Continuous Density Queries for Moving Objects

Xing Hao
School of Information
Renmin University of China
haoxing@ruc.edu.cn

Xiaofeng Meng
School of Information
Renmin University of China
xfmeng@ruc.edu.cn

Jianliang Xu
Dept of Computer Science
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

ABSTRACT

Monitoring dense areas, where the density of moving objects is higher than the given threshold, has many applications like traffic control, bandwidth management, and collision probability evaluation. Although many studies have been done on density queries for moving objects in highly dynamic scenarios, they all focused on how to answer *snapshot* density queries. In this paper, we focus on *continuously* monitoring dense regions for moving objects. Based on the notion of *safe interval*, we propose effective algorithms to evaluate and keep track of dense regions. Experimental results show that our method can achieve high efficiency when monitoring dense regions for moving objects.

Keywords

Moving objects, continuous density queries, safe interval, Quad-tree

1. INTRODUCTION

Continuing advances in embedded systems, mobile communications, and positioning technologies have given rise to new applications like vehicle fleet tracking, watercraft and aircraft navigation, and emergency E911 services for cellular phone users. Such applications have triggered new research towards supporting location-based services in mobile environments. Current work in this area focuses mainly on modeling and indexing of moving objects, and optimizing spatio-temporal range and aggregation queries, k-nearest-neighbor queries, and selectivity estimation.

In this paper, we focus on density queries, which are important but have received attention only recently [2, 4, 7]. A region is *dense* if it has a high concentration of moving objects. Identifying dense regions is valuable for many applications like traffic control, resource scheduling, and collision probability evaluation [2]. For example, traffic congestion in large cities may be alleviated if traffic databases have been enhanced with the ability to *predict* dense regions that might be developed in the near future, and commuters could thus choose their routes to avoid jams.

Hadjieleftheriou *et al.* [2] introduced the problem of density queries, and presented several algorithms to evaluate such queries. Jensen *et al.* [4] have defined *effective density queries*. In both studies, a dense region is defined using the notion of *region density*: the density of a region in a 2-dimensional space is defined as the ratio of the number of moving objects in this region to its area. A density query returns the regions with a density higher than some user-specified threshold. Recently, Ni *et al.* [7] pointed out some drawbacks when answering density queries, such as ambiguity of answers, regions with no arbitrary shape or size, and no local density guarantees. The reason is that the previous work [2, 4] did not use an appropriate definition of dense regions. Ni *et al.* proposed a new definition of density, called *Pointwise-Dense Regions* (PDRs). Under this definition, they can answer density queries unambiguously and report all dense regions, regardless of shape and size. Moreover, PDR queries represent a more general class of density queries than *dense-cell queries* in [2] and the *effective density queries* in [4], since under proper conditions the answer to a PDR query includes the answer to those queries.

However, most of the prior work studied snapshot density queries only. In [4], for each cell, they maintain a histogram and record the number of objects in the cell. Once the objects update their locations, the histograms need to be updated. In this paper, we study continuous density queries for moving objects. In the beginning, we partition the space into a grid and compute initial dense regions. We then construct a Quad-tree upon grid cells. In the Quad-tree, we record the state of each cell, i.e., dense or sparse, based on which dense regions can be computed. Furthermore, we compute for each dense region the interval during which the region will keep the current state, which we call *safe interval* of the region. Comparing with maintaining histograms, the cost of maintaining safe intervals is expected to be lower. In [4], every cell has a histogram that records the object numbers at time instances $t_0, t_1, t_2, \cdots, t_m$, where t_0 is the query issuing time and $(t_m - t_0)$ is the duration of the query. In contrast, our method only maintains $(t_m - t_0)$ as a safe interval if the object numbers at those time instances are all above the dense threshold. Moreover, when an object updates, all histograms that the object corresponds to need to be re-calculated even if the update does not influence the state of the cell. In our approach, however, we only need to re-calculate the safe intervals of the regions that are influenced by the updated object.

MobiDE 08, June 13, 2008, Vancouver, Canada.

Our contributions in this paper can be summarized as follows:

- We propose an efficient method to monitor continuous density queries based on a notion of safe interval.
- We present a Quad-tree based scheme to maintain safe intervals for dense regions, which improves the efficiency of answering continuous density queries.
- Experimental results show that our method can achieve high efficiency when monitoring dense regions for moving objects.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 gives the definition and the algorithms of the continuous density queries. Experimental evaluation is presented in Section 4. Finally, Section 5 concludes the paper.

