Effective Density Queries on Continuously Moving Objects

Christian S. Jensen[1] Dan Lin[2] Beng Chin Ooi[2] Rui Zhang[2]

[1]Department of Computer Science
Aalborg University, Denmark
csj@cs.aau.dk

[2]School of Computing
National University of Singapore, Singapore
{*lindan, ooibc, zhangru1*}*@comp.nus.edu.sg*

Abstract

This paper assumes a setting where a population of objects move continuously in the Euclidean plane. The position of each object, modeled as a linear function from time to points, is assumed known. In this setting, the paper studies the querying for dense regions. In particular, the paper defines a particular type of density query with desirable properties and then proceeds to propose an algorithm for the efficient computation of density queries. While the algorithm may exploit any existing index for the current and near-future positions of moving objects, the B^x-tree is used. The paper reports on an extensive empirical study, which elicits the performance properties of the algorithm.

1 Introduction

1.1 Motivation and Background

Continuing advances in consumer electronics, mobile communications, and positioning technologies combine to render it increasingly realistic to assume that entire populations of users of mobile services, termed moving objects, can be tracked accurately. These developments offer a foundation for the delivery of increasingly sophisticated location-enabled mobile services. Motivated by this scenario, one line of research aims to provide appropriate data management foundations that enable the provisioning of efficient services. Proposals exist for the efficient computation of, e.g., window queries and nearest neighbor queries on moving objects (e.g., [7, 8]).

In this paper, we study the querying for dense regions, regions with a high concentration of moving objects. The objective is to find regions in space along with associated points in time where the regions have a density that exceeds a given threshold. Figure 1 illustrates an example where three square-shaped windows compose the answer to a density query. The density query may have applications

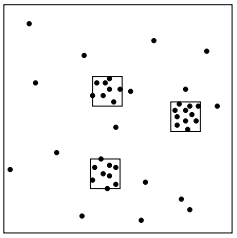


Figure 1. An example of density query results

in a range of areas. In traffic management systems, density queries may be used for identifying regions with potential for congestion and traffic jams.

Density querying for moving objects was first considered by Hadjieleftheriou et al. [3]. They define the *Region Density* as: $density(R, \Delta t) = \min_{\Delta t} N/\text{area}(R)$, where $\min_{\Delta t} N$ is the minimum number of objects inside R at any time during Δt and $\text{area}(R)$ is the area of R. They define the *Period Density Query* as: given N moving objects, a horizon H, and thresholds α_1, α_2, and ρ, find regions $R = \{r_1, ..., r_k\}$ and associated maximal time intervals $\Delta t = \{\delta t_1, ..., \delta t_k | \delta t_i \subset [t_{now}, t_{now} + H]\}$ such that $\alpha_1 \leq \text{area}(r_i) \leq \alpha_2$ and $density(r_i, \delta t_i) > \rho$ (where t_{now} is the current time, $i \in [1, k]$, and k is the query answer cardinality).

For the applications we envision, finding dense regions *for a period of time* appears to be less useful than simply finding dense regions for a point in time. For example, once a traffic jam occurs in some region, all the objects around the region will slow down, and their velocities will change dramatically. As a result, predicting the density at following timestamps according to the original velocities reported for the objects does not seem to be of much value. Therefore, we focus on the identification of dense regions as of a time $t_q \in [t_{now}, t_{now} + H]$ that is given as a parameter to the density query.

Hadjieleftheriou et al. [3] find the general density-based queries difficult to answer efficiently and hence turn to sim-

plified queries. Specifically, they partition the data space into disjoint cells, and the simplified density query report cells, instead of arbitrary regions, that satisfy the query conditions. This scheme may result in what we term *answer loss*. Consider the example shown in Figure 2 where each cell (of solid lines) is a unit square and the density threshold ρ is 3. There actually exists a dense region (the dashed square) in the center of the space, but the simplified query reports no regions. In our density query definition, we guarantee that there is no answer loss.

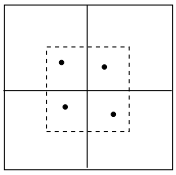


Figure 2. An example of answer loss

We proceed to formulate the problem setting and define the notions of *density*, *dense region*, and *effective density query*.

1.2 Problem Statement

We assume that a population of moving objects exists, where each object is capable of transmitting its current location to a central server. A moving object transmits a new location to the server when the deviation between its real location and its server-side location exceeds a threshold, dictated by the services to be supported. In general, the deviation between the real location and the location assumed by the server tends to increase as time passes. In keeping with this, we define a *maximum update time* (U) as a problem parameter. This quantity denotes the maximum time duration in-between two updates of the position of any moving object.

We model the position of a moving object as a linear function from time points to points in two-dimensional Euclidean space. The position of a moving object at time t, $\bar{x}(t)$, is thus given by a triple $(\bar{x}, \bar{v}, t_{upd})$ of parameters as follows: $\bar{x}(t) = \bar{x} + \bar{v}(t - t_{upd})$, where $\bar{x}$ and $\bar{v}$ are the two-dimensional position and velocity, respectively, of the object at the latest update time t_{upd} and $t \geq t_{upd}$.

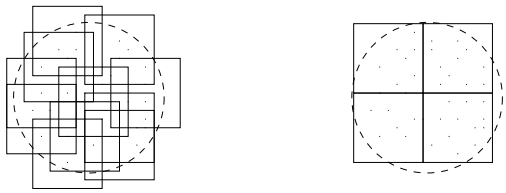
The motivation for this choice of modeling is threefold. First, the extent of a moving object is typically considered to be of little relevance for the query type we are considering. Second, studies of real positional information obtained from GPS receivers installed in cars show that representing positions as linear functions of time reduces the numbers of updates needed to maintain a reasonable accuracy by as much as a factor of three in comparison to using constant functions [2]. Linear functions are thus much better than constant functions. Third, several indexing techniques exist that index this representation and which can be reused.

With this general data setting in place, we proceed to define the density query.

