On-Line Discovery of Dense Areas in Spatio-temporal Databases

Marios Hadjieleftheriou
University of California, Riverside
marioh@cs.ucr.edu

George Kollios
Boston University
gkollios@cs.bu.edu

Dimitrios Gunopulos
University of California, Riverside
dg@cs.ucr.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

Abstract

Moving object databases have received considerable attention recently. Previous work has concentrated mainly on modeling and indexing problems, as well as query selectivity estimation. Here we introduce a novel problem, that of addressing density-based queries in the spatio-temporal domain. For example: "find all regions that will contain more than 500 objects, ten minutes from now". The user may also be interested in finding the time period (interval) that the query answer remains valid. We formally define a new class of density-based queries in spatio-temporal databases and give approximate, on-line techniques that answer them efficiently. Typically, the threshold above which a region is considered to be dense is part of the query. The difficulty of the problem lies in the fact that the spatial and temporal predicates are not specified by the query. The techniques we introduce find all candidate dense regions at any time in the future. To make them more scalable we subdivide the spatial universe using a grid, and limit queries within a pre-specified time horizon. We base our work on Dense Cell Filters and Approximate Frequency Counts (lossy counting). Finally, we validate our approaches with a thorough experimental evaluation.

1 Introduction

Databases that manage moving objects have received considerable attention in recent years due to the emergence and importance of location-aware applications like intelligent traffic management, mobile communications, sensor-based surveillance systems, etc. Typically, the location of a moving object is represented as a function of time and the database stores the function parameters [2, 1, 17, 9, 22, 21, 16, 24, 15, 27, 23, 10]. This results into a tractable update load. The system is updated only when an object changes any of its moving parameters (e.g. speed, direction, etc). The alternative of storing the object's continuously changing location is practically infeasible since it would correspond to one update per object for each time instant [23]. Most works assume that object trajectories are linear functions of time. For example, given the location $o(0)$ of object o at time $t = 0$ and its current velocity vector o_V, its position at some future time t can be computed by $o(t) = o(0) + o_V t$. A database that maintains the moving functions can compute and thus answer interesting queries about the locations of the moving objects in the future. Examples include range queries: "find which objects will be in area A, 10 minutes from now" [10, 2, 17, 9, 22, 21, 20], nearest neighbor queries: "find the closest object(s) to a given location within the next 5 minutes" [24], etc. The answer to such queries is based on the knowledge about the object movements at the time the query is issued [25, 26].

In this paper we present a framework for answering *density*-based queries in moving object databases. An area is dense if the number of moving objects it contains is above some threshold. Discovering dense areas has applications in traffic control systems, bandwidth management, collision probability evaluation, etc. In these environments users are interested in obtaining fast and as accurate as possible answers.

We identify two interesting versions of the problem: Snapshot Density Queries (SDQ) and Period Density Queries (PDQ). In SDQ the user wants to find the dense areas at a specified time instant in the future. Given a collection of objects moving on a 2-dimensional space, an SDQ example is: "find all regions that will have more than 1000 vehicles per square mile at 3:30pm". On the other hand, a PDQ query finds the dense areas along with the time periods (intervals) that the answers remain valid. For example, a basic operation in a cellular communication network is the identification of all cells that will become dense and for how long.

For the static version of the problem (i.e., where objects are not moving) a number of density based clustering methods have been proposed in recent years. The most relevant is the STING [29] algorithm, that uses a hierarchical structure to store statistical information about the dataset. However, this method cannot be used directly for the moving objects environment, since the update and space overhead will be large. Another algorithm is CLIQUE [3], suited best for high dimensional static datasets. The basic idea is to find dense areas in lower dimensions first, and continue recursively for higher dimensions.

Discovering arbitrary dense areas inside the universe is a rather difficult problem. Both STING and CLIQUE are grid-based approaches. They divide the universe uniformly into a number of disjoint *cells*. The problem is thus simplified to that of finding all the *dense cells*. We use the same approach for two reasons: For most cases (e.g. traffic management) the granularity of the grid can be set according to the user needs. In other applications (e.g. cellular networks) the grid structure is already available.

For many practical applications the exact density of a region is not critically important. For example, a query about potential traffic jams may tolerate a small approximation error. Our approach is to keep a small summary of the moving object dataset in main memory, that can be used to answer queries quickly. Also, due to high update rates the data structures must be easy to maintain in an on-line fashion. To achieve these goals we use spatio-temporal grids and propose techniques based on Dense Cell Filters and Approximate Frequency Counting (Lossy Counting). These methods guarantee against false negatives but have a few false positives. In applications where false positives cannot be ignored, our techniques can be used as a pre-filtering step. Our estimators can quickly identify the (typically few) candidate dense regions which must then be passed through a post-filtering step.

To the best of our knowledge, this is the first work that addresses density-based queries in a spatiotemporal environment. Previous work has dealt with range, join and nearest neighbor queries. Related to density queries is recent work on *selectivity estimation* for spatiotemporal range queries [27, 8]. Spatio-temporal estimators compute the number of objects that will cross a user defined spatial region at a user defined time instant in the future. However, a density query is "orthogonal" in nature since the user does not specify the spatial and temporal query predicates. One straightforward way to use a spatio-temporal estimator to identify dense areas is by computing the selectivity estimate for *each* cell in the spatio-temporal grid. While simplistic, this approach is clearly inefficient due to its large computational cost (the spatio-temporal grid typically contains too many cells).

The contributions of this paper can be summarized as follows:

- We identify two novel query types for spatio-temporal databases, based on the notion of density in space and time. We concentrate on tractable versions of the problem based on a regular spatio-temporal grid.
- We propose solutions that provide fast *approximate* answers, by building main memory summaries that can accommodate large update rates and deliver fast query responses.
- We present an extensive experimental study that validates the accuracy and efficiency of our methods. The proposed Dense Cell Filter approach has the most robust performance, requiring limited space and yielding a very small number of false positives.

2 Problem Definition

2.1 General Framework

In this section we define the general density query framework and our notation.

We assume that a database stores a set of N objects moving on a 2-dimensional plane. We model these objects as points, represented by tuples of the form: (x, y, v_x, v_y), where (x, y) is the current location and $\vec{v} = (v_x, v_y)$ the velocity vector. In our setting, objects can follow *arbitrary* trajectories (represented by generic functions of time). For illustration purposes and simplicity in the rest of the paper we refer to linear trajectories only, without loss of generality.