2. RELATED WORK

2.1 Snapshot Density Queries

Querying dense regions for moving objects was first investigated in [2]. The objective is to find the regions in space and time with a density higher than the given threshold. They found difficult to answer general density queries and hence simplified the definition of dense regions. Specifically, they partition the data space into disjoint cells, and density queries return cell-based regions, instead of arbitrary regions that satisfy the query conditions. This scheme may lead to loss of answers. To solve this problem, Jensen *et al.* [4] defined an effective density query (EDQ) to guarantee that there is no answer loss. Nevertheless, there might be multiple overlapping dense regions. Jensen *et al.* suggested reporting only a set of non-overlapping dense regions to an EDQ. However, this proposal still suffers from the drawbacks such as lack of unique answers and lack of local density. Ni *et al.* [7] argued that these drawbacks arise because of inappropriate definition of dense regions. They proposed the use of Pointwise-Dense Regions (PDRs), a new definition of dense regions, which helps avoid all the aforementioned problems. Nevertheless, the prior work all focused on *snapshot* density queries only, where the results are found based on a snapshot of the location dataset [1]. In contrast, in this paper we investigate continuous density queries for moving objects.

2.2 Continuous Spatio-Temporal Queries

Continuous queries are usually repeatedly evaluated with the available location information and their answers are changed with location updates of the moving objects [1]. In [11] the server returns a valid region of the answer, and in [12] the server returns a valid time. In general, these two approaches return the validity of the results. Assuming computational and storage capabilities at the client side, Xu *et al.* [10] proposed to cache the previous result in the client side with a validity mechanism. Previously cached results can be used to prune the search for new results of k-nearest-neighbor queries and range queries [5]. Precomputing the result was investigated in [9]. If the trajectory of the query movement is known previously, then by using computational geometry for stationary objects or velocity information for moving objects, the objects that will be nearest neighbors can be identified. If the trajectory information is changed, the query needs to be reevaluated. Monitoring continuous queries have been investigated in [6, 3]. Mokbel *et al.* [6] proposed evaluating the query incrementally. Instead of reevaluating the query and producing the whole query answer when the locations change, the query processor outputs positive and negative updates of previously reported answers. A positive update refers to an object being added to the query answer, whereas a negative update indicates an object being removed from the answer. Hu *et al.*[3] proposed a generic monitoring framework for range and kNN queries. The above studies are interested in proximate objects around query points, whereas density queries evaluate localized distributions of the objects. To the best of our knowledge, this is the first work to investigate continuous density queries. We provide a definition of continuous density queries for moving objects, which returns useful answers and is amenable to efficient computation. Furthermore, we propose the notion of *safe interval* for dense/spare regions to support efficient processing of continuous density queries.

3. CONTINUOUS DENSITY QUERIES

We assume that a collection of objects are moving on the space under consideration, where each object is capable of transmitting its location and velocity to the central server. The central server can predict the object positions based on the location and velocity information, and continuously answer density queries. When an object changes its velocity, it updates the new velocity to the central server.

Definition 1 (*Continuous Density Query*): A continuous density query returns all the regions that satisfy the following three conditions:

1. The density of the region is no less than ρ;
2. The minimum area of our interest is s and any subarea of the region with an area larger than s must be dense;
3. No two regions in the result set overlap with each other.

Conditions 1) and 2) indicate that each dense region must have more than $\rho \cdot s$ objects. Condition 3) is provided to simplify the search of dense regions, as did in the previous work.

