International Journal of Research in Computer Science
eISSN 2249-8265 Volume 2 Issue 1 (2011) pp. 1-6

www.ijorcs.org

QUERY AS REGION PARTITION IN MANAGING MOVING OBJECTS FOR CONCURRENT CONTINUOUS QUERY

Ming Qi[1], Guangzhong Sun[2], Yun Xu[3]

**School of Computer Science and Technology*
University of Science and Technology of China

Abstract: *Applications in location-based services rely on the efficient management of large scale moving objects, and one of the most important operations is real-time continuous query over moving objects, such as monitoring the objects of a certain region. In order to satisfy the high throughput and handle real-time updates and queries, it is a good choice to introduce multi-core parallel processing and managing spatial indices in main memory. In this paper, we propose a new scheme of processing continuous query on a novel spatial index based on uniform grid which is proved to be a good indexing scheme in main memory. The novelty of our index is to manage query region as the partition in the spatial index, which unified the index updating and query processing. Our experiments indicate that this sophisticated approach has better performance especially when the query is rather stationary.*

Keywords: *spatial data indexing, continuous query, query as region partition, parallel computing*

I. INTRODUCTION

Nowadays, moving devices such as GPS devices, location- aware sensors and RFID objects are growing explosively. In the near future, computing will be ubiquitous, and Location- Based Service (LBS) will be widely used in everyday life. Many LBS applications is based on query on spatial data, and many applications have been studied and commercialized, for example, vehicle monitoring [12][13] and finding friends or services near your site[14].

To efficiently support these LBS applications, several spatial/spatial-temporal indices have been studied [1], such as R-tree[8][9], Kd-tree[7] and grid-file[15] etc. In applications with real-time response requirement and high-throughput de- mand, because of the massive users' updating and querying, the performance is very likely to be a problem. To meet with the high-performance and real-time requirement, parallel computing should be considered instead of using single CPU thread. Besides, the update and query requests in fact come to the server concurrently, it is more natural to process these requests in parallel to minimize the response time.

One of the most important techniques in LBS applications is for managing and querying moving objects. And much more often than not, these query is continuous query [10] [13] rather than snapshot queries, which makes one of the distinguished characteristics of LBS [10].

Besides, since the main memory has been much more inexpensive and much more available in large capacity, the index structures used for real-time query should be in main memory instead of in the hard disk.

So, in this paper, we mainly focus on the management of massive moving objects, indexing them in main memory and providing the read-time response for location updating and continuous queries. To enhance the performance, we use multi- thread share memory parallelization on multi-core systems, with effective concurrency controls. The contributions of this paper are mainly:

1. An novel index structure for moving object management, which is using query as partition as well, and unifying the updating of the location and updating of the query result.
2. Sophisticated algorithms for update and query operations that are especially designed for our index. With these algorithms, performance of continuous query will be greatly improved.
3. Concurrency controls especially designed for moving object management with different lock grain levels, which are designed respectively for high-throughput operations and the less-frequent structure alternation operation.
4. And both algorithms and concurrency controls are analyzed and experimented with real-sized environments.

The rest of this paper is organized as follows: Section 2 reviews some related works about moving objects indexing, continuous query, as well as concurrency controls; Section 3 describes the index structure and its operations; Section 4 provides concurrency controls of updating and querying in detail, with correctness proved briefly; experiments of the update and query performance are in Section 6; and finally some conclusions and future works are discussed in Section 7.

II. RELATED WORKS

With the rapid growth of mobile devices, moving object management and spatial access methods become a hot re- search field. Paper [5] compares uniform grid structure and structures based on R-tree for indexing moving objects in main memory, which shows that “the grid can compete with R- tree in query performance and robust to varying parameters of workloads” [5]. However, paper [5] does not involve con- currency or parallel processing, which could not utilize the multi-core resources.

Besides the uniform grid and R-tree, kd-tree [6] and KDB- tree [7] are also indices for 2-dimensional points, and regional kd-tree is also a good choice for managing moving objects in main memory. However, the design and analyze of a concurrent regional kd-tree is to be studied.

Concurrent location management for moving objects using Blink -tree on space-filling and R-tree or its variants has been studied by [11] and [12], which mainly used coarse-grain locks on regional cells and locking coupling schemes. Paper [13] gives a concurrent continuous query scheme on B-tree- based framework with two specially designed hash tables, and delicate concurrency controls to ensure high-level consistency.