The objective is to find regions in space and time that satisfy interesting predicates. For 2-dimensional movements an interesting property is finding areas where objects tend to be very close to each other. We formalize this notion with the following definition:

Region Density. *The density of region R during time interval ΔT is defined as the ratio: $Density(R, \Delta T) = \frac{N_R}{Area(R)}$, where N_R is the number of objects inside R during ΔT and $Area(R)$ is the total area of R.*

An important observation is that regions with high density are not necessarily interesting. For example, two objects arbitrarily close to each other define a region with arbitrarily high density. Therefore, we allow the user to define both the level of density and the minimum and maximum area that a qualifying region must have. Given the above definition we can now state the following density based queries:

Period Density Query. *Given a set of N moving objects in space, a horizon H and thresholds α_1, α_2 and ξ, find regions $R = \{r_1, \ldots, r_k\}$ and time intervals $\Delta T = \{\delta t_1, \ldots, \delta t_k | \delta t_i \subset [T_{now}, T_{now} + H]\}$ such that: $\alpha_1 \leq Area(r_i) \leq \alpha_2$ and $Density(r_i, \delta t_i) > \xi$ (where T_{now} is the current time, $i \in [1, k]$ and k is the query answer size).*

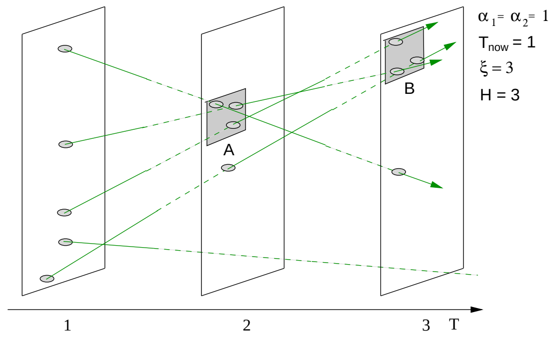


Figure 1: An example of objects moving linearly in 2-dimensional space over time.

Notice that in the above query we do not specify time or spatial predicates. Any method for answering this query must find not only the dense regions but also the time periods that these regions are dense inside the specified horizon H. Typically, we require the area to be within some size (α_1, α_2) since the most interesting cases are when a large number of objects are concentrated in a small region of space.

A special case of the period density query is:

Snapshot Density Query *Given a set of N moving objects in space, a horizon H, a time instant T_q ($T_q \in [T_{now}, T_{now} + H]$) and thresholds α_1, α_2 and ξ, find regions $R = \{r_1, \ldots, r_k\}$ such that: $\alpha_1 \leq Area(r_i) \leq \alpha_2$ and $Density(r_i, T_q) > \xi$ (where $i \in [1, k]$ and k denotes the query answer size).*

In Figure 1 we show an example of objects moving on a 2-dimensional surface. The current time is 1. The answer to a PDQ with $H = 3, \xi = 3$ and $\alpha_1 = \alpha_2 = 1$ (i.e., we are looking for regions 1 square unit in size), is $R = \{A, B\}$ and $\Delta T = \{[2, 2], [3, 3]\}$. It should be clear that by increasing $\alpha_{1,2}$ or decreasing ξ, increases the size of the answer. Meaningful values for $\alpha_{1,2}$ and ξ are, of course, application dependent.

The general queries are very difficult since the dense areas may come in many sizes and the horizon is not restricted. We proceed with their simplified versions based on spatio-temporal grids and *fixed* horizons.

2.2 Simplified Queries

We partition the universe using a grid of cells and consider the problem of finding the most dense cells. Assuming that the granularity of the grid is set according to user requirements, fast on-line discovery of the most dense cells is the first, most important step for answering general density queries. The solutions we propose are orthogonal to the grid generation process. For simplicity, we consider uniform grids but the techniques work for any partitioning. The emphasis in our treatment is in handling a very large number of cells efficiently.

We also choose a *fixed* horizon during which user queries can be answered. Our general definitions apply for a moving horizon inside which user queries can be answered. Valid answers can be provided for any time instant inside the time interval $[T_{now}, T_{now} + H]$ (the boundary of the horizon advances, as the current time advances). On the other hand, a fixed horizon would provide answers only inside the time interval $[T_{now}, H_f]$ (the horizon boundary remains constant). Simplified fixed horizons apply to most

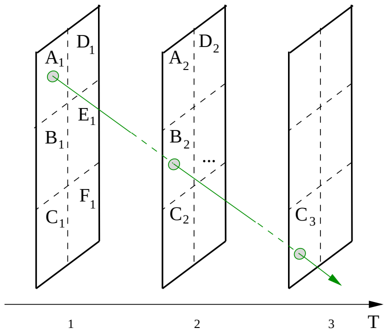


Figure 2: The space-time grid for $H_f = 3$. The trajectory of the object inserted at $t = 1$ will cross cells $\{A_1, B_2, C_3\}$.

practical moving object applications [22, 27, 8]; the user is typically interested in the near future for which the current information holds. When the horizon expires, the estimators are rebuilt [27, 8]. It is reasonable to assume that the user can decide in advance the time period during which most queries will refer to. A problem that arises with fixed horizons is that as T_{now} advances closer and closer to the horizon boundary, inevitably queries will refer to an ever decreasing time period into the future. One way to avoid this situation is by rebuilding the estimators halfway through the horizon. For example, assume that $T_{now} =$ 1:00pm and $H_f =$ 2:00pm. After half an hour (at 1:30pm), we can rebuild the estimator for an extra 30 minutes. We reuse information from 1:30 until 2:00pm and calculate for 2:00 until 2:30pm. Essentially, we can answer queries for at least 30 minutes in the future, at all times.

Using the spatio-temporal grids and the fixed horizons the queries we solve have the following two forms:

Simple Period Density Query. *Find the cells with density larger than a user specified threshold ξ inside the fixed horizon, along with the time period that the answer is valid.*

Simple Snapshot Density Query. *Find the cells with density larger than a user specified threshold ξ at a user specified time T_q, inside the fixed horizon.*

First, a grid granularity and an appropriate horizon length is decided. Conceptually, we create a spatial grid for every time instant inside the horizon. For example, assume a horizon of three time instants and let the 2-dimensional spatial universe be 100 miles long in each direction, with a grid granularity of 1 square mile. This will divide the space-time grid into $3 \times 100 \times 100 = 30,000$ cells. All cells are enumerated with unique IDs (Figure 2). One straightforward approach to address the problem follows. Since the speed and direction of a moving object are known at insertion time, we can extrapolate its trajectory and find all the cells that the object will cross in the space-time grid. Every object update is thus converted into a set of cell IDs, (at most) one ID per time instant of the horizon. By maintaining the number of crossings per cell, we know each cell's density at any time instant in the horizon. The same technique is used for handling deletions and updates, by adjusting the appropriate cells.