Definition 1 (**Density**)*: The density of a region R at a time t is the number of objects in the region at time t divided by the area of the region.*

Definition 2 (**Dense Region**)*: A region is dense at time t if its density at time t is higher than a density threshold ρ.*

Note that a part of space that contains one dense region is likely to contain many such regions. Most of these may overlap substantially, as illustrated in Figure 3(a). Reporting all such regions is not helpful. We proceed to propose an *effective density query* that only reports non-overlapping regions. The resulting answer set clearly identifies regions of high density, as shown in Figure 3(b).



(a) **(b)**

Figure 3. Overlapping versus non-overlapping regions in a density query

Definition 3 (**Effective Density Query**)*: Find all dense regions at time t that satisfy the following conditions:*

1. *Any reported region is constrained to a certain shape and an area range.*
2. *No two regions in the result overlap.*
3. *Any dense region in the argument data is in the result, or is represented in the result by a region that overlaps with it.*

The first condition provides mechanisms for ensuring that meaningful answers are reported. For example, an arbitrarily small region that contains one point object is infinitely dense, but makes little sense as a result. Therefore, we may want to specify a lower bound of the area of the result regions. Similarly, we may want to limit the region area so that it is not too large. Also not all shapes of result regions may be desirable. Imagine a (space filling) curve that goes through all the point objects. Therefore the user may require the dense regions to be squares, circles, etc. The second condition guarantees that a non-redundant result is produced. The third condition guarantees that the query result contains

evidence of any dense region in the data, ensuring that query results do not suffer from answer loss.

We purposefully define the effective density query so that different, but equally valid and useful, results may be produced for the same data argument and query parameters. An algorithm implementing the query may exploit this flexibility. For convenience, we abbreviate the term *effective density query* to *density query* in the sequel.

We assume that the time parameter t_q of a density query Q is not earlier than the current time and that it only reaches at most W time units into the future. Thus, with $iss(Q)$ being the time that query Q is issued, $iss(Q) \leq t_q \leq iss(Q) + W$. The lower bound indicates that we are not considering past data. The assumption that an upper bound exists is considered reasonable. Updates are inherently frequent, making it meaningless to look too far into the future.

Finally, define *time horizon* $H = U+W$ as the maximum duration of time that the representation of a moving object can be queried. Figure 4 illustrates the relationships among parameters U, W and H. We can see that H represents how far into the future a query may reach. In other words, a technique for computing the density query support queries that reach up to H time units from the update of an object into the future.

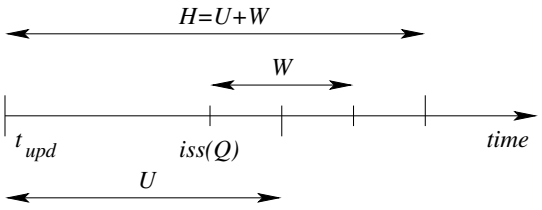


Figure 4. Problem parameters

1.3 Contributions

This paper provides a definition of the density query for moving objects that avoids answer loss. Based on this definition, the paper proposes a specialization of the density query that returns useful answers and is amenable to efficient computation.

The paper then proposes an algorithm that aims to process the resulting density query efficiently. The algorithm utilizes temporal histograms of counters for each partition in a partitioning of the data space. These histograms help prune the majority of regions in an initial filtering phase of the query processing. As the histograms consume substantial storage space, which in turn increases I/O, techniques are proposed that use the Discrete Cosine transform (DCT) to compress the histograms. This compression incurs very few errors in the answer set, but offers space savings of up to 90%, which also reduces I/O.

The paper also reports on extensive empirical studies of the behavior of the proposed algorithm. The results suggest that the algorithm offer an improvement of a factor of 4 in terms of I/O, compared to a naive algorithm. The results also indicate that although we reduce the storage usage greatly by using DCT, the answers are still highly accurate. It is shown that it is easy to trade a small number of false negative answers for performance.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the framework. Section 4 presents our algorithm used to realize the framework. Section 5 reports the experimental results. Section 6 concludes the paper.

2 Related Work

Any index for moving objects that supports predictive window queries is able to answer density queries by using our framework as presented

As will become clear in Section 3 any index for moving objects that supports predictive window queries may be used by our proposal for computing the density query. Representative indexes include the TPR-tree (Time-Parameterized R-tree) family of indexes (e.g., [14, 15]) and transformation-based indices such as STRIPES [12], which supports efficient updates and queries at the expense of higher space requirements, and the B^x-tree [4], which uses the B$^+$-tree to manage moving objects efficiently. We employ the B^x-tree as our underlying index structure.

Several proposals [9, 10, 11] exist for the computation of spatio-temporal aggregation queries, which are similar to density queries in some sense, since density queries also need to know the numbers of objects inside certain ranges. However, a key difference is that the query ranges are given for aggregation queries, while density queries must locate ranges that satisfy the density threshold.

Existing clustering algorithms can represent the most dense areas by, e.g., the centers of the clusters. As good examples, Yiu and Mamoulis [16] cluster objects at a certain timestamp; and Li et al. [6] cluster moving objects, but at the expense of high maintenance costs. These techniques do not meet the requirements posed by density queries. They are unable to identify those regions with a density higher than the specified threshold that are not given by the cluster centers, or they are not effective in tracking the continuously changing positions of moving objects.

The most closely related work is by Hadjieleftheriou et al. [3]. As mentioned in Section 1.1, their definition permits query results not to include evidence of all dense regions. They propose three algorithms: (a) coarse grid, (b) lossy counting, and (c) dense cell filter. However, none of these are applicable to our definition due to the answer-loss problem.

3 The MODQ Framework

As a precursor to considering the processing of density queries, we constrain the general setting for density queries as presented in Section 1.2 with the objective of rendering the query processing manageable. We term this setting the Moving Object Density Query (MODQ) framework.

