Geoinformatica (2009) 13:355–396
DOI 10.1007/s10707-008-0050-7

Efficient MaxCount and threshold operators of moving objects

Scot Anderson · Peter Revesz

Received: 31 July 2007 / Revised: 15 February 2008 /
Accepted: 11 April 2008 / Published online: 30 May 2008

Abstract Calculating operators of continuously moving objects presents some unique challenges, especially when the operators involve aggregation or the concept of congestion, which happens when the number of moving objects in a changing or dynamic query space exceeds some threshold value. This paper presents the following six *d*-dimensional moving object operators: (1) MaxCount (or MinCount), which finds the Maximum (or Minimum) number of moving objects simultaneously present in the dynamic query space at any time during the query time interval. (2) CountRange, which finds a count of point objects whose trajectories intersect the dynamic query space during the query time interval. (3) ThresholdRange, which finds the set of time intervals during which the dynamic query space is congested. (4) ThresholdSum, which finds the total length of all the time intervals during which the dynamic query space is congested. (5) ThresholdCount, which finds the number of disjoint time intervals during which the dynamic query space is congested. And (6) ThresholdAverage, which finds the average length of time of all the time intervals when the dynamic query space is congested. For these operators separate algorithms are given to find only estimate or only precise values. Experimental results from more than 7,500 queries indicate that the estimation algorithms produce fast, efficient results with error under 5%.

Keywords Moving point objects · Spatiotemporal data · Aggregation operator · MaxCount · ThresholdCount · CountRange

S. Anderson
Southern Adventist University, Collegedale, TN, USA
e-mail: sanderson@southern.edu

P. Revesz (✉)
University of Nebraska—Lincoln, Lincoln, NE, USA
e-mail: revesz@cse.unl.edu

1 Introduction

Safety can often be reduced to a problem of congestion. The safety of flight depends on separation of airplanes or more generally the maximum number of airplanes that a particular airspace can safely contain, and the maximum number of airplanes that air traffic controllers responsible for directing airplanes can safely track. When considering epidemics, the presence of a single animal with Bird Flue does not indicate the start of an epidemic. Instead the presence of a certain number of instances of the disease indicates a high risk of starting an epidemic, or actual epidemic conditions. Consequently, we see that congestion often links to safety and can predict high risk or even dangerous conditions.

Congestion is defined differently depending on the application. Hence it is necessary to provide moving object operators that take a threshold value as a parameter to define congestion.

In relational databases, MAX, MIN, COUNT, SUM and AVERAGE form the set of natural aggregation operators. Spatiotemporal databases containing continuously moving objects can not apply these operators in the same way. However, these operators may still function in interesting ways for moving objects. For example, one can ask how many moving point objects exist within a moving and changing (or *dynamic*) rectangular area *at a certain time*, or what is the maximum distance between two moving points *during a time interval*. Obviously, when we are interested in discrete time instances, then the moving point object database can be reduced to a relational database and the above queries can be expressed as simple COUNT or MAX queries.

Moving object databases naturally suggest new aggregate operators that have no equivalents in relational databases. For example, one may ask what is the maximum number of moving-point objects that exist simultaneously within a dynamic rectangular area at any time during a time interval T? We call this the MAXCOUNT operator. One also may ask during what time intervals in T does there exist more than M moving objects within a rectangular area? We call this the THRESHOLDRANGE operator. We show that a strong relationship exists between MAXCOUNT and THRESHOLDRANGE, and we show that THRESHOLDRANGE forms the bases for a family of threshold operators that include: THRESHOLDCOUNT, THRESHOLDSUM, and THRESHOLDAVERAGE. A related, though less complex, operator answers the question: what is the number of moving objects that exist within or intersect a dynamic rectangular area at any time instance during interval T. We call this the COUNTRANGE operator.

Throughout this paper we use the concept of a dynamic *query space*, which is a 3-dimensional space that may move and change size or shape over a continuous time interval T. In most papers, including ours, the query space is considered to be some cube, whose corner vertices are defined by two moving points. (We define query space more precisely later in Definition 26.) The moving object operators described in this paper are defined as follows:

Definition 1 (MAXCOUNT (MINCOUNT)) Let S be a set of moving points. Given a dynamic query space R defined by two moving points Q_1 and Q_2 as the lower-left and upper-right corners of R, and a time interval T, the MAXCOUNT (*Min-Count*) operator finds the time $t_{\max(\min)}$ and maximum (or minimum) number of points $M_{\max(\min)}$ in S that R can contain at any time instance within T.

Definition 2 (CountRange) Let S be a set of moving points. Given a dynamic query space R defined by two moving points Q_1 and Q_2 as the lower-left and upper-right corners of R and a time interval T, the CountRange query returns the total number of points that intersect R in T.

Definition 3 (ThresholdRange) Let S be a set of moving points. Given a dynamic query space R defined by two moving points Q_1 and Q_2 as the lower-left and upper-right corners of R, a time interval T, and a threshold value M, the ThresholdRange operator finds the set of time intervals T_M where the count of objects in R is larger than M.

Definition 4 (ThresholdCount) Let M and T_M be as in ThresholdRange. Then ThresholdCount returns the number of time intervals in T_M.

Definition 5 (ThresholdSum) Let M and T_M be as in ThresholdRange. Then ThresholdSum returns the total length of time T_s during which the count is above M. That is, for each $T_i \in T_M$, ThresholdSum returns:

$$T_s = \sum_i |T_i| \tag{1}$$

where $|T_i|$ means the length of the interval T_i.

Definition 6 (ThresholdAverage) Let M and T_M be as in ThresholdRange. Then ThresholdAverage returns the average length of the intervals in T_M.

The following examples illustrate the use of the new moving object operators.

Example 7 Airplanes are commonly modeled as linearly moving objects with preestablished flight plans. Suppose, at any time, at most a constant number M of airplanes is allowed to be in the O'Hare airspace to avoid congestion. Suppose also a new airplane requests approval of its flight plan for entering the O'Hare airspace between times t_a and t_b. The air traffic controllers can avoid congestion as follows. If after adding a new flight plan, the MaxCount between t_a and t_b is still less than M, then they can approve the flight. Otherwise, they need to find some alternative path, and check it again against the database.

Air traffic controllers try to direct airplanes as linearly moving objects for fuel efficiency, among other reasons. If they recognize a developing congestion too late, then they often must direct the airplane to fly in circles until the congestion has cleared. That solution wastes fuel. On the other hand, if they recognize the developing congestion early, then they can often simply tell the airplane to change its speed, which saves fuel. Therefore, it is important to identify congestions as early as possible. We may identify congestions by using a MaxCount query where a moving box around the airplane and a time interval $[t_a, t_b]$ define the query. If the MaxCount predicts congestion, then the airplane's speed can be adjusted early in the flight.

Example 8 Suppose we want to alert pilots if their current flight path takes them through at least one congested region.

A *Traffic Alert/Collision Avoidance Systems* is a system that provides similar functionality. TCASs only provide alerts for current congestion, not predictive congestion. Although TCASs were implemented in 1986, we continue to have mid-air collisions and near misses indicating that the system still needs improvement. ThresholdRange is a modification of MaxCount that returns all predicted time intervals on the flight path where the Count exceeds a given threshold. Hence using ThresholdRange we can alert a pilot of predicted congestions where more than M other airplanes will be within the space B around the airplane. Predicting and avoiding these areas can significantly reduce the chances of mid-air collisions.

Example 9 Suppose we are especially concerned about a rush-hour period $[t_a, t_b]$ that is particularly stressful to air traffic controllers. Suppose controllers can direct at most M airplanes safely. We can determine the number of controllers needed during the rush-hour time by executing the CountRange query over the controlled airspace during the rush-hour and dividing by M. By ensuring that a sufficient number of controllers are present, safety is achieved and controllers are not over stressed.

The rest of this paper is organized as follows. Section 2 introduces novel data structures used to build buckets. These buckets can then be used in various indexing algorithms to fit the type of application used. Section 3 develops the MaxCount estimation algorithm using a running example. Section 4 develops algorithms for the CountRange and the threshold operators. Both CountRange and ThresholdRange are based on MaxCount. Based on ThresholdRange, algorithms are given for the other threshold operators ThresholdCount, ThresholdSum, and ThresholdAverage. Section 5 gives the experimental results of the implementation. Section 6 reviews the related work. Finally, Section 7 suggests some future work and conclusion.

2 Bucket data structures

This section presents an updatable *skew-aware* bucket for indices that models the skewed point distributions in each bucket. The skew-aware technique allows the index structure to perform inserts, deletes, and updates in *fast constant time* using a HashTable to store the buckets. Many spatiotemporal applications, such as tracking clients on a wireless network, particularly need these fast updates and no other MaxCount presented prior to this can meet that requirement. Because the buckets are spatially defined, the bucketing technique also easily adapts to other spatial and spatiotemporal indices such as the R-tree [13]. Hence the technique performs well for applications where search operations or update operations occur more frequently by using an appropriate index.

Our algorithm uses a sweeping method to evaluate the threshold aggregation operators similar to previous approaches from Chen and Revesz [5], Revesz and Chen [25] and Anderson [1]. The algorithm differs in that the sweeping algorithm integrates a skew-aware density function over the spatial dimensions of the bucket to obtain the time dependent count function. The density function in the bucket increases accuracy over methods given in Anderson [1], Chen and Revesz [5] while maintaining the same number of buckets. This idea is a crucial improvement because we model the point distribution skew in a bucket, whereas previous methods adapted to skew by increasing the number of buckets or changing their shape and contents. We also present a precise algorithm for evaluating the threshold aggregation operators that requires no index and runs in $O(N) + O(n \log n)$ time and $O(n)$ space where N is the number of points in the database and n is the value of a CountRange query using the same query space and time. Both the threshold aggregation algorithms and the skew-aware bucket data structure presented are implemented and analyzed in 3-dimensional space. We show that the approximation achieves good results while significantly reducing the running times.

Section 2.1 describes the problems related to creating buckets and a specific solution for creating 6-dimensional buckets for 3-dimensional linearly moving points. In all cases, we can extend our method to d-dimensions. Section 2.2 describes the method for inserting and deleting a point from a bucket and shows that updates take constant time. Section 2.3 applies two different data structures to contain the buckets suited for applications where either inserts and deletes or threshold operations dominate.