While simplistic, this approach has several disadvantages. First, in any practical application the total number of cells of the space-time grid is expected to be very large (in millions). Keeping a density counter

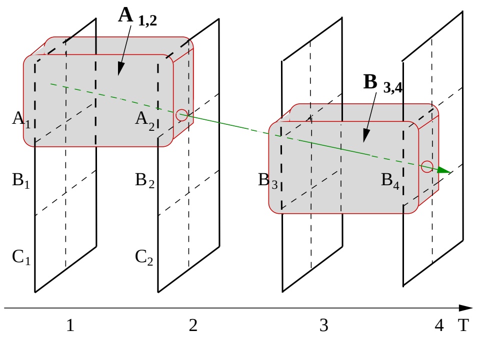


Figure 3: A coarse grid of granularity 2. Cells A_1 and A_2 are combined into a single bucket with counter $A_{1,2}$.

for each cell consumes unnecessary space. We could reduce space by keeping only the cells that have been crossed so far and discard cells with zero density. For real datasets however this will not decrease space substantially, since most cells will be crossed at least once.

We propose several techniques that compress the density information by building estimators that identify only the most dense cells and discard the rest. The penalty of using estimators is two fold. First, the reported densities are approximations. Second, in order to guarantee no false dismissals a number of false positives will be reported along with the correct answers.

3 Grid-Based Techniques

We first describe a simple improvement based on *coarse grids*. We then present an *Approximate Frequency Counting* algorithm (termed Lossy Counting) that appeared in [18] and can be modified and used in our setting. Finally, we introduce another approach the *Dense Cell Filter*.

3.1 Coarse Grids

Coarse grids are a general technique that can be utilized by all subsequent estimators for decreasing their size. Instead of keeping one density counter per cell for the whole space-time grid (figure 2), multiple consecutive cells in time are combined into a single bucket associated with only one counter (figure 3). The granularity of the coarse grid is defined as the number of consecutive cells in time that belong to the same bucket. By making the buckets larger (decreasing the granularity) we can decrease the size of the original grid substantially It should be noted that coarse grids compress the grid in the time dimension only. Combining cells in the spatial dimensions is not applicable since it would decrease the user defined grid granularity.

Figure 3 shows an example of a coarse grid with granularity 2 (buckets are illustrated with the shaded regions). Essentially, the number of total counters is reduced by half. An object trajectory crosses buckets $A_{1,2}$ and $B_{3,4}$, and thus these counters should be adjusted.

The coarse grid guarantees no false dismissals but may introduce false positives. For example, the reported density for cell A_2 is the value of counter $A_{1,2}$. $A_{1,2}$ increases once for every cross of any cell (A_1 or A_2) that belongs to that bucket. If the number of crosses of either A_1 or A_2 exceeds threshold ξ, $A_{1,2}$ will also exceed ξ, i.e., no false negatives are created. On the other hand, very fast moving objects may introduce false positives. If the slope of a trajectory is large, the object might pass through a bucket without crossing all the cells that belong to it. In such cases, the densities of some cells are overestimated. Obviously, the larger the bucket width (and the faster the objects) the more false positives that may be created.

Answering an SDQ is straightforward. We locate the cells that correspond to time instant T_q and report the ones that belong to dense buckets. To answer a PDQ we scan the grid that corresponds to the beginning of the horizon (assume T_{now} for simplicity) and locate all the dense buckets. For every dense bucket B, we check later time instants to find how many consecutive buckets are also dense.

3.2 Lossy Counting

Lossy counting is a deterministic algorithm introduced in [18] for computing item frequencies over a stream of transactions. It guarantees an upper bound of $\frac{1}{\epsilon}\log \epsilon L$ space, where L is the current length of the stream. The input to the algorithm is an error threshold ϵ. The output is a set containing the most frequent items. The estimated frequencies differ from the actual frequencies by at most ϵL. The algorithm calculates the frequency of the items as a percentage of L and detects the ones that appear with the highest rate (as L continuously increases). For example, a query has the following form: "Find items that appeared until now in the stream for more than 70% of the time.". The answer to this query would be all items with frequency $f \geq 0.7L$.

The actual algorithm appears in figure 4. The incoming stream is divided into consecutive buckets. Every bucket B consists of w items, where $w = \frac{1}{\epsilon}$ is the width of each bucket. The algorithm maintains a set F of the most frequent items. Every entry in F has the form $E = < id, f, \Delta >$, where id is the item identifier, f the estimated frequency and Δ the maximum possible error in f. Initially F is empty and $B = 1$. Whenever a new item I arrives, the algorithm checks whether I is in F. If it is, the entry is updated by increasing its frequency ($I.f$). If it is not, a new entry is inserted in F. Every w items the bucket boundary is reached and F is updated by removing the items that are not frequent anymore. At any point, the estimator can return all items that appeared more than pL times (where $p \in [0, 1]$ and is query specified), by outputting all entries in F for which $E.f \geq (p - \epsilon)L$.

This technique works well for finding the most frequent items. Once such items are inserted into F their frequency is being calculated and, most probably, they will never be deleted from F. Items that are not as frequent will be inserted and later removed from F. We can apply the Lossy Counting approach for density queries with some *modifications*. Instead of a stream of items we have a sequence of object insertions and deletions. Every insertion is converted into a set of cell IDs (Figure 2). We input the IDs that are being crossed by the object trajectories as the streaming values of the Lossy Counting algorithm. The dense cells, whose IDs appear more frequently, are kept in F. Less interesting cells are omitted. A

Input: Stream of items $\{I_1, I_2, \ldots\}$. Error ϵ.
Output: Set F of entries $E =< id, f, \Delta >$ representing the most frequent items in the stream.

$F = \emptyset, w = \frac{1}{\epsilon}$
$B = 1, L = 0$

New item I appears in stream:
  // Check for bucket boundary first.
  If $L > 0$ and $L \bmod w = 0$:
    For all $E \in F$:
      If $E.f + E.\Delta \leq B$: remove E from F
    $B+ = 1$
  // Process the new item.
  $L+ = 1$
  If $I \in F$: $I.f+ = 1$
  Else: add $E =< I.id, 1, B - 1 >$ in F

Return the most frequent items:
  For all $E \in F$:
    If $E.f \geq (p - \epsilon)L$: return $E.id$

Figure 4: The Lossy Counting algorithm.

deletion is straightforward. For a cell C crossed by the old trajectory, if $C \in F$ we decrease $C.f$ by one. Cells that become non-frequent will be deleted at the bucket boundary.