We use the TPR-tree to index the moving objects [8]. In the TPR-tree, the position of a moving object is represented by a vector including the reference position and the velocity — $(p(t_{ref}), v)$. We can predict the future location at time t using the following formula:

$$p(t) = p(t_{ref}) + v \cdot (t - t_{ref}).$$

In order to find local dense regions, we recursively partition the space by a Quad-tree. The Quad-tree is used to store the state (i.e., dense or sparse) of a subspace, as well as the validity in time which we call safe interval of the subspace. Thus, a node in the Quad-tree is represented as ((*row*, *col*), *level*, *state*, *safe_interval*), where (*row*, *col*) is an index to identify the node, and *level* denotes the level of the tree that the node belongs to. If the node is a leaf, the *state* can be 0 or 1, which indicates the region represented by the node is sparse or dense. For a non-leaf node, the *state* can be 0, 1, or 2, where 0 indicates all its children nodes are sparse, 1 indicates all its children nodes are dense, and otherwise the

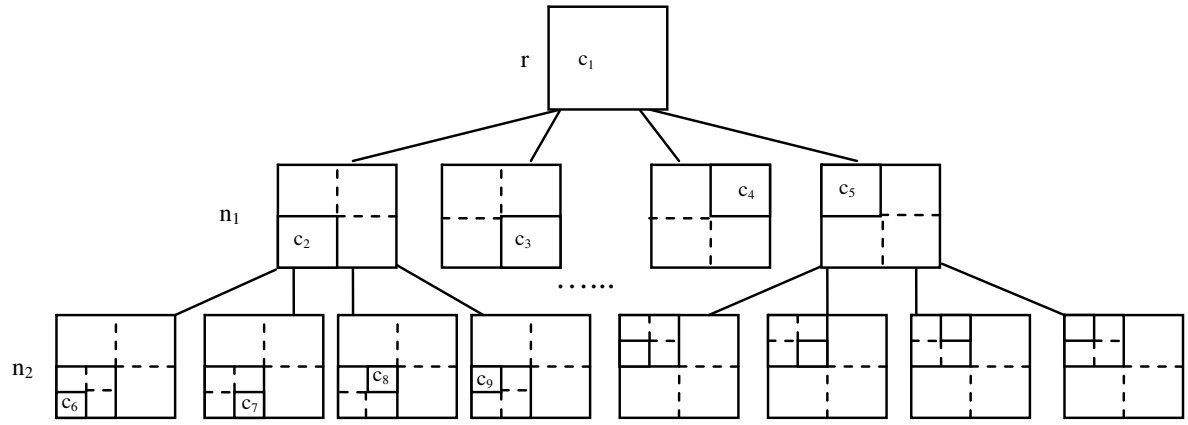


Figure 1: An example of Quad-tree.

state is 2. The *safe_interval* is the valid time of the state, which is formally defined as follows:

Definition 2 (*Safe Interval*): The *safe interval* is the time period for which the region keeps its current state. For example, if the region is dense, it will remain dense for at least a time period of *safe interval*. After that, the state of the region may or may not change.

Next we will proceed to discuss how to build a Quad-tree and compute the safe intervals, followed by how to answer continuous density queries using the Quad-tree.

3.1 Building the Quad-Tree

As aforementioned, to facilitate searching dense regions, we partition the space into a grid by employing a Quad-tree. More specifically, the space is recursively divided into four quadrants until the area of the subspace is less than the threshold s given in the density query definition. We set s as the stop condition since it is the minimum area we should consider for a dense region according to the definition. Given a space with an area of S, the depth of the Quad-tree is:

$$L = \lceil \log_4 \frac{S}{s} \rceil + 1. \quad (1)$$

In the Quad-tree, each node corresponds to a cell in the grid. Recall that a node is represented by ((*row*, *col*), *level*, *state*, *safe_interval*). The cell can be easily determined by some of these parameters. More specifically, the left-bottom point of the cell is given by:

$$\frac{\sqrt{S}}{2^{level}} \times [row - 1, col - 1]$$

The right-upper point of the cell is given by:

$$\frac{\sqrt{S}}{2^{level}} \times [row, col]$$

Figure 1 shows an example of the Quad-tree. Given S=32, s=2, and ρ=1.5, based on Equation (1), the level number of the Quad-tree is 3. The root of the Quad-tree corresponds to the largest cell c_1. Its level number is 0, the row value is 1, and the col value is also 1. Each internal node is one quadrant of the root, including c_2, c_3, c_4, and c_5. The leaf nodes correspond to the minimum cells (called *leaf cells* hereafter), such as c_6, c_7, c_8, and c_9.

Based on the Quad-tree, initially we count the number of moving objects for each leaf cell and determine if the cell is dense or spare. By definition, a high-level cell is dense if and only if all the leaf cells below it are dense. For example, in Figure 1, if c_6 through c_9 are dense while some other leaf cell is sparse, then c_2 is returned as a dense region but c_1 is not.

