LETTER

Continuous Range Query Processing over Moving Objects

Yong Hun PARK[†], **Kyoung Soo BOK**[††], ***Nonmembers***, ***and*** **Jae Soo YOO**[†a)], ***Member***

SUMMARY In this paper, we propose a continuous range query processing method over moving objects. To efficiently process continuous range queries, we design a main-memory-based query index that uses smaller storage and significantly reduces the query processing time. We show through performance evaluation that the proposed method outperforms the existing methods.

key words: *databases, location based services, moving objects, continuous queries*

1. Introduction

To provide high-quality location-based services, one of the most important technical problems is the efficient query processing methods of moving objects [1], [2]. When objects frequently move, the results of many continuous queries must be changed rapidly. Therefore, such continuous queries are needed for location-based services. Continuous range queries are used to monitor moving objects inside the query ranges [6]. For an example, a taxi cab service company can dispatch a taxi to a customer at a specific location using the result of a continuous range query such as "Find all the taxi cabs currently located within 5 blocks away from a specific location." As another example, for a security service, a continuous range query such as "Find all people who enter a security range without permission." can be used.

A method of continuous range query processing using a query index was first proposed in [6] and many methods have been proposed until recently [3]–[5]. These methods are classified by disk-based methods [5], [6] and main-memory-based methods [3], [4]. We only mention the main-memory-based method because our method is based on main-memory and the disk-based method is not as efficient as the main-memory-based method even if the disk-based methods are modified for main-memory access [4]. The CES method shows better performance than other existing methods [4].

The existing methods have performance limitations caused by two characteristics of query indices. First, query identifiers are used to find which queries are affected by the movement of an object. In other words, for the affected queries, it modifies the query results when an object enters into query ranges newly or goes out from query ranges in which the object lay previously. To efficiently process continuous range queries, it is important to quickly and efficiently find which queries are affected by the movement of objects. If the process is conducted with query identifiers, it needs to compare between the set of query identifiers which contain the previous position of an object and the set of query identifiers which contain the new position of an object. The process involves a number of operations to compare objects of one set and objects of another set. In addition, the process is conducted although there are no queries affected by the movement of an object. Second, a query identifier is redundantly stored in each cell contained in the query range to manage the relationship between cells and queries and leads to a waste of storage space. In the memory based method [3], [4], the query index uses the grid structure composed of the same-sized cells and the number of redundantly saving query identifiers is increased according to the growth of query range.

To overcome the limitations of the existing methods, we propose a novel query index. Each query has a bit identifier and each cell in a grid has a bit pattern which represents the relationship between cells and queries. Using the bit patterns, we can compute quickly which queries overlap a cell in a grid and reduce the number of unnecessary operations by comparing the bit patterns without comparing the query identifiers for query reevaluation when objects move. In addition, the management of cells in the grid by groups prevents from wasting the storage space through the increase of the length of the bit pattern and increasing the comparison costs of bit patterns. We also propose an efficient continuous range query processing method with the index.

The rest of this paper is organized as follows. Section 2 proposes the novel query index and Sect. 3 shows the performance evaluation. Finally, Sect. 4 presents the conclusions of this paper.

2. Continuous Query Processing

2.1 The Proposed Query Index

We propose an efficient continuous range query processing method and a new query index that uses smaller storage and makes the processing time lower than the previous methods. Figure 1 shows an example of the new query index structure. The entire index range is composed of the same-sized

Manuscript received April 28, 2008.
Manuscript revised July 8, 2008.

[†]The authors are with the Department of Computer and Communication Engineering, Chungbuk National University, Cheongju, Korea.
[††]The author is with Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea.
a) E-mail: yjs@cbnu.ac.kr

DOI: 10.1093/ietisy/e91–d.11.2727

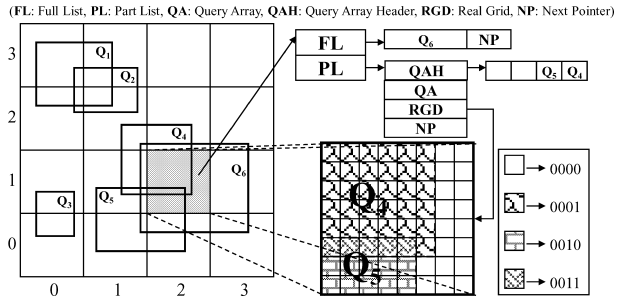
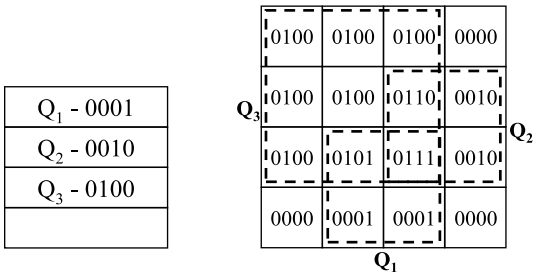


Fig. 1 An example of our proposed query index.