Paper [2] studied in-memory structure for continuous queries, and provides a two layer grid scheme, which is with an optional second layer of refined uniform grid to resolve the data skewness. However, a hybrid structure with uniform grid and regional kd-tree is more convenient to resolve the skewness problem and control the number of objects within a threshold; and second layer of grid may arise unnecessary partitions. Meanwhile, neither concurrent access method nor concurrency control is discussed in this paper.

The previous works about continuous query, however, have a deficiency in common, that they did not use the data inside the structure itself to format the query result but use a separate query result set. This separation causes data redundancy as well as the complexity of data management, especially in concurrent environments.

III. NOVEL INDEX STRUCTURE

As we surveyed in the related studies, spatial indices with regional partitioning is the primary access method for indexing moving points in main memory. Based on the fact that the query in LBS is rather continuous and stationary instead of dynamic, our approach is to also use the continuous query region as part of the regional partition, so that the update and query algorithms are greatly simplified.

B. Query as Regional Partition

A. Problem Formulation and Operations

Assumptions are made toward this problem: Each object is abstracted to a 2-dimensional point; all the index structures are in the main memory, and additional attributes related to the specific application are in the secondary storage, which is not considered in this paper.

The moving objects management contains mainly four types of operations:

A **where-am-I** operation is to find the given location in the index, which is usually used in the object location update.

An **object location update** operation is given the object id and its new location coordinate, to find this object, and if the old and new locations are in different region, update the object in the structure according the new location. Traditionally, the location updating can be seen as a transaction of a deletion operation followed by an insert operation.

A **continuous region query** operation is to retrieve the objects which are now inside the query.

A **query alteration** operation is to create, delete or alter the continuous query by setting or modifying the location or the region of the query. In the LBS applications, query alteration operation is much fewer than the continuous region query operation. Each query is a rectangle region with both sides parallel to the coordinate axis, and to return the set of objects inside this region.

The update and continuous query operations should process concurrently in multi-thread environment, so concurrency controls should be introduced to ensure the data consistency. In order to achieve efficient concurrent operations of the moving objects, the index structure should have the following characteristics:

1. The structure should be suitable to be in main memory.
2. The whole map should be divided into regions, each region contains a bucket of objects.
3. Given a location or an object id, find its region bucket efficiently.
4. Update operations and continuous query operations in the index should have efficient concurrent implementation.
5. The query alteration operation should be implemented in good amortized performance.

In this section, we use query as regional partition scheme on the uniform grid and kd-tree respectively, from those we can achieve both much more efficient continuous query as well as locate update.

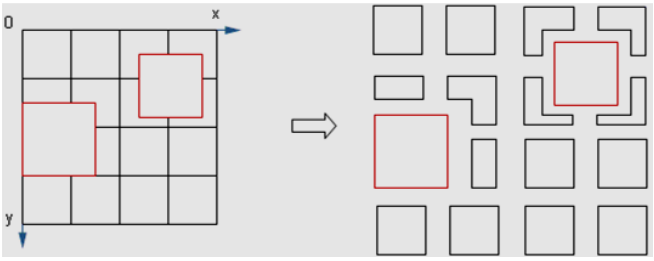

Fig. 1. Query as Region Partition in Uniform Grid

To use query as partition in uniform grid is demonstrated in Figure 1. In every cell bucket of the uniform grid, the grid maintains a list of references of the query regions that are overlapping with this grid cell. And the region of this cell is presented as the former rectangle region minus the overlapped region. There may be more than one query regions which are overlapping with this cell region, and one query may be overlapping with several grid cells.

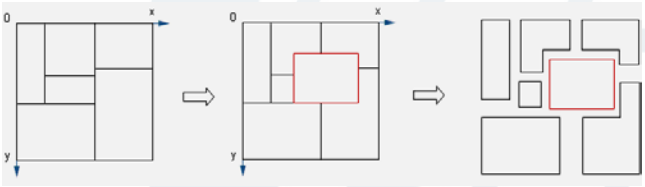

Fig. 2. Query as Region Partition in Regional Kd-tree

In kd-tree, the query region can be managed the same as in the uniform grid, instead that if the kd-tree region is splitting or merging, the overlapping query references list will be changed accordingly. Differently from the kd-tree regions, each query region is a sustainable region that does not have overflow threshold or any splitting strategy.

The query region can be overlapped with each other. And, in our scheme, when query region is overlapping each other, the objects in the intersect subregion is to be duplicated and in each of the query results. This is reasonable, because every query should has its result set.