In this framework, we constrain the dense regions to be square-shaped and of certain sizes. We maintain all moving objects in an index structure. During query processing, we first partition the data space into equal-sized, square-shaped cells of the smallest size that satisfies the query constraint. Then we issue window queries that explore these cells, and we report the dense regions found.

Note that a dense region is not necessarily a cell in the partitioning—it might instead intersect with cell partitions, as does the dashed, square region in Figure 2. Therefore we may need to issue a window query that is defined by two or more adjacent cells. By exploring all cells (and maybe all combinations of two or more adjacent cells), we are able to report evidence of all dense regions.

While the framework applies to dense regions of a range of sizes, we focus on finding dense regions of one size in this paper.

4 Density Computation

4.1 Overview

If we process the density query straightforwardly using window queries on the index, we are likely to end up issuing too many window queries. Instead, we propose a two-phase algorithm that efficiently computes the density query as state in the previous section.

The algorithm relies on certain information that needs to be maintained. We maintain a counter that records the number of objects for each cell at point in time. As each object is updated, we calculate the trajectory of the object and obtain the cells it intersects during the query time window $[t_{now}, t_{now} + H]$. For each $(\text{cell}, \text{time})$ pair of a cell intersected and the time of intersection, we increase the corresponding counter by one (in case of deletion, we decrease the counter).

At the same time, the object is maintained and updated in an index structure. This index may well be maintained already for other types of queries, such as window and nearest neighbor queries, so the major space overhead is that of the space for maintaining the counters. When the numbers of cells and time windows are large, the number of counters needed is huge. We describe a method to efficiently maintain them in a compressed fashion in Section 4.2.

Next, query processing consists of two phases:

1. **The filtering phase:** We use the counters to quickly prune the cells that are surely not in the answer set and produce a set of candidate cells for the next phase.
2. **The refinement phase:** To extract the final answers from the candidate cells obtained in the filtering phase, we issue window queries corresponding to the cells on the index and this way determines the actual positions of the objects in the cells. Then we can determine the dense regions. Without loss of generality, we exploit the B^x-tree [4] to maintain the moving objects.

The query processing algorithm are given in Section 4.3.

4.2 Density Histogram

We maintain a two-dimensional *density histogram* (DH) equal sized, square cells, where each cell contains a counter of the number of objects in the cell at all times in $[t_{now}, t_{now} + H]$. The DH is the main structure used for the filtering phase. As we need to maintain a histogram for a long time period for each cell, the total memory use may b prohibitively large. Moreover, if the DH is stored on disk, substantial I/O is needed to maintain it and use it during query processing. Therefore, it is critical to reduce the size of the DH. We propose to use the Discrete Cosine Transform (DCT) [13] for compression. The detailed algorithm and analysis are as follows.

4.2.1 Histogram Construction

For each cell, the number of objects in the cell varies across time, but the number is not likely to change greatly for adjacent time points. We thus view the time-varying count in each cell for a time range as a signal $s(t)$, and we then perform the Discrete Cosine Transform (DCT) on $s(t)$. As the DCT is a good approximation of the Karhunen-Loève Transform (KLT) [13], the first few components of the DCT of s carry the major information in s. We store only the first few (typically 10–20%) components and discard the rest. We can restore the $s(t)$ by inverse DCT when we maintain the histograms or use them for query processing.

The DCT of a signal $s(t)$ of length H is also a signal $G(k)$, of length H. The transform is defined as follows:

$$G(k) = c(k) \sum_{t=0}^{H-1} s(t) \cos \frac{\pi(2t+1)k}{2H} \quad \text{where} \quad (1)$$

$c(0) = \sqrt{1/H}, \quad c(k) = \sqrt{2/H}, \text{ and } k = 0, 1, ..., (H-1)$

The inverse DCT is defined as follows:

$$s(t) = \sum_{k=0}^{H-1} c(k) G(k) \cos \frac{\pi(2t+1)k}{2H}, t = 0, 1, ..., (H-1) \quad (2)$$

Figure 5 shows an example of transform between the signal and the DCT.

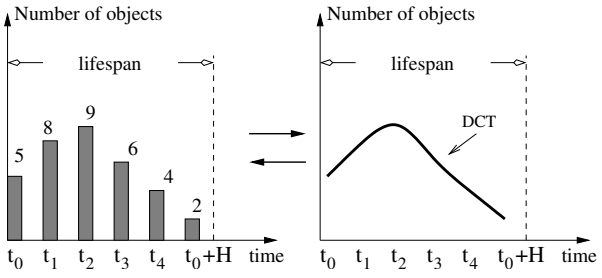


Figure 5. An example of the DCT

After trimming some components of the DCT, the restored signal $s'(t)$ differs from $s(t)$ because of the information loss. Therefore, the results of the query is approximate. However, the DCT has the good property that even though we trim off a great portion of the components, the difference between $s'(t)$ and $s(t)$ is still quite small.

Note that the difference between $s'(t)$ and $s(t)$ may be positive or negative, that is, $s'(t)$ may overestimate or underestimate $s(t)$. As a result, we may get both false positives and false negatives when we choose candidate cells for further examination in the refinement phase. False positive candidate cells increase the query processing cost, while false negatives would cause answer loss (although the loss could be very small). To avoid false negatives, we may add something to $s'(t)$ so that $s'(t)$ is guaranteed to overestimate $s(t)$. We derive the bound of $s(t) - s'(t)$ next. We assume that we use the first g coefficients of the DCT to compress and trim the remaining ones.

$$\begin{aligned} s(t) - s'(t) &= \sum_{k=g}^{H-1} c(k)G(k)\cos\frac{\pi(2t+1)k}{2H} \\ &= \sqrt{2/H}\sum_{k=g}^{H-1} G(k)\cos\frac{\pi(2t+1)k}{2H} \\ &\le \sqrt{2/H}\sum_{k=g}^{H-1} |\max_{k=g}^{H-1}\{G(k)\}\cos\frac{\pi(2t+1)k}{2H}| \\ &\le \sqrt{2/H}\max_{k=g}^{H-1}\{|G(k)|\}\sum_{k=g}^{H-1} |\cos\frac{\pi(2t+1)k}{2H}| \quad (3) \end{aligned}$$

where $t = 0, 1, ..., (H-1)$ and $\max_{k=g}^{H-1}\{|G(k)|\}$ denotes the maximum absolute value of $G(k)$ for $k = g, ..., H-1$. Therefore if we estimate $s(t)$ by $s'(t)$, the *error bound* E_b is given as follows.