(a) QA in a group (b) Bit parrerns of cells in a group

Fig. 2 Query management and bit patterns of cells.

groups which include cells and a group that has a full list and a part list. The size of the groups is 8×8 in Fig. 1. A full list FL maintains the queries fully containing the group. The FL uses the data structure of the general list to store the query identifiers. A partial list PL maintains the queries partially containing the group cell. For example, since Q_6 fully contains the group (2,1), and Q_4 and Q_5 partially contain it, Q_6 is stored in the FL, and Q_4 and Q_5 are stored in the PL of the group.

In addition, the PL manages information such as a query array (QA), a query array header (QAH), a real grid (RGD), and the next pointer (NP). The query identifiers contained in the group are stored in QA. QAH represents the status of the QA, RGD represents the cells of size 1×1 in the group as a bitmap table, and NP points to the next PL node to support the scalability of the query index. A bit identifier is assigned to each query as a query identifier in a group and each cell has a bit pattern made by the combination of bit identifiers. The bit pattern represents the relationship between the cells and the queries. If the length of QA in PL is set as 8, the bit pattern of each cell is represented in 8 bits. If more than 8 queries are inserted, a new PL node with 8 queries is created and the remaining queries are inserted into the new PL node. The PL nodes are represented in linked lists by NP.

Figure 2 shows the management of queries with bit identifiers and the bit patterns which represent the relationship between cells and queries in a PL. As shown in Fig. 2 (a), the bit identifiers of queries are related to the positions of the query identifiers in a query array, so the bit identifiers don't consume any storage cost. As shown in Fig. 2 (b), the cells with the bit pattern '0000' are not contained in any query, and the cells with the bit pattern '0101' are contained in Q_1 and Q_3. The query search in the query index is conducted using the bit pattern of the cell.

To reduce the storage space and the computation cost, the entire space is partitioned by groups according to the following criteria. The groups are not overlapped each other and their shapes are squares. All groups are also the same in size and the size of a group is determined according to the average size of queries related to G_{pavg} and G_{favg} in the Eq. (3). Query identifiers only overlapped with the groups are stored in the groups. Bit identifiers are assigned to queries in a group. The length of a bit identifier in a group is the same as the number of query identifiers in the group. Therefore, the length of a bit identifier is reasonable. The bit identifier of queries in a group is the same as the length of bit patterns in the group. The shorter a bit pattern is, the smaller the comparison cost of the bit pattern is to process continuous queries.

The Eq. (1) represents the storage cost (S_{pl}) of a PL node. S_{qid} and S_{ptr} represent the size of qid and the size of pointer respectively. L_{bp} represents the length of the bit pattern of a cell in a PL node. C represents the number of cells in a group. '$L_{bp}{}^{*}S_{qid}$' is the size of QA, '$L_{bp}{}^{*}1bit$' is the size of QAH, '$L_{bp}{}^{*}C^{*}1bit$' is the size of RGD in a PL node. '$1^{*}S_{ptr}$' is the size of a pointer for NP in a PL node. The Eq. (2) represents the storage cost (S_{fl}) of a FL node.

$$S_{pl} = L_{bp}{}^{*}S_{qid} + L_{bp}{}^{*}C^{*}1bit + L_{bp}{}^{*}1bit + 1^{*}S_{ptr} \quad (1)$$

$$S_{fl} = S_{qid} + S_{ptr} \quad (2)$$

The Eq. (3) represents the total storage cost (S_{total}) of our index structure. $|G|$ represents the number of groups in the index and $|Q|$ represents the number of queries inserted into the index. G_{pavg} and G_{fage} are the average number of groups that a query overlaps partially and fully respectively.

$$S_{total} = |G|^{*}S_{prt} + |G|^{*}\lceil(G_{pavg}{}^{*}|Q|)/|G|\rceil^{*}S_{pl} + G_{favg}{}^{*}|Q|^{*}S_{fl} \quad (3)$$

'$|G|^{*}S_{prt}$' is the storage cost for the grid structure. '$|G|^{*}\lceil(G_{pavg}{}^{*}|Q|)/|G|\rceil^{*}S_{pl}$' is the storage cost of PL nodes and '$G_{favg}{}^{*}|Q|^{*}S_{fl}$' is the storage cost of FL nodes in the index. As given in the equations, if the size of a group increases, C increases, and G_{pavg} and G_{favg} decreases. G_{pavg} and G_{favg} are also determined according to the size of queries.

