, we aim to propose efficient and scalable bulk update approaches based on the RUM-tree structure.

Extensions to Other Indexing Structures. The proposed update memo inside the RUM-tree is general in the sense that it is not limited to the R-trees, or limited to spatial indexes. The update scheme employed by the RUM-tree can potentially be applied to many other spatial and non-spatial indexes to enhance their update performance. Currently, we are investigating the update

performance of the enhanced Grid File [35] by applying an update memo to the original structure. In the near future, we plan to apply our techniques to more index structures, e.g., SP-GiST [4] and B-trees.

4 Acknowledgements

This work is supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120.

References

1. Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing Moving Points. In *PODS*, May 2000.
2. Ning An, Kothuri Venkata Ravi Kanth, and Siva Ravada. Improving performance with bulk-inserts in oracle r-trees. In *VLDB*, 2003.
3. Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Pervasive Location Aware Computing Environments (PLACE). http://www.cs.purdue.edu/place/.
4. Walid G. Aref and Ihab F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems , JIIS*, 17(2-3), 2001.
5. Lars Arge, Klaus Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1), 2002.
6. Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.
7. Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.
8. Christian Bohm and Florian Krebs. The k-Nearest Neighbor Join: Turbo Charging the KDD Process. In *Knowledge and Information Systems (KAIS), in print*, 2004.
9. Ying Cai, Kien A. Hua, and Guohong Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *Mobile Data Management, MDM*, 2004.
10. V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, 2003.
11. Kaushik Chakrabarti and Sharad Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. In *ICDE*, 1998.
12. Sirish Chandrasekaran and Michael J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.
13. Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2), 2003.
14. Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.
15. Li Chen, Rupesh Choubey, and Elke A. Rundensteiner. Bulk-insertions into r-trees using the small-tree-large-tree approach. In *GIS*, 1998.
16. Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi. Storage and Retrieval of Moving Objects. In *Mobile Data Management*, January 2001.

17. Rupesh Choubey, Li Chen, and Elke A. Rundensteiner. Gbi: A generalized r-tree bulk-insertion strategy. In *SSD*, 1999.
18. Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, 1997.
19. Bugra Gedik and Ling Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.
20. Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
21. Gsli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *TODS*, 24(2), 1999.
22. Glenn S. Iwerks, Hanan Samet, and Ken Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, 2003.
23. Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *VLDB*, 2004.
24. Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.
25. Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *CIKM*, 1993.
26. Norio Katayama and Shin'ichi Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *SIGMOD*, May 1997.
27. George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On Indexing Mobile Objects. In *PODS*, 1999.
28. Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.
29. Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic Queries over Mobile Objects. In *EDBT*, 2002.
30. Mong-Li Lee, Wynne Hsu, Christian S. Jensen, and Keng Lik Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.
31. Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, 1997.
32. Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Towards Scalable Location-aware Services: Requirements and Research Issues. In *GIS*, 2003.
33. Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2), 2003.
34. Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.
35. J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1), 1984.
36. Apostolos Papadopoulos and Yannis Manolopoulos. Performance of Nearest Neighbor Queries in R-Trees. In *ICDT*, 1997.
37. Jignesh M. Patel, Yun Chen, and V. Prasad Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD*, 2004.
38. Kriengkrai Porkaew, Iosif Lazaridis, and Sharad Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *SSTD*, Redondo Beach, CA, July 2001.
39. Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10), 2002.

40. Yvn J. Garca R, Mario A. Lpez, and Scott T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *GIS*, 1998.
41. Nick Roussopoulos, Stephen Kelley, and Frederic Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.
42. Simonas Saltenis and Christian S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.
43. Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.
44. A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and Querying Moving Objects. In *ICDE*, 1997.
45. Zhexuan Song and Nick Roussopoulos. Hashing Moving Objects. In *Mobile Data Management*, 2001.
46. Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD*, 2001.
47. Zhexuan Song and Nick Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management, MDM*, January 2003.
48. Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous Nearest Neighbor Search. In *VLDB*, 2002.
49. Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.
50. Jamel Tayeb, Özgür Ulusoy, and Ouri Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3), 1998.
51. Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An Efficient Method for KNN Join Processing. In *VLDB*, 2004.
52. Xiaopeng Xiong and Walid G. Aref. R-trees with Update Memos. In *ICDE*, 2006.
53. Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, 2005.
54. Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne Hambrusch, and Sunil Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In *SSDBM*, 2004.