$$E_b = \sqrt{2/H}\max_{k=g}^{H-1}\{|G(k)|\}\sum_{k=g}^{H-1} |\cos\frac{\pi(2t+1)k}{2H}| \quad (4)$$

Before trimming the coefficients from the DCT of $s(t)$, we get $\max_{k=g}^{H-1}\{|G(k)|$ and store it with the g remaining coefficients; and $\sum_{k=g}^{H-1} |\cos(\pi(2t+1)k/2H)|$ can be calculated on the fly, giving us the error bound. To guarantee no false negatives, we just need to add the error bound to $s'(t)$. However, this increases the number of false positives and hence increases the query processing cost. In some applications, we may be willing to trade a small number of false negatives for better performance.

To capture the degree to which we are willing to tolerate false negatives, we introduce the parameter *error factor* $e_f \in [0, 1]$ that can be specified by the user. We then estimate $s(t)$ by $s'(t) + e_f \cdot E_b$. When $e_f = 1$, we guarantee no false negatives. As e_f decreases, the probability of false negatives increases, and when $e_f = 0$, we estimate $s(t)$ by $s'(t)$. In the experiments reported in Section 5, there are no false negatives in most of the cases, even when $e_f = 0$.

4.2.2 Histogram Maintenance

A location update contains the old and new information of a moving object, including the position, velocity and the time when these apply. When such an update is received, we compute both the old and new trajectories of the object. Then we adjust the DCT functions in the cells that the moving object passes by.

The adjustment comprises three steps. First, we unwrap the DCT function, i.e., we compute the number of moving objects during each time point within the lifespan of the function. The second step treats deletion and insertion differently. For a deletion, we simply decrease the number of moving objects by one during the period that the old trajectory intersects with the cell and then modify the start time of the lifespan of the function to the current time. For an insertion, we set the lifespan from the current time t_{now} to $t_{now} + H$, and initialize the number of moving objects exceeding the old period to zero. Then we increase the number of moving objects during the intersection period by one. Third, we calculate new DCT functions for the affected cells.

Algorithm DH_maintenance$(P_o(x, v, t), P_n(x, v, t))$
Input: P_o and P_n are the old and new object, respectively

1. compute the trajectory of P_o during $[P_o.t, P_o.t + H]$
2. $L_o \leftarrow$ list of cells intersected by P_o
3. compute trajectory of P_n during $[P_n.t, P_n.t + H]$
4. $L_n \leftarrow$ list of cells passed by P_n
5. $L \leftarrow L_o \bigcup L_n$
6. **for** each cell in L **do**
7. set the start time of lifespan to current time
8. $V \leftarrow$ value list of DCT function in its lifespan
9. **if** there is a deletion in this cell **then**
10. decrease corresponding value in V by one
11. **if** there is an insertion in this cell **then**
12. set the end time of lifespan to $P_n.t + H$
13. extend V to $P_n.t + H$, adding value 0
14. increase the corresponding value in V by one
15. compute new DCT function from V

end DH_maintenance.

Figure 6. DH maintenance algorithm

Figure 6 summarizes the maintenance algorithm. First, note that deletion and insertion may affect the same cell if the new trajectory does not deviate much from the old one. In such a situation, we only unwrap and recompute the DCT functions of the cell once, since we do the deletion and insertion together in the second step. Second, the lifespan we maintain is no longer than H, either in the deletion or in the insertion. This makes it possible to use the same number of parameters to represent the DCT function.

To exemplify, Figure 7(a) depicts an original DCT function of a cell before updates, and Figures 7(b) and (c) illustrate an independent deletion and an insertion in this cell. As shown in Figure 7(b), an object intersecting the cell during time t_2 to t_3 is deleted, and hence, counters at corresponding time points are decreased by one. Figure 7(c) shows an insertion of an object at time t_1. The lifespan of the function is then changed to $[t_1, t_1 + H]$. The trajectory of this object intersects with the cell from time t_4 to $t_1 + H$, and hence the corresponding counters are increased by one.

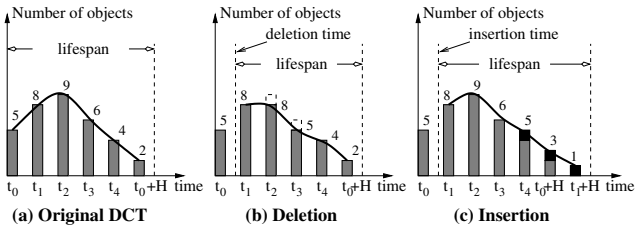


Figure 7. Maintenance in DH

4.3 Query Processing

4.3.1 The Filtering Phase

The filtering phase aims to identify areas that may possibly contain answers to the density query. The output of this step is a list of grid cells of sizes one to four times larger than the query window size. This is because the dense squares may intersect with one to four cells, as shown in Figure 8 (the shaded areas represent dense squares).

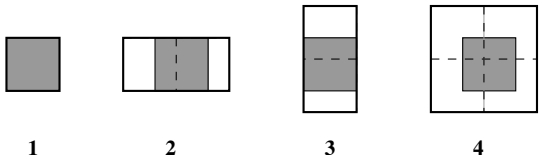


Figure 8. Intersection between the final answer and DH cells

We examine the cells of the DH in the order from left to right and top to bottom. The algorithm is shown in Figure 9.