2.1 Bucket data structure

Definition 10 (Hex Representation) Given a linearly moving point in three dimensions

$$Q(t) = \begin{cases} q_x = v_x t + x_0 \\ q_y = v_y t + y_0 \\ q_z = v_z t + z_0 \end{cases} \tag{2}$$

the corresponding *hex representation* of $Q(t)$ is the tuple $(v_x, x_0, v_y, y_0, v_z, z_0)$.

If we divide the 6-dimensional space into axis-aligned hyper-rectangles, then each hyper-rectangle becomes a *bucket* B_i that contains the moving points whose hex representations fall inside the hyper-rectangle. For each such bucket B_i, we define the following.

Definition 11 (Bucket Dimension) Given a bucket B_i, its dimensions can be described by inequalities of the form:

$$\begin{aligned} &v_{x,L_i} \le v_x < v_{x,U_i} \bigwedge x_{0,L_i} \le x_0 < x_{0,U_i} \bigwedge \\ &v_{y,L_i} \le v_y < v_{y,U_i} \bigwedge y_{0,L_i} \le y_0 < y_{0,U_i} \bigwedge \\ &v_{z,L_i} \le v_z < v_{z,U_i} \bigwedge z_{0,L_i} \le z_0 < z_{0,U_i} \end{aligned} \tag{3}$$

where the subscripts with L indicate lower and the subscripts with U indicate upper bound constants. Similarly to the hex representation of Definition 10, we denote the bucket lower bound as:

$$\left(v_{x,L_i}, x_{0,L_i}, v_{y,L_i}, y_{0,L_i}, v_{z,L_i}, z_{0,L_i}\right) \tag{4}$$

Table 1 The hex representation of a set of moving points

v_x	x_0	v_y	y_0	v_z	z_0
5.345	7.543	5.345	8.158	5.345	5.488
6.354	9.023	6.354	5.488	6.354	5.159
7.159	8.885	7.159	6.685	7.159	7.346
7.645	9.117	7.645	5.159	7.645	8.885
8.153	7.346	8.153	6.335	8.153	7.543
8.156	6.335	8.156	7.346	8.156	9.023
9.125	5.159	9.125	9.117	9.125	9.117
9.118	6.685	9.118	8.885	9.118	6.335
9.688	5.488	9.688	9.023	9.688	8.158
9.874	8.158	9.874	7.543	9.874	6.685

and the bucket upper bound as:

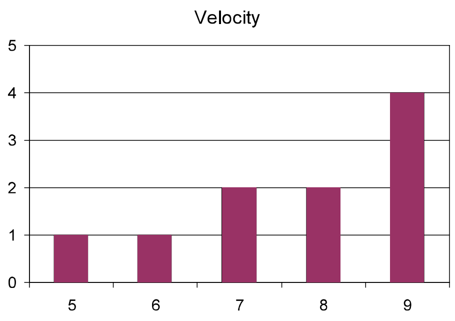
$$\left(v_{x,U_i}, x_{0,U_i}, v_{y,U_i}, y_{0,U_i}, v_{z,U_i}, z_{0,U_i}\right). \tag{5}$$

A pair of lower and upper bounds is a concise description of the dimensions of B_i.

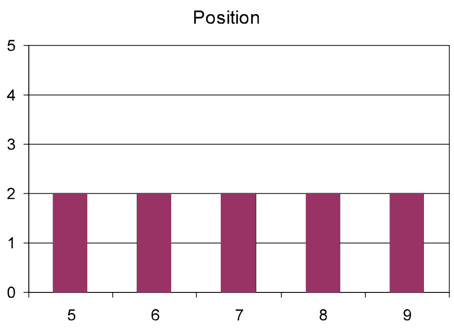
Definition 12 (Histogram) Given a bucket B_i and a constant s, we build *histograms* $h_{i,1}$,...,$h_{i,6}$. Each histogram $h_{i,j}$ is built using the following three steps: (1) Project the points in bucket B_i onto the jth dimension. (2) Subdivide the projection into s equal-size intervals. (3) Record separately the number of points within each subdivision.

Example 13 Supose bucket B_i has lower bound $(5, 5, 5, 5, 5, 5)$ and upper bound $(10, 10, 10, 10, 10, 10)$ and contains the points shown in Table 1. Also suppose the number of subdivisions is $s = 5$.

Figure 1a shows $h_{i,1}$. It is built as follows. The projection onto velocity v_x is the first column of Table 1. Since in the v_x dimension bucket B_i ranges from 5 to 10, the five subdivisions are $5 < v_x \leq 6$, $6 < v_x \leq 7$, $7 < v_x \leq 8$, $8 < v_x \leq 9$, and $9 < v_x \leq 10$. The first bar of histogram $h_{i,1}$ rises to level 1 because the first subdivision $5 < v_x \leq 6$



(a) $h_{i,1}$: Histogram for points projected onto v_x.



(b) $h_{i,2}$: Histogram for points projected onto x_0.

Fig. 1 Two histograms of the points in B_i. The x-axis is the subdivision left-endpoint and the y-axis is the number of points

contains one point. The second bar of histogram $h_{i,2}$ rises to level 1 because the second subdivision $6 < v_x \leq 7$ also contains one point. The other values of $h_{i,1}$ can be determined similarly.

Figure 1b shows $h_{i,2}$. It is built using the projection onto position x_0, which is the second column of Table 1. Here we use the subdivision $5 < x_0 \leq 6, \ldots, 9 < x_0 \leq 10$. In this case the first bar of histogram $h_{i,2}$ rises to level 2 because the subdivision $5 < x_0 \leq 6$ contains two points. The other values of $h_{i,2}$ can be determined similarly.

Further, in this example $h_{i,1} = h_{i,3} = h_{i,5}$ because all the points listed in Table 1 have the same velocities in the x, y, and z directions. Also, $h_{i,2} = h_{i,4} = h_{i,6}$ because the columns for x_2, x_4, and x_6 all have the same values but in different orders.

Definition 14 (Trend Function) Given a bucket B_i and axis j, the *axis trend function* $f_{i,j}$ is some polynomial function such that the following hold:

1. $f_{i,j} \geq 0$ over the j-dimension of B_i.
2. $f'_{i,j}$, the derivative $f_{i,j}$, does not change sign over the j-dimension of B_i.

The *bucket trend function* f_i for bucket B_i is the product of the axis trend functions, that is:

$$f_i = \prod_{j=1}^{6} f_{i,j} \tag{6}$$

Condition 1 ensures that the bucket trend function, built from the axis trend functions, does not contain a negative probability region. Condition 2 requires that the bucket density increase, decrease, or remain constant when considering any single axis.

Example 15 Histograms $h_{i,1}$ and $h_{i,2}$ in Example 13 can be represented as the following set of points:

$$h_{i,1} = \{(5,1), (6,1), (7,2), (8,2), (9,4)\} \tag{7}$$

$$h_{i,2} = \{(5,2), (6,2), (7,2), (8,2), (9,2)\} \tag{8}$$

where the first coordinate is v_x in $h_{i,1}$ and x_0 in $h_{i,2}$, and the second coordinate is y, the number of points.

To find a trend function, we can choose a least squares method to fit each of these histograms to a line. The least square line for $h_{i,1}$ is:

$$y = 0.7v_x - 2.9$$

The least square line for $h_{i,2}$ is:

$$y = 0x_0 + 2$$

We need to ensure that Condition 1 in Definition 14 holds. A simple way to ensure that is to find the minimum of the line in the range of the histogram and if it is a negative value, then shift the line upward by that amount. For a line segment the minimum occurs always at one of the end points. Taking the minimum of the lease square line equations at the histogram range end points 5 and 10 gives for $h_{i,1}$ the value $\min(y(5) = 1,\ y(10) = 4.3) = 4.3$, and for $h_{i,2}$ the value $\min(y(5) = 2,$

$y(10) = 2) = 2$. Since in this case Condition 1 is satisfied by the least square lines, they do not need to be shifted upward. Condition 2 also is satisfied by line segments. Hence the least square lines can be taken as axis trend functions.

Further, since in Example 13 $h_{i,1} = h_{i,3} = h_{i,5}$ and $h_{i,2} = h_{i,4} = h_{i,6}$ the other axis trend functions can be found similarly to those of $h_{i,1}$ and $h_{i,2}$. Hence the trend function becomes:

$$\begin{aligned} f_i &= (0.7v_x - 2.9)(0x_0 + 2)(0.7v_y - 2.9)(0y_0 + 2)(0.7v_z - 2.9)(0z_0 + 2) \\ &= 8(0.7v_x - 2.9)(0.7v_y - 2.9)(0.7v_z - 2.9) \end{aligned}$$

Note: Normally the least square line does not have a negative value in the range of the histogram. If it is negative, it is an indication that the distribution of points in the bucket is unusual. In that case the bucket may be broken into smaller ones, or we may try to find another polynomial axis trend function.

The next lemma considers some properties of axis trend functions in general.

Lemma 16 *Given a bucket B_i with axis trend functions $f_{i,j}$, let r_1 and r_2 be identically sized regions in bucket B_i. If the density in B_i along each axis monotonically increases from r_1 to r_2, then the following holds:*

$$\int_{r_2} f_i \, d\phi \geq \int_{r_1} f_i \, d\phi \tag{9}$$

Proof Increasing densities from r_1 to r_2 translates into histograms that also increase from r_1 in the direction of r_2 along each axis. The translation from histograms to the axis trend functions gives the following conditions:

$$f_{i,j}(x_{2,j}) \geq f_{i,j}(x_{1,j}) \tag{10}$$

where $x_{1,j}$ and $x_{2,j}$ are the j^{th} coordinates of the points in r_1 and r_2 respectively, and are located the same distance from the j^{th} coordinates of the lower bounds of r_1 and r_2 respectively. Since this constraint holds for each j and $f_{i,j} \geq 0$ we have:

$$f_i(x_2) \geq f_i(x_1) \tag{11}$$

Hence by the properties of integration we conclude

$$\int_{r_2} f_i \, d\phi \geq \int_{r_1} f_i \, d\phi \tag{12}$$

□

Note: Definition 14 and Lemma 16 allow other polynomial functions beside the least square line. In Example 15, we chose the least square line only as a simple example. As the experimental results in Section 5 show, the least square line is a good practical approximation. The approximation works well because the buckets have small sizes. Adjacent buckets can have very different trend functions. While the entire range of the space in which all the points lie would be poorly approximated by a single least square line, the series of least square lines associated with a sequence of small buckets in each spacial dimension serves as a good piecewise linear approximation.