This algorithm can find the cells that have been crossed by more than pL trajectories, where $L \approx NH$ is the number of IDs that have appeared on the stream, a number roughly as large as the number of object updates N, times the remaining horizon length H. Since not all trajectories cross one cell per time instant of the horizon but some extend outside of the universe, L might be smaller than the above product. However, user queries have a pre-specified density threshold ξ that represents an absolute number of objects, while Lossy Counting can only produce entries that are frequent with respect to L. Thus we have to express ξ as a percentage of L. In particular, in order to retrieve all the answers we need to find p such that $pL \leq \xi$. However, since $p \leq \frac{\xi}{L} \Rightarrow \lim_{L \to \infty} p = 0$, as L increases no p may satisfy the query. To see this, suppose we return all cells with frequencies as in:

$$A = \{C | C.f \geq (\xi - \epsilon)L\} \Rightarrow A = \{C | C.f \geq \xi L - \epsilon L\}$$

while the correct answer is:

$$CA = \{C | C.f \geq pL - \epsilon L\}$$

Since $pL \leq \xi$, we get:

$$Cardinality(A) \leq Cardinality(CA)$$

This means that we might miss some of the answers. Unfortunately, this pitfall cannot be avoided unless

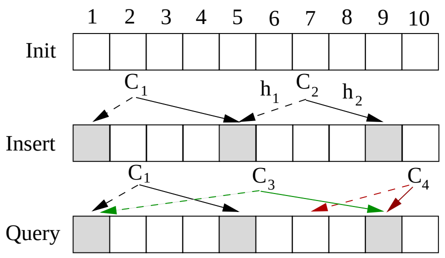


Figure 5: A Bloom Filter using $K = 2$ hash functions and $M = 10$ bits.

the estimator is rebuilt more frequently such that L does not become too large (so that a p exists that satisfies the query).

Regarding implementation, set F can be organized as a collection of hash tables, one for every time instant of the horizon. To answer an SDQ with time T_q and threshold ξ we find all cells with density larger than ξ in the appropriate hash table. To answer a PDQ we scan the hash table that corresponds to the beginning of the horizon (assume T_{now} for simplicity) and locate all the dense cells. For every dense cell $C_{T_{now}}$, we probe the hash table at time instant $T_{now} + 1$ to see if the corresponding cell $C_{T_{now+1}}$ is also dense, and continue as long as the cell remains dense.

An interesting observation for Lossy Counting, is that cells that may be false positives have densities in the interval $[(\xi - \epsilon)L, \xi L]$. All cells with larger densities belong surely to the correct answer. However, in the worst case, all reported cells may have densities in the aforementioned interval. That is, the method does not provide any guarantees on the number of false positives, neither this number can be estimated.

3.3 Dense Cell Filter

To overcome some of the above limitations, we introduce the Dense Cell Filter approach. This novel solution uses a (non-trivial) modification of a basic Bloom Filter to create a small summary of the dense cells. A Bloom Filter [5] is a simple randomized data structure for representing a set in order to support membership queries, and it is very space efficient.

Given a set $S = \{C_1, \ldots C_E\}$ of E elements the problem is to find if item X belongs to S (membership query). A basic Bloom Filter is a hashing scheme that uses a bit-vector array with M bits (initialized to zero) and a set of K independent hash functions $H = \{h_1, \ldots, h_K\}$, that produce values in the range $[1, M]$. For every C_e ($e \in [1, E]$) all K hash functions $h_k(C_e)$ ($k \in [1, k]$) are computed, producing K numbers. Each number maps to a bit in the bit vector hence, the corresponding bits are *set*. To find if X belongs to S, the hash functions are applied on X to produce K values. If all K values map to bits that are set, X is in S with some probability. If at least one bit is not set, then X is not in S.

Figure 5 shows an example. The filter has $M = 10$ bits and uses $K = 2$ hash functions. Initially all bits are *reset* (set to 0 and depicted as white boxes). During the insertion stage we insert set $S = \{C_1, C_2\}$. Hash functions $h_1(C_1) = 1, h_2(C_1) = 5$ and $h_1(C_2) = 5, h_2(C_2) = 9$ are computed and the resulting bits are set (shaded boxes). During the query stage, three membership queries are performed. For C_1 and C_3

the resulting bits are already set thus these items might be in S or not. It is apparent that C_3 is a false positive, since the item was never inserted in the filter. On the other hand, C_4 certainly does not belong to S, since $h_1(C_4) = 7$ is not set.

Assuming perfect hash functions it can be shown that the probability of a false positive is $(1 - e^{-\frac{KE}{M}})^K$. By knowing the values of E and M, the number K of hash functions that minimize the number of false positives can be computed. This is achieved for $K = \ln 2(\frac{M}{E})$ [6]. By increasing M we decrease the probability of false positives, but at the same time the size of the Bloom Filter increases.

There are many interesting variations that improve on the basic Bloom Filter. For example, one problem is that two or more hash functions can map to the same bit causing a collision (as shown in figure 5). To overcome this issue, instead of using K hash functions that map to the same M values, we can use K bit vectors (each vector is referred to as a *stage*), each one with a separate hash function and size equal to $\frac{M}{K}$. Another variation, directly related to our approach, is the *counting Bloom Filter* [12]. Each bit of the bit vector is replaced with a small counter (for example 2 bytes). Every time a new element is inserted, the corresponding counters are incremented (instead of the bits just being set). When an element is deleted, the counters are decremented.

In our environment we have a spatio-temporal grid consisting of a set $S = \{C_1, \ldots, C_E\}$ of E distinct cells (or buckets in case of coarse grids) and we want to store the density of each cell using as little space as possible. We also want to identify the most dense cells quickly. We thus build a counting Bloom Filter that stores S, and maintain it on-line. Every time an object update is performed, we convert it into a sequence $U \subset S$ of cell IDs and insert them into the filter. If the counter of a cell C_i becomes larger than ψ, we insert C_i into the list of dense cells (DCL). From that point, the frequency of C_i is updated in the DCL only, without affecting the filter. Deletions are straightforward. They just reverse the effect of the corresponding insertion. If the estimated density of a cell becomes less than ψ, this cell is removed from the DCL. A detailed description of our algorithm is shown in figure 6.

One improvement for this algorithm is to keep a separate Bloom Filter per time instant of the horizon. As time progresses, Bloom Filters that refer to the past can be discarded to save space. Also, since fewer cells are associated with each filter, we expect to have fewer false positives. Since the DCL list can grow very large, it should be organized as a set of in memory hash tables, one for every time instant of the horizon.