3.2 Safe Interval Computation

A safe interval of a dense (sparse) cell means the minimum time period for which the cell is still dense (sparse). Due to the movement of objects, a dense cell may turn into a sparse one, and vice versa. Thus, to support continuous density queries, we maintain the safe intervals for leaf cells of both types, but the safe intervals for high-level cells only if they are dense (i.e., only for dense regions). In the following, we discuss how to compute the safe intervals for dense and sparse leaf cells. The safe interval of a dense high-level cell can be recursively set as the smallest one of its child nodes.

3.2.1 Safe Interval of Dense Leaf Cell

For a dense leaf cell, to simplify the computation, we only focus on the objects leaving from it, without considering the entering objects. This is because an entering object will not change the state of a dense cell. It can only change the state of a sparse cell, that is, makes the sparse cell to be dense. Thus, we compute the shortest time interval for which the cell remains dense.

Figure 2 gives an example, where cell C is dense. There are totally five objects in C, i.e., o_1, o_2, o_3, o_4, and o_5. Let the object number threshold for a dense cell to be three. We compute the time before each object will leave this cell to obtain the *safe interval* of the dense cell. Suppose the leaving times of these objects are t_5, t_3, t_1, t_4, and t_2, sorted in an ascending order. Then t_1 is the *safe interval* of the dense cell since this cell may become sparse after o_1 leaves.

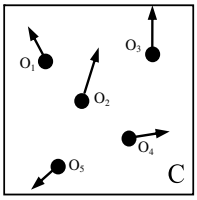


Figure 2: An example of dense region.

Algorithm 1 formally describes how to compute the safe interval for a dense leaf cell *cell*, where $(xmin, ymin)$ and $(xmax, ymax)$ are the bounding coordinates of *cell*, (x, y) is the coordinate of *obj* at time t, and (vx, vy) is the object's speed in the x and y dimensions. We use a heap H to store the last several objects leaving from the cell. Let S_{cell} be the area of the cell. The size of H is set to $\rho \cdot S_{cell}$, which is the density threshold of the cell in terms of the number of objects. For every object in the cell, we compute its leaving time and push the time into H. After processing all the objects, when the object who has the minimum leaving time in H leaves from the cell, the object number in the cell will be fewer than the density threshold if not considering the objects potentially entering from the outside. Hence, the minimum value in H is the earliest possible time that the cell changes its state. This value is returned as the safe interval of *cell*.

Algorithm 1 SIofDense(*cell*)

```
1: H is a min-heap, whose size is ρ · S_cell
2: for every obj in cell do
3:     if (obj.vx>0) then lx=cell.xmax-obj.x
4:     else if (obj.vx<0) then lx=cell.xmin-obj.x
5:     else
6:         lx=cell.xmax-cell.xmin
7:     end if
8:     if (obj.vy>0) then ly=cell.ymax-obj.y
9:     else if (obj.vy<0) then ly=cell.ymin-obj.y
10:    else
11:        ly=cell.ymax-cell.ymin
12:    end if
13:    Push min(lx/vx, ly/vy) into H
14: end for
15: Return the minimum value in H
```

Note that the safe interval of a dense leaf cell we compute is the shortest time interval for which the dense state remains. Hence, when the safe interval expires, the state of the cell may not be changed if there have been some other objects entering into this cell. Thus, the state of this cell and the corresponding safe interval need to be re-calculated upon expiration.

3.2.2 Safe Interval of Sparse Leaf Cell

Similar to the dense leaf cell, we only focus on the entering objects for sparse cell, without considering the leaving objects. Suppose that N is the density threshold for the sparse cell, and that presently there are M objects in the cell. Then after (N - M) objects move into this cell, its state might be changed. To reduce the cost of scanning outside objects, we expand the cell level by level until the expanding region contains (N - M) objects. When all the objects in this expanding region enter into the cell, the cell's state may be changed. On the other hand, a fasting moving object outside this expanding region may have also entered into the cell. Such earliest time is given by

$$t_o = \frac{L}{V_{max}}, \tag{2}$$

where V_{max} is the known maximum moving speed and L is the length of the expanding distance. Thus, within the interval t_o, we only need to scan the objects in the expanding region and estimate whether these objects can change the state of this sparse cell by computing their entering times.