In the following, we assume that the trend functions are polynomial functions derived from the product of linear functions, which are obtained by using the least squares method for each histogram as in Example 15.

Definition 17 (Normalized Trend Function) Let n be the number of points in the database, b_i the number of points in bucket B_i, and f_i be given by Eq. 6. The *normalized trend function* F_i for bucket B_i is:

$$F_i = \frac{b_i f_i}{n \int_{B_i} f_i \, d\phi} \tag{13}$$

and the *percentage of points* in bucket B_i is:

$$p = \int_{B_i} F_i \, d\phi. \tag{14}$$

Example 18 Assume that bucket B_i in Example 13 is only a part of an index that contains $n = 100{,}000$ points. Then calculating F_i from Eq. 13 requires integrating f_i over the bucket dimensions where $\int_{B_i} \equiv \int_5^{10} \ldots \int_5^{10}$ and where $d\phi \equiv dv_x dx_0 dv_y dy_0 dv_z dz_0$ rounded to the nearest integer gives:

$$\int_{B_i} f_i d\phi = 8 \int_{B_i} (0.7x_0 - 2.9)(0.7x_2 - 2.9)(0.7x_4 - 2.9) d\phi$$
$$\approx 1{,}622{,}234. \tag{15}$$

In this case the number of points in bucket B_i is $b_i = 10$. Hence we have:

$$\begin{aligned} F_i &= \frac{b_i f_i}{n \int_{B_i} f_i \, d\phi} \\ &\approx \frac{10 \times 8(0.7v_x - 2.9)(0.7v_y - 2.9)(0.7v_z - 2.9)}{100{,}000 \times 1{,}622{,}234} \\ &\approx 49.312 \times 10^{-11}(0.7v_x - 2.9)(0.7v_y - 2.9)(0.7v_z - 2.9) \end{aligned}$$

Using Definition 17, we can calculate the number of points in a bucket in $O(1)$ time using the following lemma.

Lemma 19 *Let B_i be a bucket, n the total number of points in the database, and p be given by Definition* 17. *Then np is the number of points in bucket B_i and np is calculated in $O(1)$ time.*

Proof By Eqs. 13 and 14 we have:

$$\begin{aligned} np &= n \int_{B_i} F_i \, d\phi \\ &= n \int_{B_i} \frac{b_i}{n} \frac{f_i}{\int_{B_i} f_i \, d\phi} d\phi \end{aligned}$$

$$= n\frac{b_i}{n} \cdot \frac{\int_{B_i} f_i \, d\phi}{\int_{B_i} f_i \, d\phi}$$

$$= b_i. \tag{16}$$

Clearly the above calculations take only $O(1)$ time. □

Using the above definitions we can now define the bucket data structure used throughout the rest of this paper.

Definition 20 (Skew Aware Buckets) A *skew aware bucket* is a hyper-rectangle with dimensions given by Definition 11 and that maintains histograms given by Definition 12, additional data for the least squares method, and the normalized trend function given by Definition 17.

Throughout the rest of this paper we refer to skew aware buckets as simply buckets.

2.2 Inserts and deletes

We can maintain the bucket (and hence the index) while deleting or inserting a point for any bucket B_i by recalculating the trend function F_i for the bucket.

Lemma 21 *Insertion and deletion of a moving point can be done in* $O(1)$ *time.*

Proof When we insert or delete a point, we need to update the histograms and the normalized trend function. Let the point to insert/delete be P_a represented using the hex representation as $(a_0, a_1, a_2, a_3, a_4, a_5)$, let d_j, for $0 \leq j \leq 5$ be the bucket width in the j^{th}, and let s be the number of subdivisions in each histogram. The concatenation of $id_0, \ldots, id_5$ gives the ID_i of bucket i to insert (or delete) P_a into where each id_l and $0 \leq l \leq 5$ is defined by:

$$id_l = \left\lfloor \frac{a_l}{d_l} \right\rfloor. \tag{17}$$

The calculation of ID_i and retrieving bucket B_i takes $O(1)$ time using a HashTable.

Let $hw_{i,j}$ be the histogram-division width for the j^{th} calculated as $hw_{i,j} = \left\lceil \frac{d_j}{s} \right\rceil$. Then p is projected onto each dimension to determine which division of the histogram to update. For the j^{th} dimension the k^{th} division of histogram $h_{i,j}$ is given as follows:

$$k(j) = \left\lfloor \frac{a_j - id_j * d_j}{hw_k} \right\rfloor \tag{18}$$

Let $h_{i,j,k}$ be the histogram division to update for each histogram. Update $h_{i,j,k}$ and the sums $\sum y_i$, and $\sum x_i y_i$ from the normal equations in the least squares method. N, $\sum x_i$ and $\sum x_i^2$ from the normal equations do not need updating since the number of histogram divisions s is fixed within the database.

We can now recalculate each $f_{i,j}$ in constant time by solving the 2×3 matrix corresponding to the normal equations of the least squares method for each histogram. For each $f_{i,j}$ calculate the endpoints to determine the required shift amount (Definition 14, property 1) and calculate f_i from Eq. 6. Now we calculate F_i using Eq. 17. Each of these steps depends only on the dimension of the database. Hence for any fixed dimension we can rebuild the normalized trend function F_i in $O(1)$ time. □

2.3 Index data structures

There is no need to create a bucket unless it contains at least one point. We consider two classes of data structures for organizing the buckets: HashTables and Trees.

For databases where inserts and deletes are the most common operation, the HashTable approach allows these operations to run in constant time. However, the MaxCount operation will require an enumeration of all the buckets and thus at least a running time of $O(B)$. As long as the number of buckets is reasonable, this approach works well.

For databases where MaxCount is the most common operation, we may use an R-tree structure [3], [13] where the elements to be inserted are the buckets. This approach speeds up the MaxCount query to $O(\log|B| + R)$ where R is the number of buckets needed to calculate the query. The insert and delete costs for these R-trees are $O(\log|B|)$, because buckets do not overlap.

Since buckets do not change shape, the database is decomposable and allows each type of aggregation to be calculated from simultaneous executions on subspaces of the index space. We discuss the method and ramifications of this capability at the end of Section 3.4.

3 The MaxCount operator

Section 3.1 reviews point domination in higher dimensions. Section 3.2 examines finding the percentage of points in a bucket that are in the query space as a function of time. Section 3.3 puts the two previous sections together to create an efficient approximate MaxCount algorithm for d-dimensions. Section 3.4 describes an inefficient exact MaxCount algorithm. The naive but exact MaxCount algorithm is useful as a comparison tool with the efficient but approximate MaxCount algorithm in Section 3.3. The comparison is presented later in Section 5.

3.1 Review of point domination

Definition 22 (Point Domination) Given two linearly moving points in three dimensions

$$Q(t) = \begin{cases} q_x = v_x t + x_0 \\ q_y = v_y t + y_0 \\ q_z = v_z t + z_0 \end{cases} \quad \text{and} \quad P(t) = \begin{cases} p_x = x_1 t + x_2 \\ p_y = x_3 t + x_4 \\ p_z = x_5 t + x_6 \end{cases} \tag{19}$$

$Q(t)$ dominates $P(t)$ at time t if and only if the following condition holds:

$$p_x < q_x \bigwedge p_y < q_y \bigwedge p_z < q_z \tag{20}$$

Further, the *dominating lines* of $Q(t)$ at time t are the following:

$$l_x \text{ is th; line } x_2 = -x_1 t + (v_x t + x_0).$$

$$l_y \text{ is th; line } x_4 = -x_3 t + (v_y t + y_0).$$

$$l_z \text{ is th; line } x_6 = -x_5 t + (v_z t + z_0).$$

For convenience in visualizing the hex representations, we introduce the following definition.

Definition 23 (x-view, y-view and z-view) The projection of a 6-dimensional space of hex representations onto the first two dimensions is the *x-view*, onto the second two dimensions the *y-view*, and onto the third two dimensions the *z-view*.

It is now easy to prove the following lemma.

Lemma 24 *Let $Q(t)$ and $P(t)$ be moving points, and let l_x, l_y, and l_z be the dominance lines of $Q(t)$ at time t as in Definition 22. Let x-view, y-view, and z-view be the projections of the hex representations of moving points as in Definition 23. Then $Q(t)$ dominates $P(t)$ at time t if and only if (x_1, x_2) lies below l_x in the x-view, (x_3, x_4) lies below l_y in the y-view, and (x_5, x_6) lies below l_z in the z-view.*

Proof We rewrite the condition (20) in Definition 22 as follows:

$$\begin{aligned} & p_x < q_x \bigwedge p_y < q_y \bigwedge p_z < q_z \\ & \quad = x_1 t + x_2 < v_x t + x_0 \bigwedge x_3 t + x_4 < v_y t + y_0 \bigwedge x_5 t + x_6 < v_z t + z_0 \\ & \quad = x_2 < -x_1 t + (v_x t + x_0) \bigwedge x_4 < -x_3 t + (v_y t + y_0) \bigwedge x_6 < -x_5 t + (v_z t + z_0) \end{aligned} \tag{21}$$

Clearly, this condition is satisfied if (x_1, x_2) lies below l_x in the x-view, (x_3, x_4) lies below l_y in the y-view, and (x_5, x_6) lies below l_z in the z-view. □

Example 25 Suppose that in the hex-representation $Q(t) = (4, 6, 5, 6, 9, 10)$. Then $Q(t)$ dominates which points $P(t) = (x_1, x_2, x_3, x_4, x_5, x_6)$ at time $t = 1$?