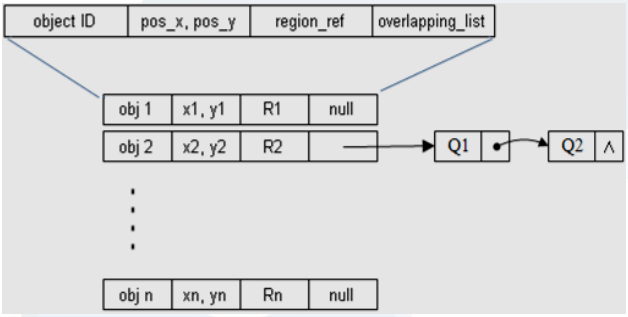

Fig. 3. The Auxiliary Index on Object ID

The auxiliary index, which is in fact a hash table on object id, is like Figure 3. In the auxiliary, we can get the region reference (either normal region or query region) which the object is in. And, if this object has duplicates in overlapping query regions, a list of reference of the overlapping regions in maintained in the overlapping_list in the auxiliary index.

C. Algorithms of Updating and Continuous Query

Through the spatial index and the auxiliary index, we can then describe the algorithms for our operations. Finding the region bucket of a given position, which is also the where-am-I operation is trivial in uniform grid and also very straight forward in the kd-tree. And, in our scheme which uses make query region the same as region partition, the continuous query operation is just return the content that inside the query region. The following is the algorithm of object location updating, in which we assume query region has the same behavior with other grid/kd-tree regions, except when the query regions are overlapping.

OBJECTUPDATE(objectid, newpos)

1. oldpos = getPosByAux(objectid);
2. **if** newpos is not in R1
3. R2 = getRegionByPos(newpos)
4. SwitchBetweenRegion(R1, R2, objectid);
5. Regions set O1 = OverlapSetInAux(objectid);
6. Regions set O2 = getOverlapByPos(newpos);
7. Regions set O = O1 ∩ O2;
8. **for** each region R in set O if O is not empty
9. **if** the newpos is inside R and the oldpos is not
10. insert a duplicate objectid into R
11. insert R into the O1
12. **if** the oldpos is inside R and the newpos is not
13. delete the element with objectid form R
14. delete R from O1
15. setAuxPos(newPos);

The old position of the object is get form auxiliary index by object ID. The function getRegionByPos(pos) in this algorithm means to get the first query covers pos, and if this grid cell have none query or none of the queries covers this pos, return the cell region itself(in this situation the pos is not in any queries but in the cell region itself). On the contrary, the function getOverlapByPos(pos) in this algorithm means to get the queries that cover this pos after the first matched query, and if no other query is matching, return empty. So, in this algorithm, each element in the region set O1 and O2 this regions must be an overlapping query.

In the real applications, the non-overlapping is common while the overlapping query is relatively much fewer. So that the main work of object update is to get two regions and switch the object from the old one to the new one. If in the application system, the queries are definitely without any of the query overlapping, this algorithm can be simplified.

D. Query Creating, Deleting and Altering

If we are to create a new query into a uniform grid or kdtree, there are generally three steps.

1. Step 1, initialize the new query with a query region and bucket.
2. Step 2, to get the region cells in the spatial index which are intersecting with the query region. In uniform grid, this step is trivial; and in kd-tree this step is a typical recursively region search, which may be with multiple search paths.
3. Step 3, For each of these cells in Step 2, insert the reference of this query into the query lists of this cell, then move the objects of this cell whose location is in the query region.

If this new query is intersecting with another query, then duplicate the objects of this query whose location is in the query region. However, if this new query is somehow overlapping with more than one previous queries, objects should be duplicated from previous queries but with same object only once.

When a query is ceased to exist, the deleting processing is straightforward, which is to delete the query reference from correspondence grid or kd-tree cells, and reinsert the objects this query contained into the correspondence grid or kd-tree cells. And, if one object is to be reinsert into a overlapping query region, check should be down to notify if this object has already have a object in it, and do nothing if so.

Query altering, which is to alter the size and/or location of the query region, can be down by query deleting and creating. However, as a more sophisticated process is as the follow steps, alike query creating:

1. Step 1, to get the region cells in the spatial index which are affected by this alternation, and modify the query list when necessary.
2. Step 2, check the objects in these cells, move the object to the query bucket if the object is inside the new query region.
3. Step 3, check the objects in the alternating query, if one is to be ousted, reinsert it to other corresponding cells. Similarly, object duplication should be checked if overlapping queries exist.