Given a query window R and a query threshold ρ, we have $N_{min} = R \cdot \rho$, which is the minimum number of objects that should occupy a dense square. This way, we transform the density threshold ρ to the number of objects N_{min}. Then, for each cell, we compute the number of objects it contains at the query time t_q. If it contains at least N_{min} objects, it is added to the final answer list directly. Otherwise, we check the *square* consisting of four cells and having the current cell at the top left corner. If this square has less than N_{min} objects, it is obvious that this square does not contain any dense square. We can safely prune the current cell and set the tags of combinations of any cells in this square to false (i.e., we need not take into account these combinations next time).

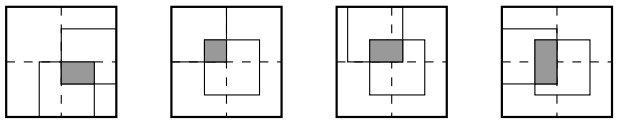
If the number of objects in the square is larger than N_{min}, we check the density of each cell in this square and report those cells satisfying the density threshold themselves. In the remaining cells, we check the cells of types 2 and 3 as shown in Figure 8. If the number of objects in the combinations is no smaller than N_{min}, we pass them to the refinement phase. Only when all three types of cells fail to contain any final answer, we pass the whole square to the refinement phase. Each time we get an answer from the refinement phase, we decrease the number of objects in the corresponding cells.

To avoid overlaps among final reported ranges, the area covered by the answer is tagged and will not be considered during the following search. We also adopt heuristics to help speed up the processing. As shown in Figure 10, in one square, cells of types 2 and 3 are not allowed to coexist; the fourth type of cell is not allowed to coexist with any other

Algorithm Density_query(ρ, R, t_q)
Input: threshold ρ, query window R, query time t_q

```
1.  N_min ← R · ρ
2.  for each cell in the space do
3.      N_b ← number of objects in the cell at t_q
4.      if N_b > N_min then
5.          report this cell as a final answer
6.      else
7.          N_s ← number of objects in square S_4
            // S_4 consists of four cells
8.          if N_s ≥ N_min then
9.              flag ← true
10.             for each combination of two cells S_2 do
11.                 N_2 ← the density in the two cells
12.                 if N_2 ≥ N_min then
13.                     invoke Refinement(S_2, ρ, R, t_q)
14.                     if an answer is found then
15.                         modify histogram
16.                         flag ← false
17.             if flag then
18.                 invoke Refinement(S_4, ρ, R, t_q)
19.                 if an answer is found then
20.                     modify histogram
end Density_query.
```

Figure 9. Density query algorithm



2 and 3 4 and 1 4 and 2 4 and 3

Figure 10. Conflicting types of cells

type of cell. Once an answer of one type is confirmed, we do not need to search the conflicting type.

4.3.2 The Refinement Phase

We introduce a new structure, which we term an *object pool*, that temporarily stores information for the objects in retrieved cells at the query time.

The refinement phase needs to obtain the objects in each candidate area. The algorithm first checks whether any part of a given candidate area has ever been retrieved. If this is the case, we load the objects from the object pool and tag this part. Then a general window query covering the untagged parts of the candidate area is issued on the index. We store the newly retrieved objects in the object pool.

After obtaining objects, algorithms that differ only slightly are applied to the different types of cells in order to identify final answers. There are cells of types 2, 3, and 4.

For cells of type 2, we sort the positions of the objects according to their x coordinates. Then we count the number of objects every l length units (l is the query window size) along the x axis until the count reaches or exceeds the threshold N_{min}, which means we have identified an answer. We handle the cells of type 3 similarly to those of type 2, except that we sort the object positions along the y axis this time. For type 4 cells, we first sort the object positions along the x axis. When we count the number of objects every l length units along the x axis, we maintain an array that stores the number of objects along the y axis. As the starting point of the counting moves forward, we decrease the corresponding values in the array. Once we obtain a count that reaches or exceeds N_{min}, we will check along the y axis as in the case of type 2.

Objects in the top-left cell are discarded from the object pool since this cell may not be accessed any more according to our scan order. Therefore, the object pool only needs to store up to a row of cells. Moreover, each time we identify an answer, we remove the objects in the answer set from the object pool.

The detailed algorithm is shown in Figure 11.

Algorithm Refinement(S, ρ, R, t)
Input: candidate area S, density threshold ρ
query window R, query time t

1. $N_{min} \leftarrow R \cdot \rho$, $S_r \leftarrow \phi$, $L_1 \leftarrow \phi$
2. **for** each cell B in S **do**
3. **if** the cell B has been retrieved **then**
4. load objects from object pool to L_1
5. $S_r \leftarrow S_r \bigcup B$
6. $L_2 \leftarrow$ WindowQuery$(S - S_r, t)$
7. $L \leftarrow L_1 \bigcup L_2$
8. $l \leftarrow \sqrt{R}$
9. **if** S is of type 2 or 4 **then**
10. sort objects in L along x-axis
11. project objects to x-axis
12. **else**
13. sort objects in L along y-axis
14. project objects to y-axis
15. $N \leftarrow$ the number of objects within each l length
16. **if** any N larger than N_{min} **then**
17. **if** S is not of type 4 **then**
18. report an answer
19. **else**
20. project objects to y-axis
21. $M \leftarrow$ number of objects within each l length
22. **if** any M larger than N_{min} **then**
23. report an answer

end Refinement.

Figure 11. Refinement algorithm

5 Experimental Study

5.1 Experimental Settings

All the experiments were run on a 2.6G Pentium IV desktop with 1 Gbyte of memory. The page size is 4K.

We employ the B^x-tree [4] as the index for the refinement phase. It has three phases and a node capacity of 200 entries. An LRU page buffer of 50 pages is used [5], with the internal nodes of a tree being pinned in the buffer.

The space domain is 1000×1000 units. The datasets are generated using an existing data generator, where objects move in a network of two-way routes that connect a given number of uniformly distributed destinations [14]. Objects start at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Whenever an object reaches one of the destinations, it chooses the next target destination at random. Objects accelerate as they leave a destination, and they decelerate as they approach a destination. In most experiments, the average interval between two successive updates of an object equals 60 time units. Unless noted otherwise, the number of moving objects is 100,000.