An interesting issue that arises with this approach is that the DCL contains all cells that exceed threshold ψ, specified at construction time. If the user poses a query with $\xi < \psi$, the DCL cannot report an answer. There are two solutions for such queries. Either the filter has to be rebuild with a full database scan, or it has to adapt to the lower threshold ξ gradually, while postponing the answer to the query. With the adaptive threshold approach the filter cannot guarantee anymore that some dense cells will not be missed. Nevertheless, the Dense Cell Filter is still a valuable approach since for most applications the lower possible threshold can be decided at construction time.

In contrast with Lossy Counting, the Dense Cell Filter can provide certain probabilistic guarantees on

Input: Set of cells $S = \{C_1, \ldots, C_E\}$. Number of stages K and hash functions $h_{1,\ldots,K}$. Counters per stage M. Threshold ψ. Horizon H.
Output: A dense cell list (DCL).

$DCL = \emptyset$
$V_{1,\ldots,K}$ = vectors of M counters
For $k = 1 \to K, m = 1 \to M : V_k[m] = 0$
Object insertion:
$U \subset S$ = set of cell IDs crossed by object during H
For all $C_i \in U$:
  If $C_i \in DCL : C_i.density+ = 1$
  Else:
    For $k = 1 \to K : V_k[h_k(C_i)]+ = 1$
    If all $V_k[h_k(C_i)] \geq \psi$: add C_i in DCL
Object deletion:
$U \subset S$ = set of cell IDs crossed by object during H
For all $C_i \in U$:
  If $C_i \in DCL$:
    $C_i.density- = 1$
    If $C_i.density < \psi$: remove C_i from DCL
  Else:
    For $k = 1 \to K : V_k[h_k(C_i)]- = 1$

Figure 6: The Dense Cell Filter algorithm.

the number of false positives introduced. Below, we show that the probability of a cell with density $c < \xi$ being reported is small and depends on the number of objects, the number of counters and c and ξ.

Lemma 1 *Given N moving objects, a query threshold ξ and a Bloom Filter with one stage of M counters per time instant of the horizon, the probability that cell C with density $c < \xi$ is being reported by the filter, is in the worst case $P \leq \frac{1}{M}\frac{N-c}{\xi-c}$.*

Proof: Consider the c-th update for cell C (which makes the density of this cell equal to c) and assume that after this update the counter for C ($h(C)$) gets the value ξ. The cell will be added to the DCL and thus reported by the filter. Since the cell already contains c objects, the counter contains another $\xi - c$ objects from other cells. We have N objects in total and in the worst case $N - c$ of those can be distributed in $\frac{N-c}{\xi-c}$ counters. In that case, the probability of C being added to the DCL is equal to the probability of C hashing to one of the $\frac{N-c}{\xi-c}$ out of M counters, which is equal to $\frac{1}{M}\frac{N-c}{\xi-c}$. In the general case of K stages per filter, this probability is generalized to $P \leq (\frac{1}{M}\frac{N-c}{\xi-c})^K$.

Note that the lemma provides an upper bound and the probability of false positives is much smaller in practice. Nevertheless, from the formula we can infer that be increasing the number of counters M or the threshold ξ, the probability for false positives decreases. It is also apparent that increasing the number of stages K per filter will have a negative impact. Both observations are validated by our experimental results (section 5).

4 Discussion

First we comment on the advantages and disadvantages of each of the three grid-based techniques. Then, we consider methods for eliminating false positives.

4.1 Comparison of Grid-Based Algorithms

The coarse grid uses a simple idea to compress the space-time grid. Since the compression process is independent of the actual cell densities, the number of false positives will increase substantially for large granularities. Furthermore, since only one density counter is kept per bucket, the estimated densities will not be very accurate either.

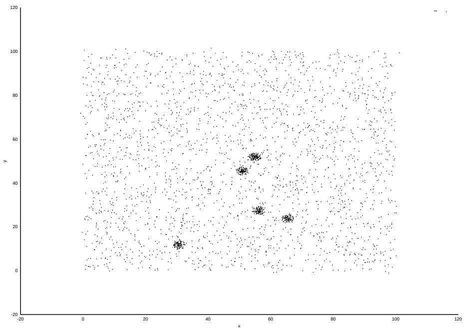
An advantage of Lossy Counting is that (if certain conditions are met) it computes the estimated densities with great accuracy, since the most dense cells are always stored in memory. However, it provides no guarantees about the number of false positives in the answer. Moreover, the estimator needs to be rebuilt frequently so that it does not miss actual dense cells. Since a large number of cells are added and dropped at the bucket boundaries continuously (i.e. these cells are considered by the algorithm to be almost frequent but not frequent enough) their estimated frequencies have a higher error. If the user threshold is small enough to be close to the density of these cells, the algorithm will yield a higher number of false positives with an increased estimation error. Finally, the algorithm has a substantial computational overhead since it has to perform one linear scan of the in memory list per bucket boundary. The cost will increase, of course, when the list is larger (for smaller user thresholds or a multitude of dense cells).

In contrast, the Dense Cell Filter provides strong guarantees on the number of false positives and can be easily adjusted according to space requirements. Moreover, when dense cells are identified they are added in the DCL and their estimated densities are computed with high accuracy. Another benefit is that the algorithm is very efficient with an expected small computational overhead. A drawback of Dense Cell Filter is that the threshold cannot be dynamically set (the lowest possible threshold has to be decided at construction time).

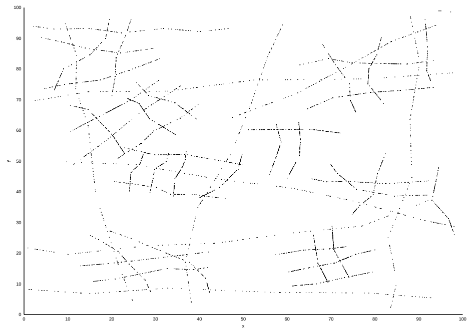
4.2 False Positives

One approach to minimize false positives is by using a spatio-temporal histogram. Such histogram provides an estimate of the selectivity of each reported cell. Unfortunately, recently proposed techniques [27, 8] do not provide any guarantees on the selectivity estimates they report. An alternative is the use of *sketching* (proposed in [14] and based on the seminal work of [4]). Sketches can be used to approximate the spatio-temporal grid by summarizing the information associated with the cells. The advantage of sketching is that the estimation is highly accurate with high probability. The disadvantage is that they are computationally expensive.