Algorithm 2 describes how to compute the safe interval for a spare leaf cell *cell*. Again we use a heap H to store the first several objects that will enter into *cell*. The size of H is (N - M). The cell is expanded to a larger region denoted as $Cell$ which includes at least $\rho \cdot S_{cell}$ objects. We then compute the entering times of these additional objects in $Cell$. If object i's entering time, denoted by t_i, is longer than t_o, given in Equation (2), t_i is set to t_o. After processing all the additional objects in $Cell$, the maximum value in H is returned as the safe interval of *cell*.

Algorithm 2 SIofSparse(*cell*)

```
1: H is a max-heap, whose size is (ρ · S_cell)-(number of objects in cell)
2: Expand cell to Cell, which includes at least (ρ · S_cell) objects. L is the expanded distance and V_max is the maximum velocity of all the objects
3: for every additional object obj in Cell do
4:     if (obj.vx>0 and obj.x<=cell.xmin) then
5:         lx=cell.xmin-obj.x
6:     else if (obj.vx<0 and obj.x>=cell.xmax) then
7:         lx=cell.xmax-obj.x
8:     else
9:         lx=L
10:    end if
11:    if (obj.vy>0 and obj.y<=cell.ymin) then
12:        ly=cell.ymin-obj.y
13:    else if (obj.vy<0 and obj.y>=cell.ymax) then
14:        ly=cell.ymax-obj.y
15:    else
16:        ly=L
17:    end if
18:    t = lx/vx
19:    if (obj is not in cell at time t) then t=ly/vy
20:    end if
21:    if (t > L/V_max) then t = L/V_max
22:    end if
23:    Push t into H
24: end for
25: Return the maximum value in H
```

Figure 3 shows an example, where C is a sparse region. In the expanding region, the objects o_1, o_2, o_3, o_4, and o_5 are moving towards C. Suppose that their entering times are t_2, t_1, t_5, t_3, t_4, sorted in descending order, and that they are all smaller than t_o. If the region would change to a dense one after three objects move into it, we will then use t_5 as its safe interval.

Similar to the case for a dense leaf cell, the state and the safe interval of a sparse cell have to be re-computed when the safe interval expires.

3.3 Update of Objects

There are two cases in which we need to update the safe interval of a dense/sparse leaf cell: (i) when the safe interval expires, we need to recompute the state and safe interval of the cell, as discussed in last two subsections; (ii) when the velocity of the object updates, we need to recompute the states and safe intervals of those cells affected by this update. Below we discuss how to deal with the second case.

When the updating object is in a sparse cell, we do not

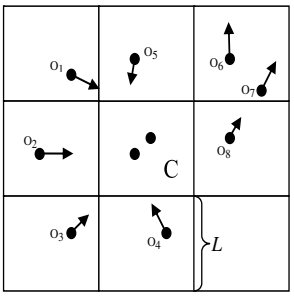


Figure 3: An example of sparse region.

need to recompute the safe interval of this cell since we consider only the entering objects from the outside. However, the object may affect the safe intervals of other sparse cells which the object's moving trajectories cross. We remark that we only need to recompute the sparse cells which the object's new trajectory crosses. For those sparse cells intersected with the old trajectory, we do not need to recompute their safe intervals until they expire, because before the current safe intervals their states would remain unchanged.

When the updating object is in a dense cell, the safe interval of this cell may be changed because we compute the safe interval for a dense cell based on the objects inside the cell. The sparse cells which intersect with the object's new trajectory also need to be recomputed .

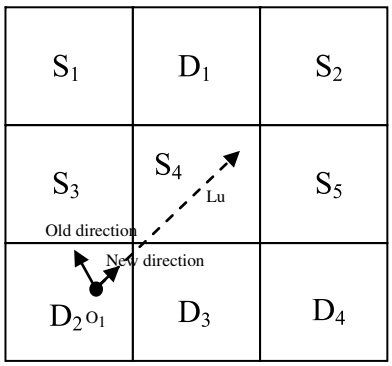


Figure 4: An example of object updating.