The query workload is 100 density queries. Each query has three parameters: (i) the density threshold ρ; (ii) the squared-shaped query window side length l; and (iii) the prediction lengths q_l. The query cost is measured in terms of CPU time and I/O.

The parameters used are summarized in Table 1, where values in bold denote the default values used.

Parameter	Setting
Page size	4K
Buffer number	50
Node capacity	200
Max update interval	**60**, 120, 240
Density threshold	0.8, 0.9, **1.0**, 1.1, 1.2
Max prediction length	30
Query window size	20, **25**, 50
Number of queries	100
Dataset size	**100K**, ..., 1M
Number of destinations in dataset	**50**, 100, 200, 300

Table 1. Parameters and their values

5.2 DCT Compression Accuracy

First, we look at a representative result of the DCT compression of the histograms. We execute 100 density queries with query window size of 25 in a 100K dataset. Figure 12 shows the actual numbers of objects ($s(t)$) and the restored numbers of objects from the DCT compression ($s'(t)$, which we term the DCT compression in the sequel) in cells of density 0.8 and 1.2, respectively, as a function of elapsed time. We see that the curves for the predicted number of objects

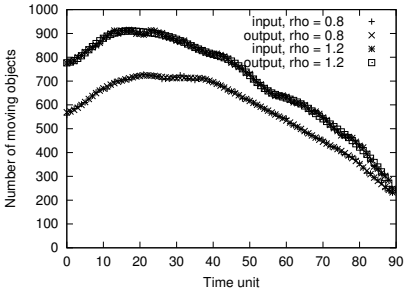


Figure 12. DCT compression accuracy

match the curves for the actual number of objects very well.

Next, we evaluate the accuracy of the DCT compression using two metrics, the rates of false positives and false negatives, while varying the number of DCT coefficients used, elapsed time, and the error factor e_f. The *rate of false positives* indicates the percentage of squares in the answer set that have lower density than the given threshold, and the *rate of false negatives* is the percentage of squares that are missing in the answer set to the total number of correct answers. The correct answers are obtained by using the histograms without compression by DCT.

5.2.1 Effect of the Number of DCT Coefficients

First, we investigate the effect of the number of DCT coefficients. We create a density histogram of 100K data at time 0, and then issue 100 square density queries of size 25, prediction length in the range $[0, 30]$, and density threshold equal to 0.8, 1.0, and 1.2, respectively.

Figure 13 shows the rates of false positives and negatives for varying numbers of DCT coefficients. As expected, the

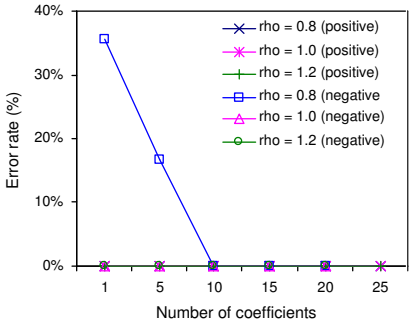


Figure 13. False positives and negatives for varying DCT coefficients

more DCT coefficients we used, the better the accuracies of the results. Both types of errors virtually disappear when the number of DCT coefficients exceeds 10. The results suggest that our method is highly accurate while saving about 90% of the space (the original DCT has 90 coefficients).

In addition, the DCT functions work well when the density threshold is large. When the density threshold is close to the average density, both types of error increase since a small deviation in the DCT function may wrongly report or prune many squares.

5.2.2 Effect of Time

We use 20 DCT coefficients in the following experiments. The density histogram and the index of 100K objects are created at time 0 and are then maintained until time 240. To avoid frequent transformations between the DCT and the real data, we employ the batch update technique where each batch contains 1,000 updates. After each maximum update interval (60 time units), density queries with the same parameters as in the previous experiment are issued. Figure 14

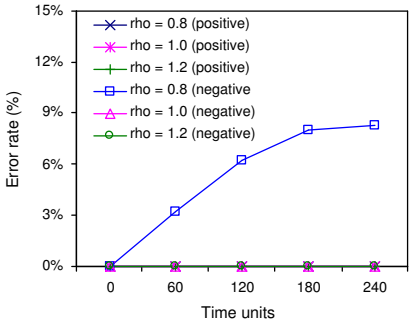


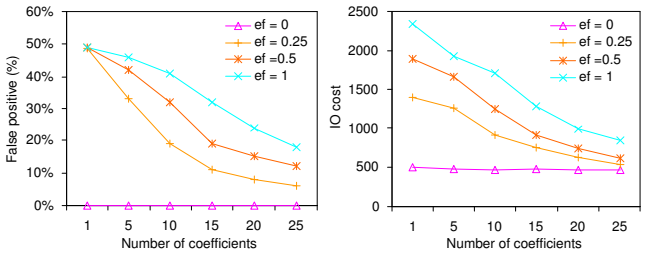
Figure 14. False positives and negatives with elapsed time

plots the false positives and negatives.

We observe that the false positive decrease to 0% as time passes, while the false negatives approach 10%. This is because the DCT compression underestimates the real number, resulting in fewer false positives, but also missing answers. Moreover, new coefficients are computed based on existing ones, leading to an increasing underestimation.

5.2.3 Effect of the Error Factor

Figure 15(a) and (b) show the false positives and the query I/O cost, respectively, for varying error factors and numbers of DCT coefficients.



(a) False positives (b) Query I/O cost

Figure 15. Effect of the error factor and DCT coefficients

The false positives decrease as the number of DCT coefficients increases—with more coefficients, the DCT becomes more accurate. We also observe that when the error factor in creased, the false positives also increase. This behavior is expected since the error factor indicates how much we overestimate the number of objects. When we guarantee no false negatives, that is $e_f = 1$, the false positives are at about 50% when using only one DCT coefficient, but decrease to about 20% when additional coefficients are used (but still significantly smaller than the total number).