If the objective is the full elimination of false positives the user may run a spatio-temporal range query using the spatial range of each cell being reported by the estimators. Such a query finds the actual objects



(a) DENSE



(b) ROAD

Figure 7: Datasets.

in that cell (i.e., an exact answer). Indexing techniques for moving points can be used for that purpose (the TPR-tree [22, 21] the duality indexing of [17] or the partitioning schemes in [11]). Those indices are dynamically updated and index the whole moving dataset. While a range query provides the exact answer, it runs at time proportional to the number of objects in the cell (and since these are the denser cells, they will contain a lot of objects).

5 Experimental Results

All experiments were run on an Intel Pentium(R) 4 1.60Ghz CPU with 1Gb of main memory. We generated various synthetic datasets of moving objects. For the first dataset we initially pick a random number of dense locations and place a large number of objects around them (a snapshot is shown in figure 7(a)). We distribute the rest of the objects uniformly in space and let them move freely on the plane. The dataset tries to simulate vehicles that disperse from dense areas. For example, the downtown L.A. area at 3:00pm or a stadium after the end of a game. In the rest, we refer to this dataset as DENSE. The second dataset represents a network of highways and surface streets (denoted as ROAD, figure 7(b)). Each road is represented by a set of connected line segments (231 line segments were generated in total). The 2-dimensional universe for both datasets is 100 miles long in each direction. Every simulation lasts for 100 time instants. We generated 1 million moving objects per dataset. Every time instant at least 1% of the objects issue an update (which changes the speed and/or direction of the object). The velocities of the vehicles are generated using a skewed distribution, between 10 and 110 miles per hour.

For our measurements we used a 250×250 uniform grid; this gives 62,500 cells in total, while every cell is 0.4 miles wide. The horizon was fized at the start of each simulation. As time proceeds, queries refer to time instants that fall inside the horizon interval. For example, if the horizon was set to $H_f = 20$ and the current time is $T_{now} = 9$, the estimators provide answers for queries that refer to time instants between 9 and 20. When the current time reaches halfway through the horizon. all estimators are automatically rebuild.This is typical for horizon-based solutions [22, 27, 8].

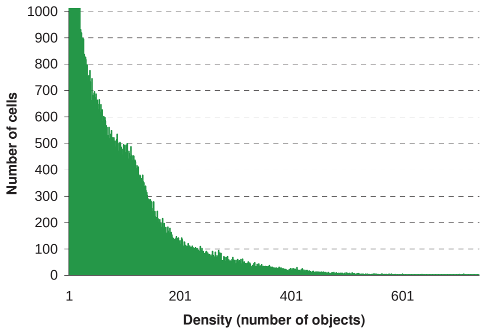


Figure 8: A horizon density histogram.

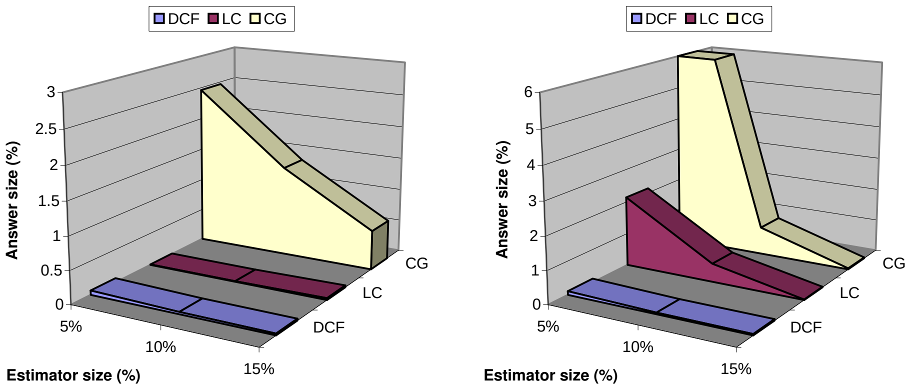
Figure 8 shows an actual density histogram from one of our experiments. It plots the number of cells that have a specific density between time instants 9 and 20 for the ROAD dataset, given the current information at time $T_{now} = 9$ and $H_f = 20$. Since there are 12 time instants until the end of the horizon and each instant carries a grid with 62,500 cells, there are 750,000 cells in that plot. A good estimator should try to compress this information by keeping only the most dense cells which are, typically, very few.

We also created synthetic query workloads. For each experiment we set a fixed density threshold, both for period and snapshot queries. Each experiment is run multiple times varying the density threshold from 1000 to 2500 objects. For snapshot queries the time predicate is uniformly distributed inside the remaining period of the current horizon. For every simulation we run 200 queries in total.

We compare 3 techniques: Dense Cell Filters (denoted as DCF), Lossy Counting (LC), and coarse grids (CG). We tested every technique using several configurations in order to "optimally" tune their performance. We run DCF with several different combinations of stages and counters, LC with varying bucket widths and CG with decreasing granularities. In the rest of this section for every technique we plot the best result given among all configurations with similar size. To compare the efficiency of our techniques we use the *answer size* as a measure. We define the answer size as the total number of false positives plus the actual dense cells reported by each estimator. In addition, we compute the answer size as a percentage of the total number of cells in the spatio-temporal grid.

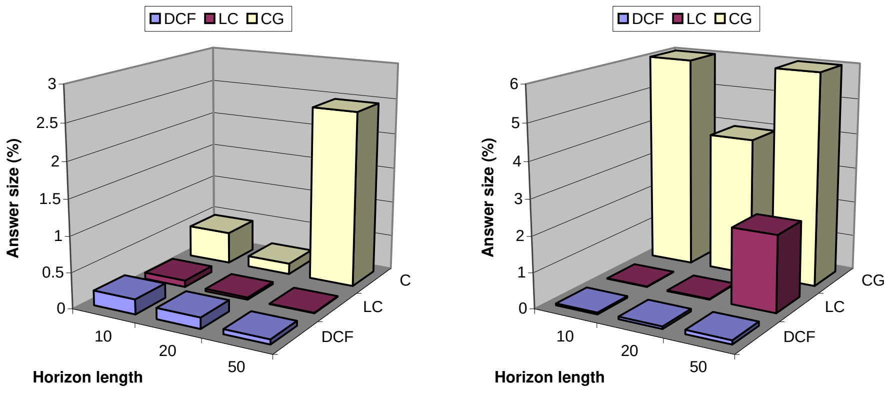
The efficiency of the techniques as a function of the estimator size for snapshot queries is shown in figure 9. The size of the estimators is computed as a percentage of the size of the entire spatio-temporal grid. For this graph we used a density threshold equal to 1500 objects and a fixed horizon of 50 time instants (i.e., 3,125,000 cells). Assuming 4 bytes per cell, the 5% estimator uses only 600KB of main memory. We can deduce that for DCF and LC a 5% estimator size yields very few false positives (less than 2% answer size). Increasing their size beyond 5% does not yield substantial improvements, especially for DCF. On the other hand, CG benefits even for sizes greater than 10%. The CG estimator in order to reduce the grid to 5% of its original size has to become very coarse thus, its performance deteriorates due to very large bucket lengths. For the rest of the graphs we compare all techniques using estimators with 5% size (to be fair, we used 2 bytes per counter for DCF since this is enough to represent a maximum reasonable threshold).