Applying condition (20) in Definition 22 and substituting we obtain:

$$\begin{aligned} & x_2 < -x_1 t + (v_x t + x_0) \bigwedge x_4 < -x_3 t + (v_y t + y_0) \bigwedge x_6 < -x_5 t + (v_z t + z_0) \\ & \quad = x_2 < -x_1 + 10 \bigwedge x_4 < -x_3 + 11 \bigwedge x_6 < -x_5 + 19 \end{aligned} \tag{22}$$

Hence at time $t = 1$, the point $Q(t)$ dominates those points $P(t)$ which satisfy the above condition.

Unlike in Example 25, in many applications we are interested in two moving points $Q_1(t)$ and $Q_2(t)$, which define the query space as follows.

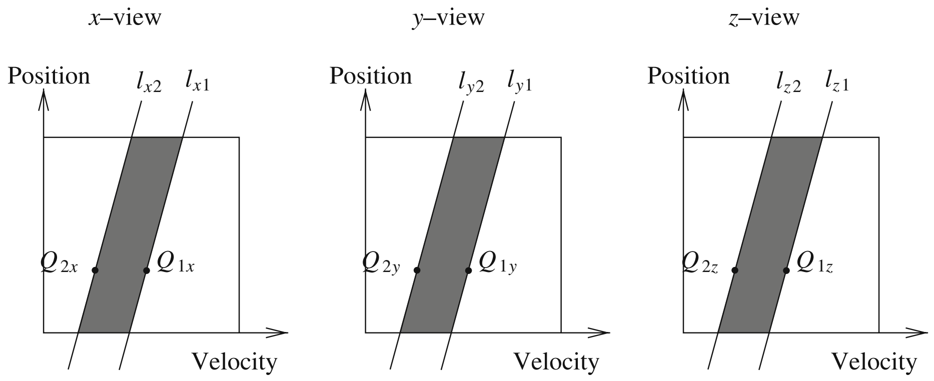


Fig. 2 Views

Definition 26 (Query Space) Given two moving points $Q_1(t)$ and $Q_2(t)$ with respecting dominating lines l_{x1}, l_{y1}, l_{z1}, and l_{x2}, l_{y2}, l_{z2} at time t, the *query space* within the 6-dimensional space of hex representations is a hyper-tunnel formed by the cross product of the areas between l_{x1} and l_{x2} in the x-view, between l_{y1} and l_{y2} in the y-view, and between l_{z1} and l_{z2} in the z-view as shown in Fig. 2.

Two moving objects $Q_1(t)$ and $Q_2(t)$ define a query space at any time t. The query spaces are similar to each other. At any time t the projection of a query space onto each view is an area between two parallel lines with slopes $-t$ and passing through points $Q_1(t)$ and $Q_2(t)$. Hence we can visualize the query space as it changes over time as a moving region sweeping through space as the slopes of the lines change with time.

Intuitively, when at time t a moving point $P(t)$ is in the query space of moving points $Q_1(t)$ and $Q_2(t)$ as in Definition 26, then it dominates $Q_1(t)$ and is dominated by $Q_2(t)$ in the hex representation. Simultaneously, in the three dimensional space in which the objects move, $P(t)$ is in the box whose corner vertices are $Q_1(t)$ and $Q_2(t)$. In the following sections, like many other authors, we use the hex representation as a useful dual or alternative representation that is more amenable to efficient indexing. However, the exact indexing method presented in this paper is novel.

3.2 Approximating the number of points in a bucket

A basic query is to find the number of points that lie in the query space. To answer this basic query we need to look at each of the buckets separately. When the query space includes an entire bucket, then we need to count the every point in the bucket. That can be done using the normalized trend function of the bucket as defined in Definition 17. However, in many cases the query space includes only a part of a bucket. In that case we need to count only the points that are in the intersection of the bucket and the query space. In this section, we show how to do that using a *percentage function* that estimates the percentage of total points in the query space to be within the intersection of the bucket and the query space. This requires to extend Definition 17 as follows.

Definition 27 (Percentage Function) If only one boundary line of the query space goes through a bucket, and r_1 is the region of the bucket in the query space and is above the boundary line, then the *percentage function* is:

$$p = \int_{r_1} F_i \, d\phi \tag{23}$$

If two boundary lines go through the same bucket as in Fig. 2 and regions r_1 and r_2 correspond to regions above Q_1 and Q_2, respectively, then the *percentage function* is:

$$p = \int_{r_1} F_i \, d\phi - \int_{r_2} F_i \, d\phi. \tag{24}$$

The percentage function can be computed for each bucket as shown in the next lemma.

Lemma 28 *For any time t and query space, the percentage function can be computed for each bucket that intersects the query space at time t.*

Proof The proof requires a case analysis. In each of the three views the query space intersects the plane giving the cases shown in Fig. 3.

For each case shown in Fig. 3, we describe the function that results from integration in one view. To extend the result to any number of views, we take the result

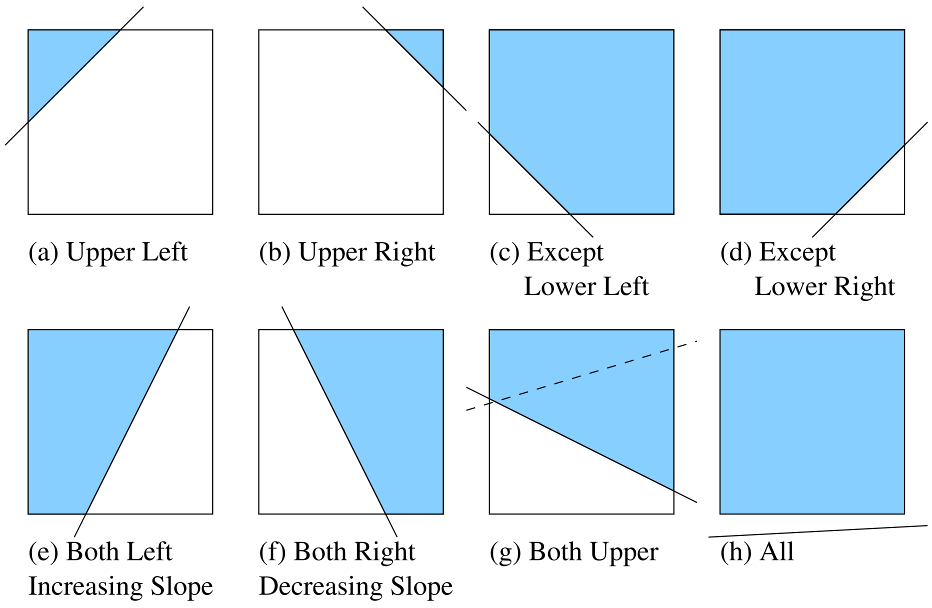


Fig. 3 Eight cases of intersection of buckets and query space

from the last view and integrate it in the next view. If the region below the line were desired, $p_{lower} = \frac{b_i}{n} - p$ gives the percentage of points below the line.

For cases (a)–(h) below, let $Q = (x_{1,q}, x_{2,q}, ..., x_{6,q})$. For the x-view, let the lower left corner vertex be $(x_{1,l}, x_{2,l})$ and the upper right corner vertex be $(x_{1,u}, x_{2,u})$. In addition each line denoted l is given by $x_2 = -t(x_1 - x_{i,q}) + x_{i+1,q}$ and corresponds to a line shown in the corresponding case in Fig. 3.

Case (a) For this case l crosses the bucket at $x_{1,l}$ and $x_{2,u}$. The integral over the shaded region is:

$$p_a = \int_{x_{1,l}}^{\frac{x_{2,u}-x_{2,q}}{-t}+x_{1,q}} \int_{-t(x_1-x_{1,q})+x_{2,q}}^{x_{2,u}} F_i \, dx_2 dx_1 \tag{25}$$

Notice that the lower bound of the integral over dx_2 contains x_1. This dependence within each view does not affect the integration in the remaining four dimensions. The solution to Eq. 25 has the form:

$$at^2 + bt + c + \frac{d}{t} + \frac{e}{t^2}. \tag{26}$$

Case (b) For this case l crosses the bucket at $x_{1,u}$ and $x_{2,u}$. The integral over the shaded region is:

$$p_b = \int_{-\frac{(x_{2,u}-x_{2,q})}{t}+x_{1,q}}^{x_{1,u}} \int_{-t(x_1-x_{1,q})+x_{2,q}}^{x_{2,u}} F_i \, dx_2 dx_1. \tag{27}$$

The solution has the form of Eq. 26.

Case (c) For this case l crosses the bucket at $x_{1,l}$ and $x_{2,l}$. The integral over the shaded region is:

$$p_c = \int_{x_{1,l}}^{\frac{x_{2,l}-x_{2,q}}{-t}+x_{1,q}} \int_{-t(x_1-x_{1,q})+x_{2,q}}^{x_{2,u}} F_i \, dx_2 dx_1 + \int_{\frac{x_{2,l}-x_{2,q}}{-t}+x_{1,q}}^{x_{1,u}} \int_{x_{2,l}}^{x_{2,u}} F_i \, dx_2 dx_1. \tag{28}$$

The solution has the form of Eq. 26.

Case (d) For this case l crosses the bucket at $x_{1,u}$ and $x_{2,l}$. The integral over the shaded region is:

$$p_d = \int_{\frac{x_{2,l}-x_{2,q}}{-t}+x_{1,q}}^{x_{1,u}} \int_{-t(x_1-x_{1,q})+x_{2,q}}^{x_{2,u}} F_i \, dx_2 dx_1 + \int_{x_{1,l}}^{\frac{x_{2,l}-x_{2,q}}{-t}+x_{1,q}} \int_{x_{2,l}}^{x_{2,u}} F_i \, dx_2 dx_1. \tag{29}$$

The solution has the form of Eq. 26.

Case (e) For this case l crosses the bucket at $x_{1,l}$ and $x_{1,u}$. The integral over the shaded region is:

$$p_e = \int_{x_{2,l}}^{x_{2,u}} \int_{x_{1,l}}^{\frac{x_2 - x_{2,q}}{-t} + x_{1,q}} F_i \, dx_1 dx_2. \tag{30}$$

The solution has the form of

$$c + \frac{d}{t} + \frac{e}{t^2} \tag{31}$$

which is like Eq. 26 with $a = b = 0$.