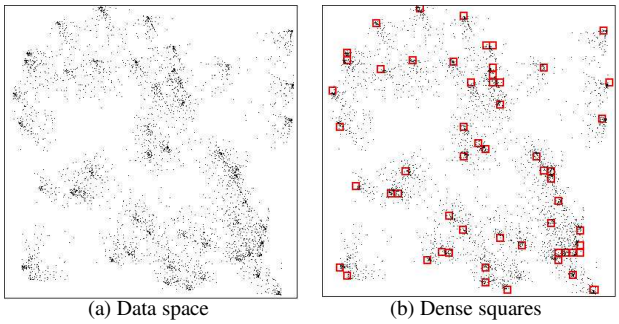
The actual false positives for these experiments are at 0%. This means that we may need to overestimate to guarantee no false negatives, although the actual numbers of false negatives are very low, even when we do not overestimate at all. The query I/O cost shown in Figure 15(b) exhibits a similar trend to that for the false positive. This is because the more the false positives, the more times we need to search the index, which increases the I/O. Similarly, there is a trade-off between the error factor and query I/O. When we use a smaller error factor, that is, larger probability of false negatives, we obtain better query performance.

5.3 Density Queries

We proceed to evaluate the efficiency of the density query processing algorithm while varying different parameters. We start by showing an example of the results of a density query.

5.3.1 An Example of the Density Query

Figure 16(a) shows a snapshot of the dataset with 50 destinations in the simulated road network. Given a density



(a) Data space (b) Dense squares

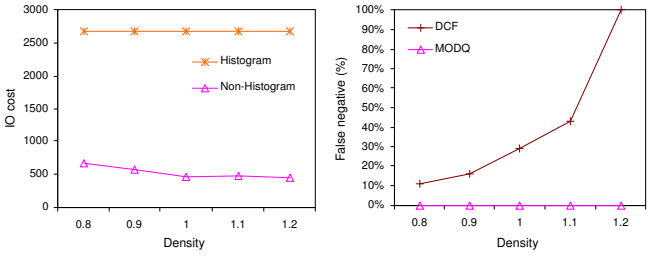
Figure 16. Density query example

query with query window size 25 and density threshold 1.0, Figure 16(b) shows the density query result (denoted by squares) obtained. The algorithm identifies all the dense regions.

5.3.2 Histogram versus Non-Histogram

This experiment evaluates the pruning effectiveness of the DH. We compare our method, which uses the DH, with a straightforward method as described in Section 3, which uses the MODQ framework and the same refinement algorithm as in our histogram-based algorithm, but does not use histograms and the two-phase query algorithm (we simply call it the non-histogram algorithm). To locate the region of the required density, the non-histogram algorithm executes a series of window queries covering the whole space, where each window query covers a square consisting of four cells. As they use the same refinement algorithm, the CPU times used by the two algorithms are similar. Therefore, we mainly compare their query I/O costs.

Figure 17(a) shows the average number of I/O operations per density query for varying density thresholds. Our algorithm improves over the non-histogram at a factor of 4 in terms of I/O cost.



(a) Histogram versus non-histogram (b) The MODQ versus the DCF

Figure 17. Comparison with non-histogram and the DCF

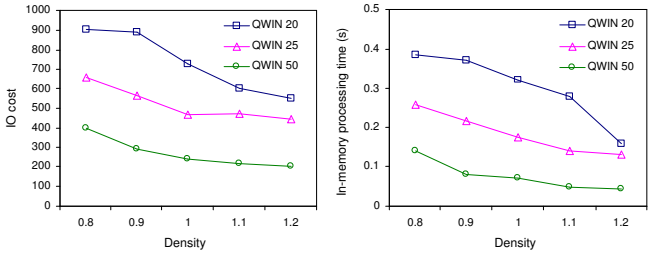
5.3.3 The MODQ versus the DCF

We also compare our algorithm with the *dense cell filter* (DCF) algorithm [3], which exhibits the best performance among other algorithms. Due to the different definitions of the density query, the DCF is not able to identify dense squares across cells. Figure 17(b) shows the percentages of lost answers for the DCF and our algorithm. We can see that the number of lost answers of the DCF increases quickly as the density threshold increases. This is because the number of correct answers decreases as the density threshold increases, with the effect that there are relatively more lost answers. As expected, our algorithm has no answer loss.

5.3.4 Effect of Density Threshold and Query Size

Next, we investigate the effect of the density threshold and the query window size. Since the cost of the refinement phase always dominates the cost of a query, in order to study the behavior of the filter and refinement algorithms independently on the index structure, we partition density query the cost into the window query cost (I/Os) and in-memory processing cost (CPU time), and then plot them separately.

Figure 18(a) and (b) measure the density query performance in the same 100K dataset for varying density thresholds and query window sizes. Figure 18(a) shows the window query I/O cost per density query. The I/O cost decreases as the density threshold increases. The reason for this behavior is as follows.



(a) Query I/O cost (b) In-memory processing time

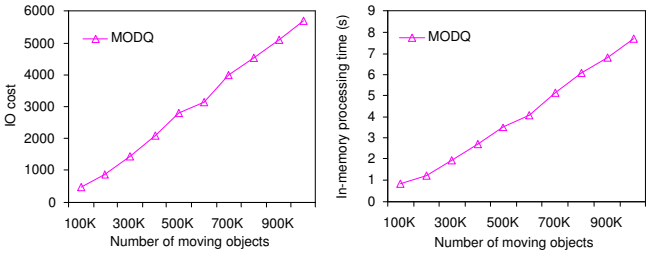
Figure 18. Effect of density threshold and query size

When the density threshold is close to the average density (about 0.5 for a query window size of 25), the pruning ability of the filtering phase decreases. This is so because most regions of the data space have average density. Even if the density of one cell is lower than the threshold, its combination with other cells still have high probability of satisfying the density threshold. Hence, many window queries are issued in the refinement phase, which results in higher query cost. When the density threshold is high, the filtering phase can prune cells that have few objects. Thus, the total window query cost is smaller. Moreover, we observe that the query cost decreases as the query window becomes larger. This is because the number of answers is smaller for larger query windows; hence, fewer window queries are issued in the refinement phase.