Figure 4 shows an example for how to find the sparse cells whose safe intervals need to be recomputed, where S_1, S_2, S_3, S_4, S_5 are sparse cells, D_1, D_2, D_3, D_4 are dense cells, and o_1 is an updating object with its velocity changed. We need not to consider the sparse cells in its old moving direction, i.e., S_3. In the new moving direction, we identify the sparse cells that o_1 may affect its safe interval. In order to reduce the computing cost, the formula

$$L_u = v_1 \cdot SI_{max} \tag{3}$$

can be used to determine the length of the trajectory, where v_1 is the new speed of o_1 and SI_{max} is the maximum safe interval among all cells. We only update the safe intervals of the sparse cells that intersect with the segment L_u (e.g., S_4 in Figure 4).

3.4 Query Processing

Having computed the states and safe intervals for all leaf cells, we are ready to find dense regions. We search the Quad-tree in a bottom-up manner. For an intermediate node, if all its child nodes are dense (i.e., with the state value of 1), this node is also dense, otherwise it is not by definition. The bottom-up search of a dense region stops until an ancestor is not dense. Then its child nodes that are dense are returned as answers. The safe interval of the dense region is set as the smallest interval of the leaf cells contained in the dense region. When the safe interval expires, this means the safe interval of a leaf cell expires. The state and safe interval of that leaf cell will be updated, based on which the dense region is also reevaluated. The formal procedure is described in Algorithm 3.

Algorithm 3 Query()

```
1: for every leaf node n do
2:     if (n.safe_interval>=query time) then
3:         if (n.state == 0) then break
4:         else
5:             n'=n
6:             while (n'.parent.state == 1 and
   n'.parent.safe_interval>=query time) do
7:                 n'=n'.parent
8:             end while
9:             output n'
10:            ignore the children of n' and get the next leaf
   node
11:        end if
12:    else
13:        count number of moving objects in n
14:        if (number>=ρ · S_cell) then SIofDense(n)
15:        else
16:            SIofSparse(n)
17:        end if
18:        if (n.state changes) then adjust value of
   n.parent and take n as the next node
19:        end if
20:    end if
21: end for
```

4. PERFORMANCE EVALUATION

4.1 Experimental Settings

This section experimentally evaluates the efficiency of our proposed Quad-tree based algorithm. The Snapshot algorithm, with repeated execution, is included for comparison. All the experiments were run on a 3.20G Pentium (R) desktop with 512MB of memory.

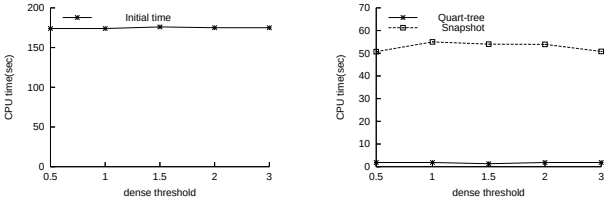
We assume a 100×100 space for the moving objects to move around, following a random movement model. The experiments study the impact of a variety of system factors, including the density threshold ρ, the minimum region area s, and the number of moving objects #mo. The parameters used in our experiments are reported in Table 1, where the values in bold denote the default settings. In each experiment, only one parameter varies while the others are fixed at their default values, and 100 continuous density queries with random durations are tested. The results reported below represent the average cost per query.

Table 1: Parameters used in experiments

Parameter	Values
dense threshold ρ	0.5,1,1.5,**2**,3
minimum region area s	15*15, 10*10, **5*5**, 2*2
number of moving objects #mo	1k, 5k, **10k**, 20k

4.2 Results

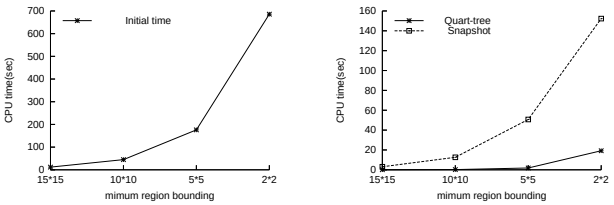
The first set of experiments studies the impact of ρ on the efficiency of the two algorithms under comparison. Figure 5 shows the CPU time as a function of ρ. Figure 5(a) plots the initialization time for the Quad-tree algorithm, i.e., the Quad-tree building time. Figure 5(b) compares the query answering time for the two algorithms. Both algorithms are not much influenced by ρ, this is because both of them have to search the whole object space, regardless of the value of ρ. Nevertheless, clearly the Quad-tree algorithm is more efficient than the Snapshot algorithm in terms of the query answering time.