IV. CONCURRENT CONTROL

Now most commercial computer systems are multi-core computer, mostly with 4 or 8 CPU cores, support 8 or 16 threads. To utilize the multi-core resource and access our index concurrently with data consistent, we should design the concurrent controls.

A. Concurrent Control Techniques

To maintain data consistence of parallel accesses to the index, we introduce our concurrency controls. The most common ways to synchronize a multi-thread program is lock-based synchronization. Concurrency controls with mutexes, critical sections or semaphores can be seen as lock-based concurrency. There are technically two types of lock implementation: one is spin-lock, which is spinning in a loop, waiting for certain conditional status; the other one is to be scheduled by kernel or mutex libraries with a signal waiting and waking up. Mutex locks and semaphores are supported in operating systems, some programming languages and thread libraries. In paper[11], locks are implemented by Java synchronized objects. In our paper, we use spin lock and pthread_rwlock functions as synchronizing locks.

The spin lock this paper used can be described as pseudocode:

```
1. while (status ≠ condition) {
2.     _asm PAUSE
3.     sleep(0)
4. }
```

PAUSE instruction here is to indicate the CPU to stop instruction prefetching, which can reduce the cost of instruction emptying. sleep(0) can give CPU ownership to other thread once while there is an another ready thread, and keep running while there is no other threads, so that this spin lock do not block any other ready threads, and the whole system running smoothly although the CPU load is 100%.

The auxiliary index is accessed only by object id, whose structure only change when the an object is creating or deleting; so, it is convenient to be implemented by a threadsafe hash map. So in this paper, we do not pay much attention to the concurrency control of the auxiliary index.

There is an optional implementation that using two different phrases for updating: during the querying phrase all update requests are buffered and during the executing phrase all the buffered requests are executed in a batch. This is the simplified implementation many system uses. However, in our concurrency controls, the reads and writes are mixed together, because the latest location should be provided to the query user, and this is especially important when the updating is very frequent and query is in realtime.

Another idea is that managing the moving objects in a proper transactional system such as Berkeley DB. However, there is no proper database systems specifically suit for the high frequent moving object management and continuous query, for which we concern this paper.

B. Fine-grain Lock on Updating and Querying

If the buckets in the index are implemented as a linked list, as we do in our paper, we can moving objects (linked list node) between different linked lists

using a fine-grain lock. This fine grain lock only locks the deleting list node and two involving list nodes which indicate the deleting and inserting position:

First, find the deleting list node by lock coupling, and lock the deleting node while its predecessor node is also locked; in the new bucket, lock the insert position. Beware of the order of locking to avoid deadlock. Then, switch the next pointers so that delete the moving object from one list and insert into another. Finally, unlock all the three nodes after pointers switching. An example of fine-grain locked updating of linked list bucket is shown in Figure 4(moving obj2 from bucket1 to bucket2).

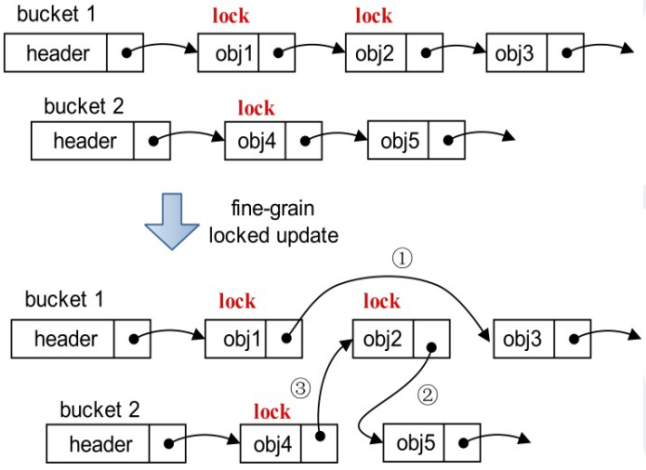

Fig. 4. The fine-grain locked update

The fine-grain lock scheme generally has better concurrent performance, due to its refined lock and higher concurrency. And, through the fine-grain lock, the object moving between regions can be seen as atomic operation, so that the inconsis- tent status is avoided using the fine-grain lock.

C. Coarse-grain Lock on Structure Alteration

For the operations of query creating, deleting and altering, as well as region splitting/merging in kd-tree, the coarse grain lock should be used to protect the whole area that the operation is affected.