Case (f) Similar to case(e), l crosses the bucket at $x_{1,l}$ and $x_{1,u}$. The integral over the shaded region is:

$$p_f = \int_{x_{2,l}}^{x_{2,u}} \int_{\frac{x_2 - x_{2,q}}{-t} + x_{1,q}}^{x_{1,u}} F_i \, dx_1 dx_2. \tag{32}$$

The solution has the form of Eq. 31.

Case (g) For this case l crosses the bucket at $x_{1,l}$ and $x_{1,u}$. The integral over the shaded region is:

$$p_g = \int_{x_{1,l}}^{x_{1,u}} \int_{-t(x_1 - x_{1,q}) + x_{2,q}}^{x_{2,u}} F_i \, dx_2 dx_1. \tag{33}$$

The solution has the form

$$at^2 + bt + c \tag{34}$$

which is like Eq. 26 with $d = e = 0$.

Case (h) The line l crosses below all the corner vertices hence the integral of the function is:

$$p_h = \int_{x_{1,l}}^{x_{1,u}} \int_{x_{2,l}}^{x_{2,u}} F_i \, dx_2 dx_1. \tag{35}$$

The solution has the form of Eq. 34.

The above cases have solutions for each view in the form of Eq. 26. Hence the percentage function for a single bucket as a function of t is of the form:

$$p = \left(a_x t^2 + b_x t + c_x + \frac{d_x}{t} + \frac{e_x}{t^2}\right) \times \left(a_y t^2 + b_y t + c_y + \frac{d_y}{t} + \frac{e_y}{t^2}\right) \times \left(a_z t^2 + b_z t + c_z + \frac{d_z}{t} + \frac{e_z}{t^2}\right) \tag{36}$$

where $t \neq 0$ when $d_x, d_y, d_z, e_x, e_y, e_z \neq 0$. Finally, renaming variables gives the general form:

$$p = a_6 t^6 + a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + c + \frac{d_1}{t} + \frac{d_2}{t^2} + \frac{d_3}{t^3} + \frac{d_4}{t^4} + \frac{d_5}{t^5} + \frac{d_6}{t^6} \tag{37}$$

where $t \neq 0$ when $d_i \neq 0$ for $1 \leq i \leq 6$. Since Eq. 37 is closed under subtraction, p from Eq. 24 will also have the same form. □

As in Lemma 19, we can still find the number of points in the bucket by multiplying the percentage function by n, the total number of points in the database.

The MaxCount operator is more complex than just the basic operator that finds the number of points in the query space at a specific time. By Definition 1, the MaxCount operator needs to return the maximum number of moving points that are in a query space *at any time instance* $t_{\max}$ *during a time interval*, and it also needs to return the time $t_{\max}$ when the maximum occurs.

Hence for a time interval, we need to consider what happens to the query space as time changes. The query space changes as the dominance lines through the query points change, keeping always a slope of $-t$. As the dominance lines change their clopes, the query space sweeps across space and through the buckets that partition the space. As the query space sweeps through a bucket B_i, it may cross the bucket corner vertices. Each time such a cross happens, the case within the proof of Lemma 28 that applies to B_i may change in one or more of the views. Hence handling large time intervals requires further case analysis. This motivates the following definitions.

Definition 29 (Bucket Time-Interval) A *bucket time-interval* for bucket B_i is a maximal time interval during which no vertex of bucket B_i enters or leaves the query space.

Definition 30 (Index Time-Interval) An *index time-interval* for a set of buckets $\mathcal{B}$ of an index structure is a maximal time interval during which no vertex of *any* bucket in $\mathcal{B}$ enters or leaves the query space.

We follow the convention of denoting bucket and index time-intervals as half-open intervals $[l, u)$ where l is the lower bound and u is the upper bound time instance. This allows a partition of large time intervals.

Definition 31 (Time-Partition Order) Let $\mathcal{B}$ be the set of buckets in an index. Let $(t^{[}, t^{]})$ be a time interval. The *Time-Partition Order* is the set of ordered time instances $t_1, t_2, ..., t_i, ..., t_k$ such that $t_1 = t^{[}$ and $t_k = t^{]}$, and each $[t_i, t_{i+1})$ is an *index time-interval.*

Now we are ready to prove the following.

Lemma 32 *For any time interval element of the Time-Partition Order, the* MaxCount *operator can be approximately computed in* $O(1)$ *time.*

Proof For any element of the time partition order we compute the sum of the percentage functions in each bucket as in Lemma 28. That gives a function of the form of Eq. 37. We find the maximum of that function in the *temporal dimension* by first taking the derivative:

$$p' = \frac{6a_6t^{12}+5a_5t^{11}+4a_4t^{10}+3a_3t^9+2a_2t^8+a_1t^7-d_1t^5-2d_2t^4-3d_3t^3-4d_4t^2-5d_5t-6d_6}{t^7} \tag{38}$$

where $t \neq 0$.

Solving $p' = 0$ requires finding the roots of this 12-degree polynomial, which is not possible using an exact method. Hence we use a common numerical method for an approximate solution. We look at the graph of p'. We check some constant c intervals of Eq. 38 for a change in sign. If there exists a sign change, use the bisection method to find the root. If two points lie within a small ϵ of 0, we perform a check for each of these intervals when no change of sign is found. If some roots exist, then we check them for maximal values along with the end points.

The percentage function p is calculated in $O(1)$ time. Finding $p' = 0$ also takes $O(1)$ time. By placing a constant bound on the number of iterations in the bisection method, we bound by a constant the time required in the numerical section of the algorithm. Plugging in the solution found by the bisection method along with the endpoints also takes $O(1)$ time. Hence, the running time to find the maximum within any time interval element of the time-partition order takes $O(1)$ time. □

Lemma 32 shows that a constant time is enough to find an estimate for MaxCount. However, it is not clear from the lemma how close the estimate is to the actual number of points. It is an interesting open problem to prove a constraint on the approximation ratio (the estimated number divided by the actual number) or some statistical confidence value for the estimation. Instead of a mathematical proof, this paper investigates experimentally the accuracy of the estimation in Section 5.

Although Lemma 32 considers only one time interval element of the Time-Partition Order, it is easy to see that all the other elements can be found easily and handled similarly. At first we find the bucket time-intervals using the following lemma.

Lemma 33 *Given* B *buckets, all the bucket time-intervals can be found in* $O(B)$ *time.*

Proof The bucket time-intervals can be found by finding for each bucket the slopes of the lines through its corner vertices and the query points. Any slope $-t$ means a crossing at time t. Since each bucket has only a constant number of corner vertices, and there are only two query points, finding the crossing times and bucket time-intervals requires only a constant number of calculations for each bucket. Since there are B buckets, this requires only $O(B)$ total time. □

We present the rest of the MaxCount algorithm in Section 3.3.

3.3 An efficient approximate MaxCount algorithm

```
MaxCount($H$, $Q_1$, $Q_2$, $t^{[}$, $t^{]}$)
input:      A set of buckets $H$ built by the index structure presented,
            query points $Q_1(t)$ and $Q_2(t)$ and a query time interval $(t^{[}, t^{]})$.
output:     The estimated MaxCount value.

01.   $TimeIntervals \leftarrow \emptyset$                                           $O(1)$
02.   for $i \leftarrow 0$ to $|H| - 1$                                              $O(B)$
03.       $CrossTimes \leftarrow$ CalculateCrossTimes($Q_1$, $Q_2$, $t^{[}$, $t^{]}$, $H_i$)   $O(1)$
04.       for $j \leftarrow 1$ to $|CrossTimes| - 1$                                 $O(1)$
05.           Union($TimeIntervals$, $TimeInterval(t_{j-1}, t_j)$)              $O(1)$
06.       end for
07.   end for

08.   $TimeIntervals$ = BucketSort($TimeIntervals$)                                  $O(B)$
09.   $IndexTimeIntervals$ = Merge($TimeIntervals$)                                  $O(B)$
10.   for each $IndexTimeInterval \in IndexTimeIntervals$                            $O(B)$
11.       calculate($MaxCount$, $MaxTime$, $IndexTimeInterval$)                      $O(1)$
12.   end for

13.   return ($MaxCount$, $MaxTime$)
```

The algorithm to compute MaxCount with each line labeled with its running time is given above. Line 01 initiates a set of bucket time-interval objects to be empty. Line 03 returns a list of ordered times when a line through Q_1 or Q_2 crosses a bucket corner vertex. As in the proof of Lemma 33 this can be found in $O(1)$ time. Line 05 turns this list into a set of *TimeInterval* objects and adds them to the set of *TimeIntervals*. We list this "for each" loop as $O(1)$ because it consists of a constant number of calculations bounded by the number of vertices in the bucket. Line 08 uses the linear time sorting algorithm BucketSort to sort the bucket time intervals. Line 09 creates the time-partition order and index bucket time intervals from the bucket time intervals in $O(B)$. An additional pass adds the bucket time intervals to the appropriate index time-intervals in $O(B)$. Lines 10-12 perform the MaxCount calculation discussed above.

In order to use the linear time BucketSort algorithm, we need the following definition and lemmas.