In the previous schemes, each cell manages the related queries by query identifiers. However, in our scheme, each cell manages the related queries by bits and the query identifiers are only managed on the group level. In the best case, each cell spends just a bit on a query. So if the number of queries is large, the increase of the storage cost in our scheme is less than that in the previous schemes.

When a query is inserted, we first find all groups that overlap with the query region, and insert the query in these groups. For each group, the query is stored in QA, and the

position of the query stored in QA represents the bit identifier of the query. Then QAH is updated and the bit patterns of all cells overlapped with the region of the query are updated by bitwise OR with the bit identifier of the query. When a query is deleted, the conducted operations are similar to the insertion. We first find all groups that overlap with the query region, and delete the query in these groups. For each group, it needs to find the position of the query stored in QA to know the bit identifier of the query. Then QAH and the bit patterns of all cells overlapped with the region of the query are updated by bitwise XOR with the bit identifier of the query. We assume that queries are static. Nevertheless, if a query moves, it can be conducted with the query insertion and the query deletion.

2.2 Continuous Range Query Processing

To process continuous range queries, query reevaluation should be conducted every time any object moves. If an object has moved within a group, the query evaluation is performed with the bit patterns of cells containing the old and new positions of the moving object. If the bit patterns are not the same, it means that the movement of the object affected some queries. In addition, to find the queries affected by the object, we need to compare only the bit patterns unlike the existing methods. For the reasons, it is applicable to frequent moving objects. If an object has moved out of a group, we can not compare the bit patterns of cells containing the old and new positions of the object. So, query identifiers have to be used to find which queries are affected by the object like the previous works.

Figure 3 (a) shows an example of the movement of an object in a group. The point P_1 is the old position of the object, and P_2 and P_3 are its new positions. Figure 3 (b) shows the query array and Fig. 3 (c) shows the comparison of the two bit patterns in the two cases as an object moves within a group. Through the comparison of the case1 that the object moves to P_2, we notice Q_2 is affected by the movement of the object and the object must be removed from the result set of Q_2. In the case2 of Fig. 3 (c), we notice Q_1, Q_2 and Q_3 are affected by the movement of the object and the object must be removed from the result set of Q_2 and Q_3, and inserted into the result set of Q_1.

In previous works, query identifiers are used and the cost can be represented by 'm*n', where m and n are the number of queries with the old position of an object and the number of queries with its new position, respectively. This is because the two query lists have to be compared each other to know whether the queries are different or not. In contrast, we use the bit identifier assigned to each query in groups and the cost can be represented by 'm+n' since each query needs only one comparison to know that the query is affected by the movement of an object. If an object has moved out of a group, the cost of our method is the same as those of the previous works.

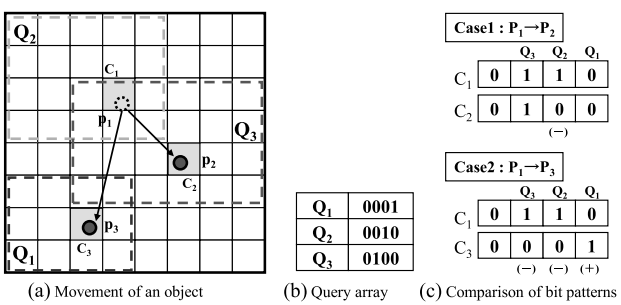


Fig. 3 An example of the movement of an object.

3. Simulation

Our experiments were performed on a Dell PC with a 3-GHz Pentium 4 processor and 1 Gbyte RAM running Windows XP. We compare our proposed method (BitID) with the CES because the CES outperforms other methods [4]. Therefore, the parameters for simulation are the same as the parameters used in [4]. In the simulations, the index region is defined by a 512×512 grid. The length of the bit pattern in a node of a group was 8 bits. The number ($|Q|$) of continuous range queries inserted into the query index was 8000. The number ($|O|$) of objects was 64,000. The maximal horizontal or vertical movement (M) of objects between two consecutive reevaluations was 1, which is the same as the side length of a cell. We assume a query is represented in a rectangle and its side length is randomly generated from 0 to 80.