We use a read-lock, SIX-lock, and exclusive-lock semaphore to implement the coarse-grain lock. Before the altering operation, SIX-locks on these regions are acquired, so that other operations cannot proceed. Then, for every region should be locked, upgrade the SIX-lock to exclusive-lock when all readlocks of this region is released. Finally, after the alteration operation, all the exclusive-locks are released.

Since there is nothing done to make sure a thread is eventually get the lock, this coarse-grain lock is deadlock-free but not starvation-free.

V. EXPERIMENTS AND RESULTS

To evaluate the performance of our query-as-partition index and concurrency controls, we introduce a series of comparison experiments. The data set we mainly used is moving object set on the road network in the city of Oldenburg [17]. In our experiments we have compared between uniform and kd-tree index structures, various query/alteration ratio, and different concurrent thread number.

In our experiment, the Oldenburg data set we use contains 200,000 objects with coordinate (x; y) in the area about 30_30km2. The queries we use is random generated queries with random size within 6 _ 6km2. In our experiment, we used a common 2-CPU server with shared memory; each CPU is 4-core Intel Xeon E5430 2.66GHz. The reason why we use a commercial server rather a super computer is that, for most moving objects management applications, a commercial server is enough, and using a cluster of common servers usually receives higher cost-effectiveness. Our programs are implemented by gcc with POSIX pthread, using spin-lock and pthread rwlock.

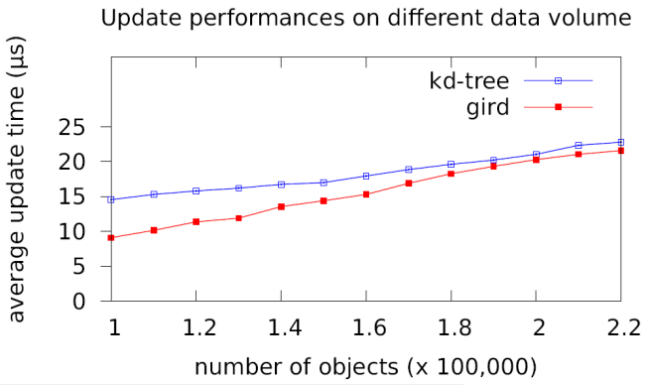

Fig. 5. Update Performances on Grid and Kd-tree

We compare the performance on uniform grid and kd-tree on various data volume in Figure 5. Generally, uniform grid outperforms kd-tree in updating, because in grid, given a new position, where-am-I operation is in O(1), while in kd-tree the where-am-I complexity is O(lgB) (B is the bucket number all over this index).

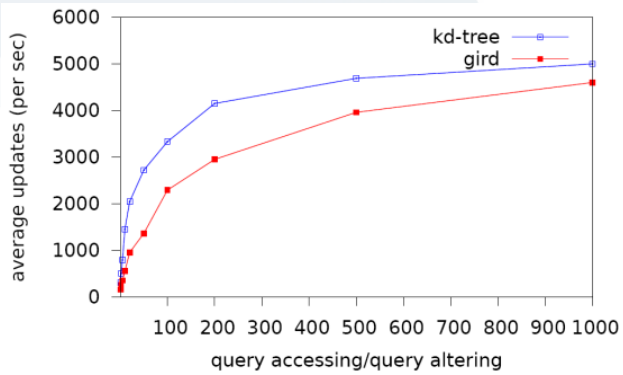

Fig. 6. Query Throughput on Various Query/Alteration Ratio

In our scheme, continuous query processing and query region alteration have very different computing complexity, so the overall performance of query is greatly affected by the alteration ratio, Figure 6. When the alteration ratio is 1:1, the query is not continuous at all, our scheme do not have any advantage; when the queries are rather continuous, the query operation is just to return the content of the query bucket, which essentially reduced the complexity of the query.

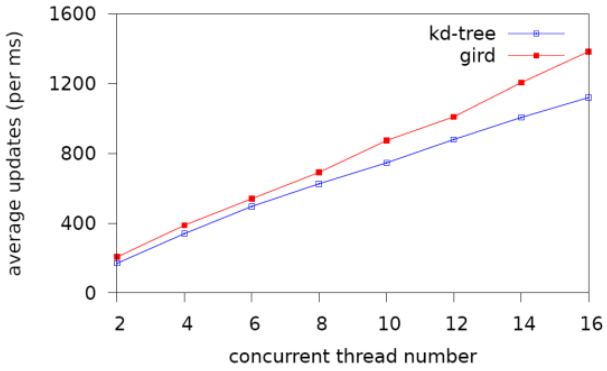

Fig. 7. Update Throughput Speedup

Using our concurrency control, we can access our index and proceed the operations concurrently. The update throughput scaling on various thread number is shown in Figure 7. Since our fine-grain lock has minimized the concurrent collision, the uniform grid gains nearly linear speedup, and in the kd-tree, since there are tree node splitting and merging, the speedup is a little lower.