Definition 34 (Time-Interval Ordering) The lexicographical ordering $\prec$ of *time intervals* A and B is:

$$A.l < B.l \Rightarrow A \prec B \tag{39}$$

$$A.l = B.l \quad \wedge \quad A.u < B.u \Rightarrow A \prec B \tag{40}$$

$$A.l = B.l \quad \wedge \quad A.u = B.u \Rightarrow A = B \tag{41}$$

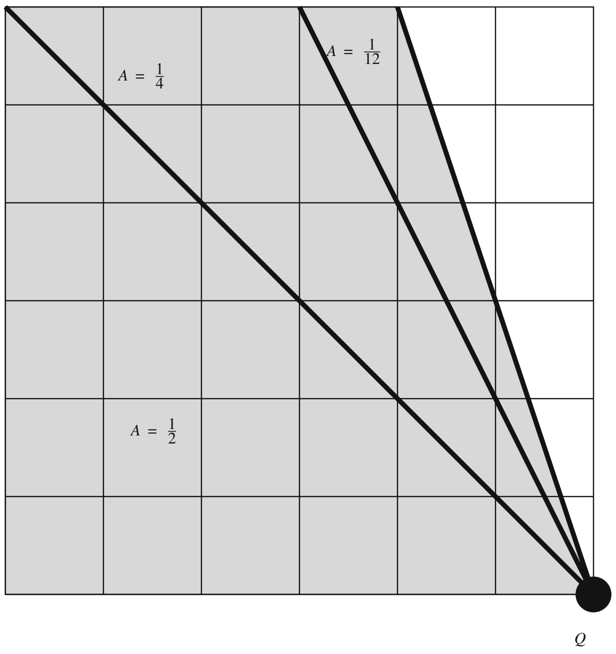
The distribution of time interval objects created in Line 08 of the MaxCount algorithm may not be uniform across the query time interval $T = [t^{[}, t^{]}]$. However, we can still prove the following.

Lemma 35 *If the distribution of buckets is uniform, then the distribution of bucket time-interval objects can be uniformly distributed within the sorting buckets of the bucket sort.*

Proof Consider the relationship between successive slopes measured as the angles between lines through a query point Q with slopes $s_i = -t_i$ and $s_{i+1} = -t_{i+1}$. Suppose $\triangle t = 1$ with $t_0 = 0$ and $t_1 = 1$, then the angle between the two lines is $\triangle s = \frac{\pi}{4}$. The solid lines in Fig. 4 show that half of the bucket corner vertices are swept by the line sweeping through Q between $s_0 = 0$ and $s_1 = -1$. Consider a query time interval [0, 10]. Half of the corner vertices, and thus half of the time intervals, are between time $t = 0$ and $t = 1$. Thus, we conclude that the time interval objects created by sweeping will not be uniformly distributed throughout the query time interval.

Let Q' be the midpoint between Q_1 and Q_2. Let $S = \{t_1, ...t_k\}$ where $t_1 = t^{[}, t_k = t^{]}$ and $t_{i+1} - t_i = L$ for some positive constant L and $1 \leq i \leq k - 1$. Let D_B be a bucket

Fig. 4 Areas of successive slopes



that contains the space in the 6-dimensional index. Model the normalized bucket function for D_B as a constant $F = 1$. Thus p, the bucket probability, from Eq. 36 becomes the hyper-volume of the space swept by the line through Q'. By Lemma 32, we can find the area for a specific time interval in S in constant time. The percentage of sorting buckets, $posb_i$, needed in any time interval $T_i = [t_i, t_{i+1}] \in S$ within the query time interval is given by:

$$posb_i = \frac{p(t_{i+1}) - p(t_i)}{p(t^{\rceil}) - p(t^{\lceil})} \tag{42}$$

Let N be the number of sorting buckets. The number of sorting buckets, $nosb_i$, assigned to interval i is:

$$nosb_i = N \cdot posb_i \tag{43}$$

If $nosb_i < 1$ we can combine it with $nosb_{i+1}$. If the query time interval is very large, then we may need to include multiple time intervals from S to get one sorting bucket. Thus, we create more sorting buckets (with smaller time intervals) in areas where the expected number of bucket time intervals is large. Conversely, we create fewer sorting buckets (with larger time intervals) in areas where the expected number of bucket time intervals is small. Hence we model each sorting bucket so that its time interval length directly relates to the percentage of bucket time intervals that are assigned to it. Thus, we conclude that we will uniformly distribute the time interval objects across all sorting buckets. □

Lemma 36 *Insertion of any bucket time-interval object T_O into the proper sorting bucket can be done in $O(1)$ time.*

Proof The distribution of sorting buckets is determined by k time intervals in Lemma 35. Call these *sorting time interval objects* where each object contains: the lower bound l, the upper bound u, the number of sorting buckets assigned to this interval b_s, the length of the time interval for the sorting bucket w and an array B_p containing pointers to these sorting buckets. Let A be the array of sorting time interval objects, and L be the length of each time interval where the time intervals are as in Lemma 35. Then, finding the correct sorting bucket for T_O requires two calculations:

$$SortingTimeInterval = A\left[\left\lfloor \frac{T_O.l}{L} \right\rfloor\right] \tag{44}$$

$$SortingBucket = B_p\left[\left\lfloor \frac{T_O.l - SortingTimeInterval.l}{w} \right\rfloor\right]. \tag{45}$$

Each of these calculations requires constant time, hence T_O can be inserted into the proper sorting bucket in $O(1)$ time. □

Now we can prove the following.

Theorem 37 *The running time of the* MaxCount *algorithm is $O(B)$ where B is the number of buckets.*

Proof Let H be the set of buckets where each bucket B_i contains the normalized trend function F_i. Let Q_1 and Q_2 be the query points and $[t^{\lceil}, t^{\rceil}]$ be the query time interval. (Lines 01-07): Calculating the time intervals takes $O(B)$ time because the cross times for each bucket can be calculated in constant time. (Line 08): By Lemmas 35 and 36, we have an approximately even distribution of time interval objects within the sorting buckets where we can insert an object in constant time. This result fulfills the requirements of the BucketSort, [9], which allows the intervals to be sorted in $O(B)$ time. (Lines 09-12): Calculate the MaxCount and time for each time interval in constant time using Lemma 32. These lines take $O(B)$ time because there are $O(B)$ time intervals. Finding the global MaxCount and time requires retaining the maximum time and count at line 11. Returning the MaxCount and time takes $O(1)$ time. Thus, the running time is given by $O(B) + O(B) + O(B) + O(1) = O(B)$. □

3.4 An inefficient exact MaxCount algorithm

```
ExactMaxCount($D$, $Q_1$, $Q_2$, $t^{\lceil}$, $t^{\rceil}$)
input:   $D$ is the database of points. The query is made up of a
         hyper-rectangle $Q$ defined by points $Q_1$ and $Q_2$ and the time
         interval $T = [t^{\lceil}, t^{\rceil}]$
output:  The exact MaxCount and time at which it occurs.

01.  $Times \leftarrow \emptyset$ //of $CrossTime$ objects                              $O(1)$
02.  for each point $p_i \in D$                                                $O(N)$
03.      if $p_i \in Q$ during $T$                                              $O(1)$
04.          $EntryTime \leftarrow CalculateEntryTime(p_i, Q, T)$              $O(1)$
05.          $ExitTime \leftarrow CalculateExitTime(p_i, Q, T)$                $O(1)$
06.          if $EntryTime \in Times$                                         $O(1)$
07.              $Times$.get($EntryTime$).$Count$++                            $O(1)$
08.          else
09.              $Times$.add(new$CrossTime$($EntryTime$))                      $O(1)$
10.          end if
11.          if $ExitTime \in Times$                                          $O(1)$
12.              $Times$.get($ExitTime$).$Count$--                             $O(1)$
13.          else
14.              $Times$.add(new$CrossTime$($ExitTime$))                       $O(1)$
15.          end if
16.  end for
17.  Sort($Times$)                                                           $O(n \log n)$
18.  traverse($Times$, $time$, $Max$-$Count$) //tracking time                  $O(N)$
                                                   //and MaxCount
19.  return (time,MaxCount)                                                   $O(1)$
```

While the efficient approximate MaxCount algorithm described in Section 3.3 is our main contribution, it is worth to compare it with a naive MaxCount algorithm (see above) which finds the exact MaxCount value. It is easy to see that the exact algorithm is inefficient. More precisely, its running time is:

$$O(N) + O(n \log n) \tag{46}$$

where N is the number of moving points and n is the number of moving points in the query space.

It is possible to slightly improve the algorithm below. First, divide the index space into k subspaces and maintain separate partial databases for each. Assign processes on individual systems to each database to calculate the MaxCount query and return the time intervals to a central process. Merging the time interval lists into a global time interval list saves time on the sorting part of the algorithm. The running time for each of k partial databases would be close to $O(\frac{n}{k} \log \frac{n}{k})$. This result is an approximate value because we do not guarantee an even split between partial databases. Placing buckets for each partial database in a Tree structure may be reasonable and could cut down the average running time to $O(\log N + n \log n/k)$.

4 The CountRange and threshold operators

Next we describe the CountRange operator in Section 4.1 and the threshold operators ThresholdRange, ThresholdCount, ThresholdSum, and ThresholdAvg in Section 4.2.

4.1 The CountRange operator

The CountRange algorithm is an adaptation of MaxCount in that it is the Count portion of the MaxCount query. Using the equations for the cases described in Fig. 3, we calculate the CountRange as follows.

For each bucket we determine if the bucket is completely in or completely out of the query space. First we find the beginning and ending time intervals. For each time interval, we get the associated function $\triangle p$ given in Eq. 24 and its components. The components $\triangle p$ given in Eq. 23 define the area above a line through Q_1 and Q_2 at times $t^{[}$ and $t^{]}$. Figures 5 and 6 show these four lines. Figure 5 shows the shaded area defined by:

$$\triangle \overleftarrow{p} = p_{Q_2,t^{[}} - p_{Q_1,t^{]}}. \tag{47}$$

Figure 6 shows the shaded area:

$$\triangle \overrightarrow{p} = p_{Q_2,t^{]}} - p_{Q_1,t^{[}}. \tag{48}$$

If $\triangle \overleftarrow{p}$ or $\triangle \overrightarrow{p}$ for bucket i is equal to the count of the bucket, then bucket i is completely contained in the query. If $\triangle \overleftarrow{p}$ and $\triangle \overrightarrow{p}$ for bucket i are equal to 0, then bucket i is not contained in the query. If neither of these is true, we approximate the