Figure 10 shows a scale-up experiment for increasing horizon lengths. We tested all estimators with



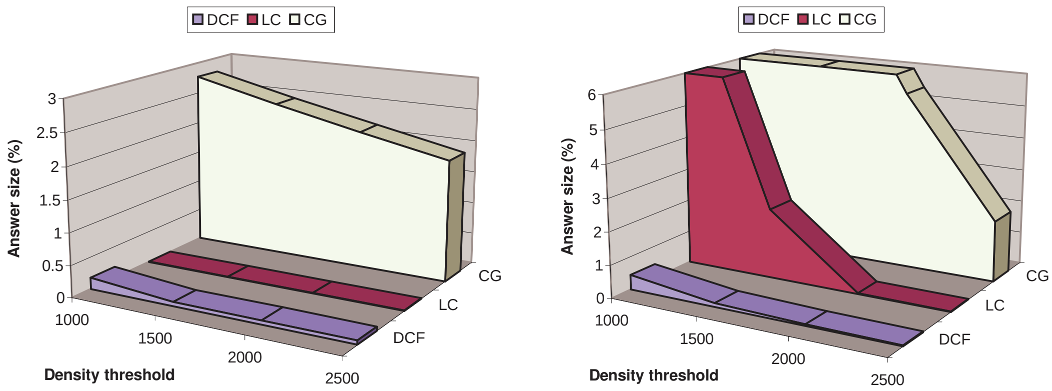
(a) DENSE (b) ROAD

Figure 9: Performance evaluation as a function of estimator size for snapshot queries.



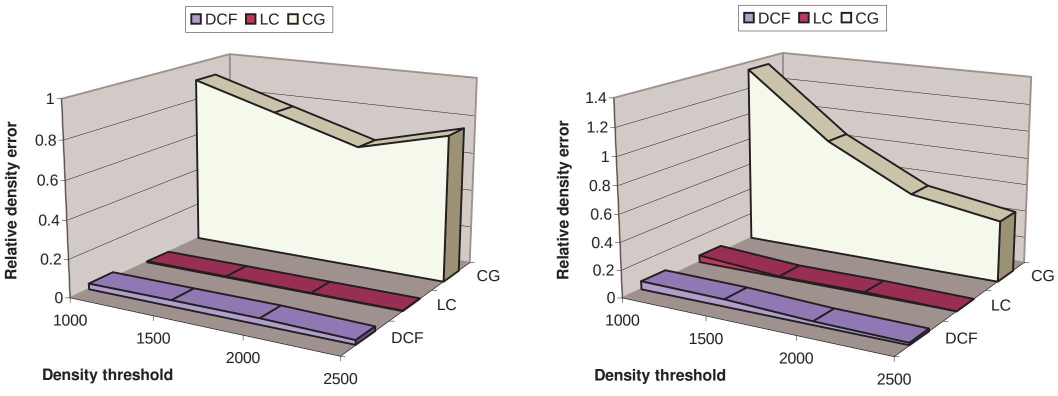
(a) DENSE (b) ROAD

Figure 10: Performance evaluation as a function of horizon length for snapshot queries.



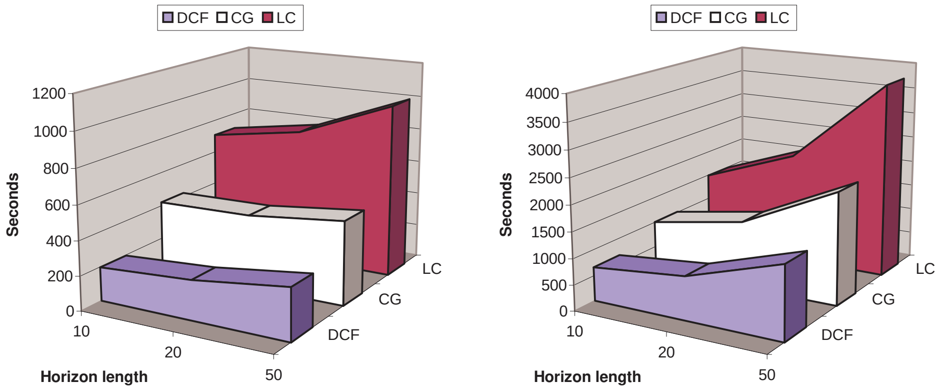
(a) DENSE (b) ROAD

Figure 11: Performance evaluation as a function of density threshold for snapshot queries.



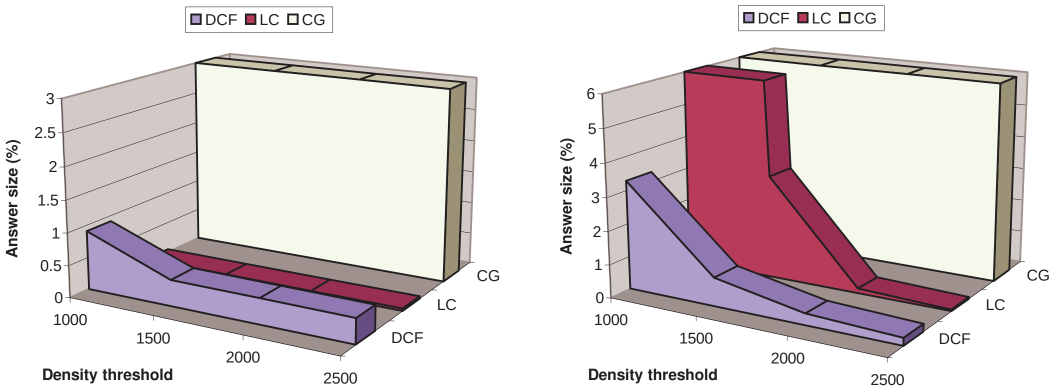
(a) DENSE (b) ROAD

Figure 12: Relative density error as a function of density threshold for snapshot queries.



(a) DENSE (b) ROAD

Figure 13: Estimator update cost as a function of horizon length.