VI. CONCLUSIONS

Through our designing and experimenting on our indices and concurrency controls, we can conclude that: using query region as spatial partition has a very distinct predominance to previous continuous query schemes, especially when the queries is rather stationary. Comparing the uniform grid and the kd-tree used in memory for moving object management, the uniform grid outperforms kd-tree due to its simplicity. And, using our concurrent control, both uniform grid and kd-tree obtain a good parallel speedup.

The structures and techniques in this paper can give us a new vision on moving object management indexing, which can also be used in other similar applications. Further study may focus on indexing moving object with multiple attributes, more sophisticated concurrent continuous query and kNN query, or using more specialized index structures effectively in realistic problems.

VII.REFERENCES

[1] Volker Gaede, Oliver G¨unther. Multidimensional Access Methods. In ACM Computing Surveys (CSUR) , Volume 30 Issue 2, June 1998. doi:10.1145/280277.280279

[2] D. Kalashnikov, S. Prabhakar, S. Hambrusch, Main Memory Evaluation of Monitoring Queries Over Moving Objects, Distributed and Parallel Databases: An International Journal, Vol. 15, No. 2, pages 117-136, 2004. doi:10.1023/B:DAPD.0000013068.25976.88

[3] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Elsevier 2008.

[4] Hector Garcia-Molina, Jeffret D. Ullman, Jennifer Widom. Database System: Implementation, Second Edition. Pearson Education Inc. 2009.

[5] Darius ˇ Sdlauskas, Simonas ˇ Sltenis, Christian W. Christiansen, Jan M. Johansen, Donatas ˇ Sulys. Trees or Grids? Indexing Moving Objects in Main Memory. In ACM GIS'09, November 4-6, 2009. Seattle, WA, USA.

[6] Hanan Samet. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers, 2006.

[7] John T. Robinson. The K-D-B-Tree: A Search Structure for Large Multideminsional Dynamic Indexes. In Proceedings of the ACM SIGMOD International conference on Management of data , 1981.

[8] Diane Greene. An Implementation and Performance Analysis of Spatial Data Access Methods. In Proceedings. Fifth International Conference on Data Engineering, 1989. doi:10.1109/ICDE.1989.47268

[9] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. 14, 2 (June 1984), 47-57. doi:10.1145/971697.602266

[10] Xiaopeng Xiong, Mohamed F. Mokbel, etc. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In Proceedings of 16th International Conference on Scientific and Statistical Database Management 2004,(SSDBM04).

[11] Jing Dai, Chang-Tien Lu. CLAM: Concurrent Location Management for Moving Objects. In ACM International Symposium on Advances in Geographic Information Systems, New York, NY, USA, 2007.

[12] Chang-Tien Lu, Jing Dai, Ying Jin, Janak Mathuria. GLIP: A Concurrency Control Protocol for Clipping Indexing, IEEE Transaction on Knowledge and Data Engineering, Vol.21, No.5, 2009. doi:10.1109/TKDE.2008.183

[13] Jing Dai, Chang-Tien Lu, Lien-Fu Lai. A Concurrency Control Protocol for Continuously Monitoring Moving Objects. In Tenth International Conference on Mobile Data Management: Systems, Services and Middleware 2009. doi:10.1109/MDM.2009.24

[14] digu.com http://2011.digu.com/indexMap

[15] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Trans. Database Syst. 1984. doi:10.1145/348.318586

[16] Harris, T. L., Fraser, K., and Pratt, I. A. 2002. A Practical Multi-word Compare-and-Swap Operation. In Proceedings of the 16th international Conference on Distributed Computing, 2002

[17] Thomas Brinkhoff. Generating Network-Based Moving Objects. In Proceedings 12th International Conference on Scientific and Statistical Database Management, 2000, IEEE Computer Society Press. doi:10.1109/SSDM.2000.869794

How to cite

Ming Qi, Guangzhong Sun, Yun Xu, "Query as Region Partition in Managing Moving Objects for Concurrent Continuous Query". *International Journal of Research in Computer Science*, 2 (1): pp. 1-6, December 2011. doi:10.7815/ijorcs.21.2011.008