The size of a group impacts both the index storage cost and continuous query reevaluation time. The CES uses partitions as the maximal size of a virtual construct. The side length of groups ranges from 4 to 64. Figure 4 (a) shows the total storage costs according to the size of a group. As

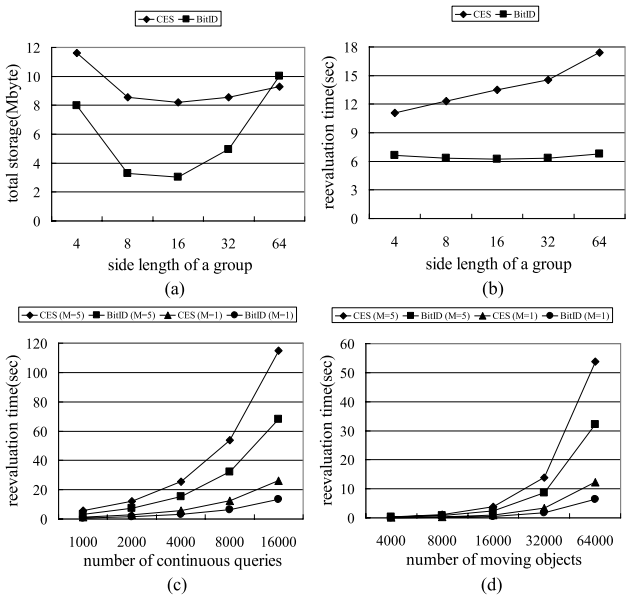


Fig. 4 The impact of the size of a group and the impact of $|Q|$ and $|O|$ on reevaluation time.

shown in Fig. 4 (a), the BitID generally uses smaller storage than the CES. Figure 4 (b) shows the query reevaluation time according to the size of a group. The reevaluation time is the total time for all range queries per each reevaluation. The performance advantage of the BitID over the CES is clearly observed. It is because the cost of bit comparison is much less expensive than the cost of the identifier comparison. In addition, the reevaluation time of the CES steadily increases, but BitID is not significantly impacted by the size of a group.

It shows the relationship between the size of a group and the total storage of the index. If the size of a group is too large, the number of groups is decreased and the redundancy of the identifiers is avoided, but the length of bit patterns in the groups is extraordinarily long. It causes the high storage cost, but it does not have a big impact on the computation cost since the comparison cost of bit patterns is very small. If the size of a group is too small, the length of a bit pattern in the group is very short, but the number of groups is increased and the identifiers of queries are redundantly stored in the groups. It also leads to the high storage cost, too. In addition, the increase of the computation cost occurs, according as the number of objects which move between groups is increased. There is trade-off between the redundancy of the identifiers and the length of the bit patterns for the storage cost. When the size of a group is too small or large, the storage cost of our method is higher than that of CES, so the appropriate size of a group is computed initially.

We compare BitID and CES under various numbers of continuous queries and moving objects. The default side length of the group is 16 because both CES and BitID produce the best performance when the side length of a group is 16. Both $M = 1$ and $M = 5$ are used. Figure 4 (c) shows the impact of $|Q|$ on reevaluation time. We varied $|Q|$ from 1,000 to 16,000. In all cases, the BitID achieves about 35% performance improvement over the CES in terms of reevaluation time. Figure 4 (d) shows the impact of $|O|$ on reevaluation time. We varied $|O|$ from 4,000 to 64,000. The performance improvement becomes more prominent as the number of moving objects increases.

4. Conclusion

We have proposed an efficient continuous range query processing and query indexing method. We used bit identifiers to represent queries and bit patterns to let cells represent the relationship between cells and queries. In addition, the management of cells in the grid by group prevents the waste of storage space through the increase of the length of the bit pattern and increasing the comparison costs of bit patterns.

Acknowledgments

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No.R01-2006-000-1080900) and the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (The Regional Research Universities Program/Chungbuk BIT Research-Oriented University Consortium)

References

[1] B.C. Ooi, K.L. Tan, and C. Yu, "Frequent update and efficient retrieval: An oxymoron on moving object indexes?," Proc. 3rd International Conference on Web Information Systems Engineering Workshops, pp.3–12, 2002.

[2] D.L. Lee, J. Xu, B. Zheng, and W.C. Lee, "Data management in location-dependent information services," IEEE Pervasive Computing, vol.1, no.3, pp.65–72, 2002.

[3] D.V. Kalashnikov, S. Prabhakar, and S.E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," Distributed and Parallel Databases, vol.15, no.2, pp.117–135, 2004.

[4] K.L. Wu, S.K. Chen, and P.S. Yu, "Incremental processing of continual range queries over moving objects," IEEE Trans. Knowl. Data Eng., vol.18, no.11, pp.1560–1575, 2006.

[5] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable incremental processing of continuous queries in spatio-temporal databases," Proc. ACM SIGMOD International Conference on Management of Data, pp.623–634, 2004.

[6] S. Prabhakar, Y. Xia, D.V. Kalashnikov, W.G. Aref, and S.E. Hambrusch, "Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects," IEEE Trans. Comput., vol.51, no.10, pp.1124–1140, 2002.