(a) DENSE (b) ROAD

Figure 14: Performance evaluation as a function of density threshold for period queries.

the SDQ query using horizons of 10, 20 and 50 time instants, while keeping the estimator sizes within 5% of the total number of cells. The DCF performance remains unaffected; since a separate filter is kept per time instant of the horizon, the horizon length should not affect the accuracy in general. It is apparent that LC is robust for uniform data, but its performance deteriorates for highly skewed distributions. The LC performance improves for the DENSE dataset but deteriorates for the ROAD dataset. Also, a much larger bucket size was needed in order to keep the estimator within the 5% limit, without loosing some of the correct answers. For CG there is a noticeable performance penalty for both datasets. In order to keep the estimator size within our space requirements the grid granularity has to become very coarse. Moreover, this technique is very fluctuant, which makes it even less attractive.

In figure 11 we plot efficiency as a function of density threshold. For this experiment we used the SDQ query and a horizon length of 50 time instants. As expected, the larger the density threshold the fewer false positives are reported. If there is only a small number of dense cells at a specific time instant, the estimators identify them with great accuracy. The problem becomes more difficult as the number of cells that have densities around that threshold becomes larger. For thresholds larger than 2000 objects all estimators report very few false positives. For the DENSE dataset DCF and LC give the best results (almost no false positives are reported) while CG does not perform well. Performance deteriorates though for LC, especially for smaller thresholds. For the ROAD dataset, DCF gives again the best results; it reports less than 1% of the total number of cells in all cases. Even when exact density answers are required, it is obvious that there is a substantial benefit when using this technique as a pre-filtering step to reduce the amount of cells that need to be checked.

Figure 12 plots the relative error in computing the exact density for the dense cells (averaged over all queries), as a function of density thresholds. All techniques report the densities of the dense cells with less than 1% error (over the exact density of that cell). The results reported by DCF and LC were very close to the actual densities. CG was considerably worse, but still below 1% error.

Figure 13 shows the update cost of the techniques as a function of horizon length. For DCF and CG the update cost does not grow substantially (CG is almost two times more expensive than DCF). The cost for LC increases proportionally for larger horizon lengths, due to the larger bucket widths needed. Moreover, LC is about three times slower than DCF.

Figure 14 plots the estimator performance for period queries (PDQ) as a function of density threshold. We observe the same trends as with snapshot queries. DCF is the best compromise between accuracy, speed and robustness for increasing density thresholds and highly skewed data (less than 3% answer size for all cases). LC works very well for the DENSE dataset, but its performance deteriorates for the ROAD dataset and small density thresholds. CG, on the other hand, reported more than 8% false positives for all cases.

From our extensive experimental evaluation, DCF presented the most robust performance making it a quite accurate, fast and reliable density estimation technique. LC gives very good density estimates but has much higher update cost and does not work well for highly skewed environments.

6 Conclusions

We addressed the problem of on-line discovery of dense areas in spatio-temporal environments. We simplified the problem by dividing the spatial universe into a uniform grid of cells and considering a fixed horizon in the time domain. We proposed efficient solutions that provide fast answers with high accuracy. Our solutions are based on Dense Cell Filters and Approximate Frequency counting algorithms. They guarantee no false dismissals (but few false positives), provide fast updates and are space efficient. An extensive experimental evaluation was also presented, showing that the Dense Cell Filter was the most robust solution. It gave fast and accurate answers, with a minimal number of false positives. In future work we will address Top-K density-based queries. Instead of specifying a density threshold the query must report the k most dense cells. Methods based on histograms [19, 28] or sampling [13, 7] should be considered.

References

[1] P. Agarwal, L. Arge, and J. Vahrenhold. Time responsive indexing schemes for moving points. In *Proc. of WADS*, 2001.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *In Proc. of the 19th ACM Symp. on Principles of Database Systems (PODS)*, pages 175–186, 2000.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. *In Proc. of ACM SIGMOD Conference*, pages 94–105, June 1998.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Journal of Computer and System Sciences*, volume 58(1), pages 137–147, 1999.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *To appear in Allerton 2002.*

[7] C.-M. Chen and Y. Ling. A sampling-based estimator for Top-k query. In *Proc of IEEE ICDE*, 2002.

[8] Yong-Jin Choi and Chin-Wan Chung. Selectivity estimation for spatio-temporal queries to moving objects. In *Proc. of ACM SIGMOD*, 2002.

[9] H. D. Chon, D. Agrawal, and A. El Abbadi. Storage and retrieval of moving objects. In *Mobile Data Management*, pages 173–184, 2001.

[10] C. Jensen (editor). Special issue on indexing moving objects. *Data Engineering Bulletin*, 2002.

[11] K. Elbassioni, A. Elmasry, and I. Kamel. An efficient indexing scheme for multi-dimensional moving objects. *9th International Conference on Database Theory, Siena, Italy (to appear)*, 2003.

[12] L. Fan, J. Almeida P. Cao, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[13] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. of ACM SIGMOD*, April 1998.

[14] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *The VLDB Journal*, September 2001.

[15] O. Ibarra H. Mokhtar, J. Su. On moving object queries. In *Proc. 21st ACM PODS Symposium on Princeples of Database Systems, Madison, Wisconsin*, pages 188–198, 2002.

[16] G. Kollios, D. Gunopulos, and V. Tsotras. Nearest Neighbor Queries in a Mobile Environment. *In Proc. of the Spatio-Temporal Database Management Workshop, Edinburgh, Scotland*, pages 119–134, 1999.

[17] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. *In Proc. of the 18th ACM Symp. on Principles of Database Systems (PODS)*, pages 261–272, June 1999.

[18] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. of 28th VLDB*, pages 346–357, August 2002.

[19] S. Chaudhuri N. Bruno and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2), 2002.

[20] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proc. of 7th SSTD*, July 2001.

[21] S. Saltenis and C. Jensen. Indexing of Moving Objects for Location-Based Services. *Proc. of IEEE ICDE*, 2002.

[22] S. Saltenis, C. Jensen, S. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *In Proceedings of the ACM SIGMOD*, pages 331–342, May 2000.

[23] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. *In Proceedings of the 13th ICDE, Birmingham, U.K*, pages 422–432, April 1997.

[24] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proc. of the SSTD*, pages 79–96, 2001.

[25] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. *Proc. of ACM SIGMOD*, 2002.

[26] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. of VLDB*, 2002.

[27] Y. Tao, J. Sun, and D. Papadias. Selectivity estimation for predictive spatio-temporal queries. *Proceedings of 19th IEEE International Conference on Data Engineering (ICDE), to appear*, 2003.

[28] M. Wang, J.S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proc. of SSTD*, pages 175–196, 2001.

[29] W. Wang, J. Yang, and R. Muntz. STING: A statistical information grid approach to spatial data mining. In *The VLDB Journal*, pages 186–195, 1997.