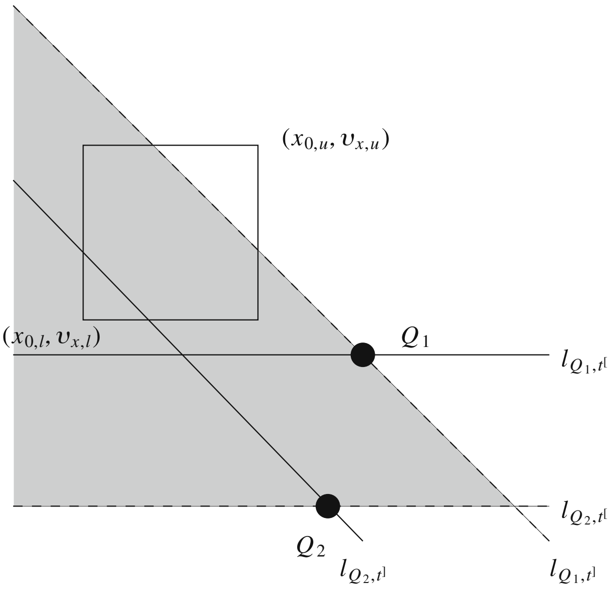


Fig. 5 CountRange Q_1 at $t^\rceil$ to Q_2 at $t^\lceil$

count for bucket i as the $\max(\triangle\overleftarrow{p}, \triangle\overrightarrow{p})$. That is, we calculate the number of points in bucket i that contribute to the CountRange as:

$$count_i = \begin{cases} b_i & \text{if } \triangle\overleftarrow{p} = b_i \vee \triangle\overrightarrow{p} = b_i \\ 0 & \text{if } \triangle\overleftarrow{p} = \triangle\overrightarrow{p} = 0 \\ \max(\triangle\overleftarrow{p}, \triangle\overrightarrow{p}) & \text{otherwise} \end{cases} \tag{49}$$

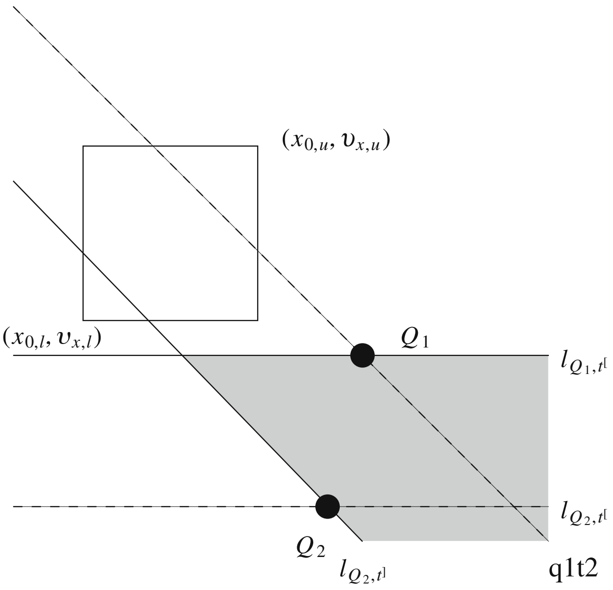


Fig. 6 CountRange Q_1 at $t^\lceil$ to Q_2 at $t^\rceil$

This calculation requires that we keep the single dimension equations for Q_1 and Q_2 available and not discard them after finding $\triangle p$ (see Eq. 24).

Hence, we have the following algorithm for CountRange:

```
CountRange($H$, $Q_1$, $Q_2$, $t^{[}$, $t^{]}$)
input:      A set of buckets $H$ built by the index structure presented,
            query points $Q_1(t)$ and $Q_2(t)$ and a query time interval $(t^{[}, t^{]})$.
output:     the estimated CountRange.

1.    $Count \leftarrow 0$                                                  $O(1)$
2.    for each $bucket\ B_i \in D$                                          $O(B)$
3.        Calculate($\triangle \overleftarrow{p}$, $\triangle \overrightarrow{p}$) //using Eqs. 47–48    $O(1)$
4.        Calculate($count_i$) //using Eq. 49                               $O(1)$
5.        $Count \leftarrow Count + count_i$                                $O(1)$
6.    end for
7.    return $Count$                                                        $O(1)$
```

Theorem 38 *The* CountRange *query runs in $O(B)$ time.*

Proof Consider two different data structures for our buckets: HashTables and R-trees. In the case of indexing using an R-tree, the worst case requires that we examine all buckets used in generating CountRange. It is possible that this list could include all B buckets giving a worst case of $O(B)$. In the case of using a HashTable, we must examine all B buckets. By Lemma 19, and because Eqs. 37 and 49 are calculated in constant time, each bucket can be examined to determine the count that contributes to the CountRange query in constant time. Therefore, the algorithm runs in $O(B)$ time. □

We note that CountRange is a simplification of the MaxCount operator in that we do not examine every time interval. Further we have a slightly different form of $\triangle p$ from Eq. 24 to find the count.

4.2 The threshold operators: range, count, sum and average

We implemented the ThresholdRange algorithm as shown below. The algorithm relates to MaxCount in the way we calculate the aggregation. We maintain a running count to find time intervals that exceed the threshold value M. If we set the threshold value near the MaxCount value ($M \rightarrow$ MaxCount), ThresholdRange finds a small interval containing the MaxCount. We demonstrate this in the experimental results, Section 5.

```
THRESHOLDRANGE($H$, $Q_1$, $Q_2$, $t^{[}$, $t^{]}$, $M$)
input:    A set of buckets $H$ build by the index structure presented,
          query points $Q_1(t)$ and $Q_2(t)$, a query time interval $[t^{[}, t^{]}]$,
          and $M$ is the threshold value
output:   The estimated set of time intervals where $R$ contains more
          than $M$ points.

01 - 08 are the same as the MAXCOUNT algorithm.
09.  $TimeIntervals \leftarrow \emptyset$                                          $O(1)$
10.  for each $TimeInterval \in TimePartitionOrder$                             $O(B)$
11.      $CMaxCount \leftarrow$ CALCULATE(MAXCOUNT, $MaxTime$, $TimeInterval$)  $O(1)$
12.      if $CMaxCount > M$                                                     $O(1)$
13.          $TimeIntervals \leftarrow TimeIntervals \bigcup TimeInterval$      $O(1)$
14.      end if
15.  end for
16.  MERGE($TimeIntervals$)                                                     $O(B)$
17.  return $TimeIntervals$
```

We can analyze the running time of ThresholdRange similarly to the running time of MaxCount and prove the following theorem.

Theorem 39 *The estimated* ThresholdRange *query runs in $O(B)$ time.*

Proof The ThresholdRange algorithm differs from the MaxCount algorithm only in lines 09-17. Lines 11-14 run in $O(1)$ time. Line 10 executes lines 11-13 $O(B)$ times. In line 16, Merge(*TimeIntervals*) is a linear walk of the time intervals that joins adjacent time intervals T_a and T_b when $T_a \bigcup T_b$ would form a continuous time interval. The calculation is trivially $O(1)$ time for joining the adjacent intervals. Hence, we conclude by Theorem 37 that the ThresholdRange runs in $O(B)$ time. □

Based on ThresholdRange, we give the following three operators:

ThresholdCount:
By adding a line between 14 and 15 in the ThresholdRange algorithm that counts the merged time intervals, we can return the count of time intervals during the query time interval where congestion occurs. This count of time intervals gives a measure of variation in congestion. That is, if we have lots of time intervals, we expect that we have a large number of pockets of congestion. Since ThresholdCount does not give information relative to the entire time interval, it may need to be examined in light of the total time above the threshold.

ThresholdSum:
By summing the times instead of using the $\bigcup$ operator in line 13 of the ThresholdRange algorithm, we can return the total congestion time during the query time interval. This total gives a measure of the severity of congestion that may be compared to the length of query time.

ThresholdAverage:
By adding a line between lines 14 and 15 in the ThresholdRange algorithm that finds average length of the merged time intervals, we can return the average length of time each congestion will last. This average gives a different measure of the severity of each congestion.

It is easy to see that and none of the changes to the ThresholdRange algorithm affects the running time. Hence the ThresholdCount, ThresholdSum, and ThresholdAverage algorithms run also in $O(B)$ time.

5 Experimental results

We collected data from over 7,500 queries that were selected from a set of randomly generated queries. The selection process weeded out most similar queries and kept a set that represents narrow queries, wide queries, near corner or edge queries, and queries outside the space contained in the database. Throughout our experiments, we did not see significant accuracy fluctuation due to any of these types of queries.

Each experimental run consists of running all of the queries at several different decreasing bucket sizes on a single data set. We made experimental runs against data sets ranging from 10,000 points to 1,500,000 points.[1]

In the following experimental analysis, we measure the percentage error of the estimation algorithm relative to the exact-count algorithm as follows:

$$Error_{Relative} = \frac{|Exact\ Operator - Estimated\ Operator|}{Exact\ Operator} \tag{50}$$

Equation 50 provides a useful measure if the query returns a reasonable number of points. Queries that return a small number of points indicate that we should use the exact method.

For ThresholdRange, we measure the percentage of intervals given by the accurate algorithm not covered by the estimation algorithm using the operator UC for uncovered. That is, UC(a, b) returns the sum of the lengths of intervals in a not covered by intervals in b. We divide the result by the accurate ThresholdSum to determine the ThresholdRange error:

$$\texttt{error} = \frac{\mathrm{UC}\,(\mathit{Ext.}\ \textsc{ThresholdRange},\ \mathit{Est.}\ \textsc{ThresholdRange})}{\mathit{Ext.}\ \textsc{ThresholdSum}} \tag{51}$$

We also measure the percentage of intervals given by the estimate algorithm not covered by the exact algorithm. We divide the result by the estimated ThresholdSum to determine the ThresholdRange excess-error.

$$\texttt{excess-error} = \frac{\mathrm{UC}\,(\mathit{Est.}\ \textsc{ThresholdRange}\backslash \mathit{Ext.}\ \textsc{ThresholdRange})}{\mathit{Est.}\ \textsc{ThresholdSum}} \tag{52}$$

We performed all the data runs on a Athlon 2000 with 1 GB of RAM. During each of the queries the program does not contact the server tier and, thus, minimizes the impact of running a server on the same computer. The program pre-loads all data into data structures so that even the exact algorithms do not contact the server tier.

[1]Threshold aggregation runs go only to 1 million points at which we already achieve acceptable error.