Figure 18(b) shows the corresponding in-memory processing cost. The trends are similar to those seen for the window query I/O cost. The reasons for this behavior are similar to those given for the I/O cost.

5.3.5 Effect of Database Size

To test the scalability of our technique, we performed the density query while varying the number of moving objects from 100K to 1M. We fix the density threshold at 0.8 and use a query window size of 25. Figure 19(a) and (b) show the window query I/O cost and the in-memory processing cost per density query, respectively. We observe that both the I/O cost and the in-memory processing cost grow linearly as the number of moving objects increases. This is because the average density in each cell increases as the number of moving objects increases. More regions that satisfy the density requirements need to be checked.

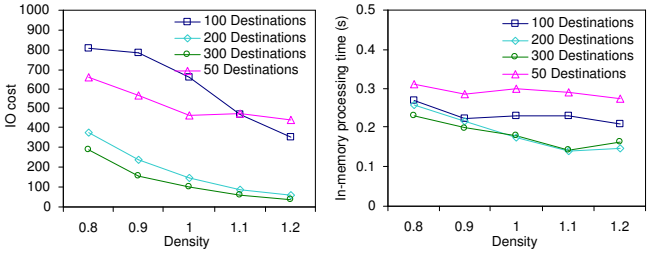


(a) Query I/O cost (b) In-memory processing time

Figure 19. Effect of database size

5.3.6 Effect of Data Distribution

We evaluate the density query performance for different data distributions by using numbers of destinations in the simulated route network in the range from 50 to 300. The fewer the destinations, the more skewed the dataset becomes. Figure 20(a) and (b) plot the query I/O cost and in-memory processing cost, respectively. We observe that the query costs of the 50- and 100-destination datasets are higher than those of the 200- and 300-destination datasets. This is because



(a) Query I/O cost (b) In-memory processing time

Figure 20. Effect of data distribution

skewed data tend to result in high density in more regions so that more queries in the index are needed.

5.4 Maintenance Cost

Finally, we evaluate the maintenance cost of our histogram-based algorithm while varying the length of the maximum update interval U (the query prediction length is fixed at 30). We create the index at time 0 and then perform object updates for the duration of a maximum update interval. Figure 21 shows the average maintenance cost per insertion or deletion. We observe that the average maintenance cost increases as the maximum update interval increases. The reason is that each update of an object incurs updates on the cells that its trajectory intersects. As the maximum update interval increases, the length of the trajectory increases, and therefore the maintenance cost also increases.

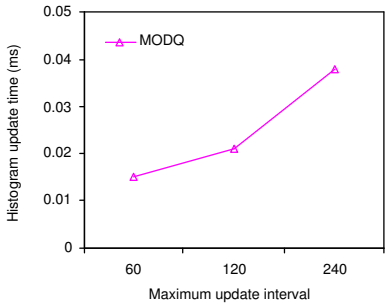


Figure 21. Maintenance cost

6 Conclusion and Research Directions

This paper assumes a setting where a population of objects—with positions represented by liner functions of time—move continuously in the Euclidean plane. In this setting, the paper studies the querying for dense regions. In particular, the paper defines a particular type of density query that eliminates answer loss. To render the query more amenable to efficient computation, while still returning meaningful results, the paper specializes the setting to consider square queries of fixed size.

The paper then proceeds to propose a two-phase filter-and-refinement algorithm that computes such queries efficiently. This algorithm uses a density histogram for the filter step that, for each cell in a uniform grid covering the data space, records the time varying count of objects that intersect the cell. This count is approximated, and thus compacted, by the use of the discrete cosine transform. Results of an extensive experimental study is reported that yields insight into the performance behavior of the algorithm. In particular, the results indicate that the algorithm is efficient for the ranges of parameter settings considered.

Several promising directions for future work exist, one being to consider dense regions of different sizes. It is convenient to extend the current algorithm to support cells of size 2^n. Another direction is to support regions of more general shapes, e.g., convex regions.

Acknowledgments This work is supported by the $SpADE$ (A SPatio-temporal Autonomic Database Engine for location-aware services) [1] project, which studies the provisioning of mobile users with location-based data, e.g., traffic data, maps, and points of interest.

Christian S. Jensen is also an adjunct professor at Agder University College, Norway.

References

[1] Spade. *http://spade.ddns.comp.nus.edu.sg/spade*.

[2] A. Civilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. In *Proc. MobiQuitous*, pp. 164–173, 2004.

[3] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databses. In *Proc. SSTD*, pp. 306–324, 2003.

[4] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proc. VLDB*, pp. 768–779, 2004.

[5] S. T. Leutenegger and M. A. Lopez. The effect of buffering on the performance of R-trees. In *Proc. ICDE*, pp. 164–171, 1998.

[6] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *Proc. KDD*, pp. 617–622, 2004.

[7] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2): 40–49, 2003.

[8] B. C. Ooi, K. L. Tan, and C. Yu. Fast update and efficient retrieval: an oxymoron on moving object indexes. In *Proc. Int. Web GIS Workshop, Keynote*, 2002.

[9] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proc. SSTD*, pp. 443–459, 2001.

[10] D. Papadias and Y. Tao. Range aggregate processing in spatial databases. *TKDE*, 16(12): 1555–1570, 2004.

[11] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *Proc. ICDE*, pp. 166–175, 2002.

[12] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *Proc. ACM SIGMOD*, pp. 637–646, 2004.

[13] K. R. Rao and P. Yip. Discrete cosine transform: algorithms, advantages, applications. *Academic Press Professional*, 1990.

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

[15] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pp. 790–801, 2003.

[16] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *Proc. ACM SIGMOD*, pp. 443–454, 2004.