(a) initialization time (b) answer time

Figure 5: CPU time vs. density threshold ρ.

In Figure 6, we show the effect of the minimum region area s. From Figure 6(b), both algorithms get a longer query answering time with decreasing s. This is because the whole area is divided into more regions for a smaller value of s. Thus, we have to compute the results for more cells. That is the same reason why the initialization time grows (see Figure 6(a)). More nodes and safe intervals have to be computed in this case. In terms of the query answering time, we can see the Quad-tree algorithm is more efficient than the Snapshot algorithm, and their performance gap enlarges for a smaller value of s.

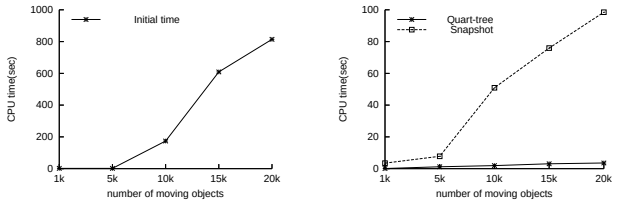


(a) initialization time (b) answer time

Figure 6: CPU time vs. minimum region area s.

Next, we examine the impact of the number of moving objects #mo. As shown in Figure 7, when the number of moving objects grows, the performance degrades for both algorithms as expected. Nevertheless, the Quad-tree algorithm is still much more efficient than the Snapshot algorithm.

To summarize, the Quad-tree algorithm does not need to



(a) initial time (b) answer time

Figure 7: CPU time vs. number of moving objects #mo.

recompute the result each time a location update occurs. As such, it outperforms the Snapshot algorithm in all cases tested, although it needs some time to initialize the Quad-tree. The good thing is that the initialization time is not too long, which should be acceptable to the queries.

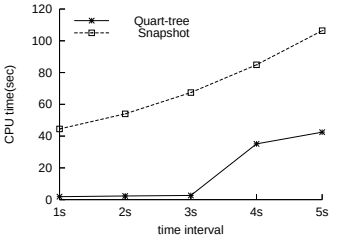


Figure 8: CPU time vs. query interval

Finally, Figure 8 shows the influence of the query interval (i.e., the time interval between two continuous queries) on the CPU time. From the result, we can see that the performance gap of the two algorithms decreases when the query interval becomes larger. For the Snapshot algorithm, because of the TPR-tree structure, more objects have to be visited when computing a cell's state. And for the Quad-tree algorithm, the CPU time increases because of more safe intervals expired and recomputed.

5. CONCLUSIONS

In this paper, we investigated the problem of monitoring continuous density queries for moving objects. We have proposed the notion of safe interval, and introduced a Quad-tree based scheme to evaluate and keep track of dense regions. Experimental results demonstrate that our method can achieve high efficiency when monitoring dense regions for moving objects.

6. ACKNOWLEDGMENTS

This research was partially supported by Natural Science Foundation of China under grant no. 60573091, China 863 High-Tech Program under project no. 2007AA01Z155, and Program for New Century Excellent Talents in University (NCET). Jianliang Xu's work was supported in part by the Research Grants Council, Hong Kong SAR, China (Project Nos. HKBU211206 and HKBU211307).

7. REFERENCES

[1] H. G. Elmongui, M. Ouzzani, and W. G. Aref. Challenges in spatiotemporal stream query optimization. In *MobiDE*, pages 27–34, June 2006.

[2] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *SSTD*, pages 306–324, July 2003.

[3] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, June 2005.

[4] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective density queries on continuously moving objects. In *ICDE*, page 71, April 2006.

[5] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, pages 269–286, March 2002.

[6] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, June 2004.

[7] J. Ni and C. V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *ICDE*, pages 1066–1075, April 2007.

[8] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A.Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, May 2000.

[9] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, August 2002.

[10] J. Xu, X. Tang, and D. L. Lee. Performance analysis of location-dependent cache invalidation schemes for mobile environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(2):474–488, March/April 2003.

[11] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, pages 443–454, June 2003.

[12] B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *SSTD*, pages 97–116, July 2001.