5.1 Data generation

Data for the experiments was randomly generated around several cluster centers. The i^{th} point generated for the database is located near a randomly selected cluster at a distance between 0 and d, where d is proportional to i. This method is similar to the Ziggurat [16] method of generating Gaussian (or normal) distributions used in the GSTD [33] and G-TERD [35] spatiotemporal data generators [18]. However, our method does not generate strictly Gaussian distributions since the distributions may stretch and compress along an axis. Our goal was to generate a cluster that represents a source location and velocity that has most elements starting near a center point and decreasing as one moves to a boundary for the cluster. This method models source regions where the objects all head about the same direction. A secondary goal was to make certain that clusters were random in size and shape. The program is also capable of approximating a Zipf distribution used in [6], [25], [31]. However, a single Zipf distribution does not test the adaptability of our algorithm well. I.e. our algorithm is capable of modeling a Zipf distribution and as such we could use a single bucket. Figure 7 shows a sample of a data set with points projected onto the three views. The clusters look even more random, because they can overlay one another. When one looks at these, they nearly resemble the lights of a city from the air.

Along with a single Zipf distribution, we also note that a randomly generated uniform-distribution is not a good distribution to use for these types of experiments. Uniform distributions do not test the ability of the algorithm to adapt. In fact from earlier experiments in [1] we have found that using such a distribution gives great (though meaningless) results. The problem resolves to a system capable (and willing to) model a uniform distribution finding a nearly perfect uniform distribution to model. Hence these results are neither realistic, nor meaningful.

5.2 Parameter effects

The index space ranges from 0 to 100 in each dimension. The *number of points* in the different data sets ranges from 10,000 to 1,500,000. The following parameters were used in creating the index and finding the MaxCount.

Size of Buckets: The size of the buckets determines the number of possible buckets in the index. In the experiments, buckets divide the space up such that there are 5 to

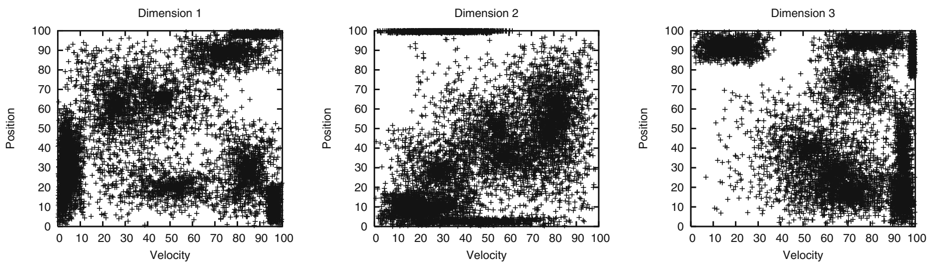


Fig. 7 Sample data view

20 divisions in each dimension.[2] These divisions equate to bucket sizes ranging from 5 to 20 units wide in each dimension. Relative to our previous work [1], this algorithm puts much more space into each bucket creating bigger buckets.

Query Location: Locating the query near the lower or upper corners affects relative accuracy because the query returns very few points. Queries in this region are not interesting because they rarely involve many points and represent a query region that moves away from points in the database or barely moves at all. The small number of points returned indicates use of the exact algorithms.

Query Types: In [1], we considered queries with several different characteristics: dense, sparse, and Euclidean distance as it related to bucket size. By modeling the skew in buckets, we minimize the effect of these characteristics to the point that they did not impact the query error. Queries where the distance between the query points was small appeared to do as well as wider queries *providing they returned a reasonable number of points*. This result is a clear improvement over previous work that assumed uniform density within a bucket.

Cluster Points: Index space saturation determines the number of buckets necessary for the index. The number of cluster points does not appear to affect error as much as the space saturation. Further, we do not consider a larger number of cluster points reasonable since the index space approaches a uniform distribution as the number of cluster points increases. Gaps introduce difficult areas to model when they are not uniform. And once again we reiterate, uniform distributions are not useful. In our experiments cluster points number between 10 and 50.

Histogram Divisions: Increasing histogram divisions to $s > 5$ had no affect on the accuracy. This result is not unexpected because histograms are used to define a trend function relative to trend functions on other axes. Increasing the histogram divisions has a tendency to flatten the lines. However, normalization flattens the trend function while maintaining the relationships between trends and hence this behavior is easily explained. Thus, increasing histogram divisions only increases the running time without increasing accuracy.

Threshold Value: The threshold value determines the accuracy when set to low values compared to the number of points in the database. As expected, these extreme point values produce accurate estimations. High values also follow this trend.

Time Endpoints: When dealing with either small time end points or small buckets, the method is susceptible to rounding error. In particular, Eq. 37 contains both t^6 and $\frac{1}{t^6}$ terms. For very small values, on the order of 1×10^{-54} for 64-bit doubles, these calculations are extremely sensitive and care must be given to guard against rounding error. Those errors showed in two ways. First, by a direct warning programmed into the solution, and second, by a series of fairly stable time values for the MaxCount

[2]Some MaxCount runs included up to 40 divisions increasing accuracy, but not enough to warrant the extra running time.

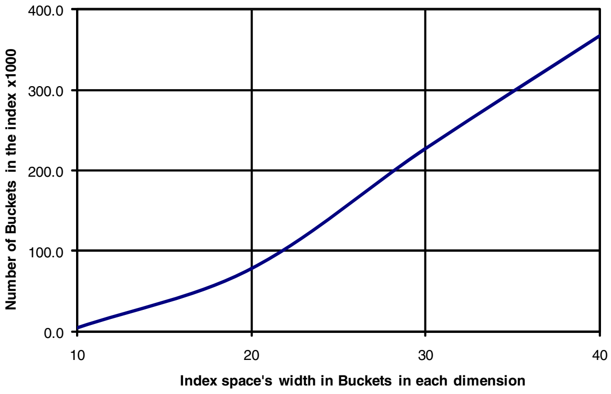


Fig. 8 Ratio of the number of buckets in the index to the width of the space measured in buckets

followed by unstable variations when increasing the number of buckets. At some point, smaller bucket sizes increase the likelihood of errors in both time and count values. Also smaller buckets contain fewer points, which impacts the size of the constants in Eq. 37. Hence, as the bucket size becomes smaller in successive runs, the existence of instability in the time values after a series of stable values predicts that an accurate MaxCount may be found in the previous larger bucket size. *Throughout our experiments, this condition was an excellent predictor of an accurate* MaxCount.

The experiments demonstrated that 6-dimensional space compounds the problem when creating small buckets. Creating an index with unit buckets would result in the possibility of having 1×10^{12} buckets. Clearly this number is unrealistic for common moving object applications where we may be dealing with million(s) of objects. In practice the number of buckets needed to reach acceptable error levels was between 78,000 and 227,000 buckets. These numbers reflect the ability to reach error levels under 5% and were roughly related to the saturation of the space by the points. It should be clear that a higher saturation of the space by points would require a larger number of buckets. Figure 8 shows that we had a roughly linear increase in the number of buckets for an exponential increase in the space. This pleasant surprise indicates that for unsaturated data sets, the exponential explosion of space is manageable.

5.3 Running time observations

Figure 9 shows the average ratio of the exact MaxCount running time to the estimated MaxCount running time as a function of the number of points in the database. This result shows a nearly exponential growth when comparing the values between 10,000 and 1,000,000. The leveling off occurs because the number of points returned by the queries of 1 million points nearly equals the number of points returned by the queries of 1.5 million points. This result precisely matches our running-time analysis of the exact and estimation algorithms.

A natural question is when to use the exact versus the estimated methods. In runs with a small number of points that need to be processed, the exact and estimation methods run about equally fast. However, when the result size reaches values greater than 40,000 (our experiments returned sets as large as 331,491), the estimation algorithms run up to 35 times faster than the exact algorithms. Further, we note that

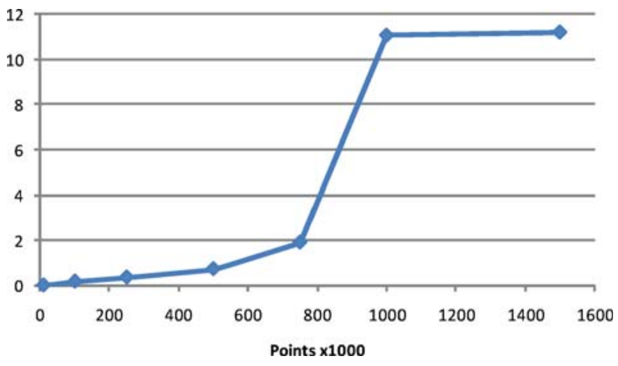


Fig. 9 Ratio of exact running time to estimated running time

the error is less predictable at smaller results sizes. Hence for small databases or in queries that return small result sets, efficiency and accuracy both indicate using the exact method. However, for large data sets greater than or equal to 1 million points, the estimation method greatly out-performs the exact method.

5.4 Operator observations

As expected, we noticed that each operator runs in about the same time as MaxCount. Only error values seemed to be different when studying different types of aggregation (e.g., when studying overlap error in ThresholdRange versus count error in MaxCount). Never-the-less, we have similarities between the results. Almost all the figures in this section look like a view of mountains from a valley. That is what we expected to see and the lower and flatter the terrain the better. Buckets increase from back to front and point set sizes increase from left to right.

5.5 MaxCount

Figure 10 shows that increasing the number of buckets to the indicated values dramatically decreases the MaxCount error. As the number of points increases we also see a decrease in the error. Note that for larger buckets (e.g. smaller values on the “Buckets per Dimension axis”), the error decreases at a slightly faster rate.

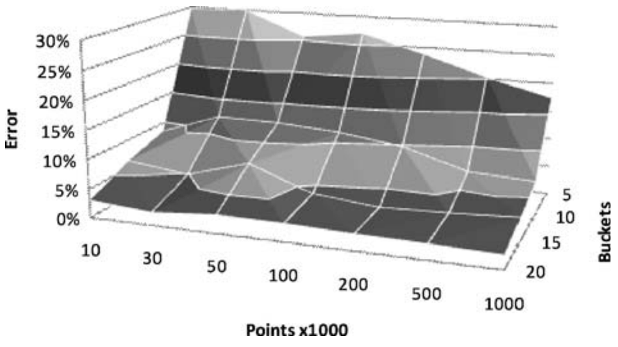
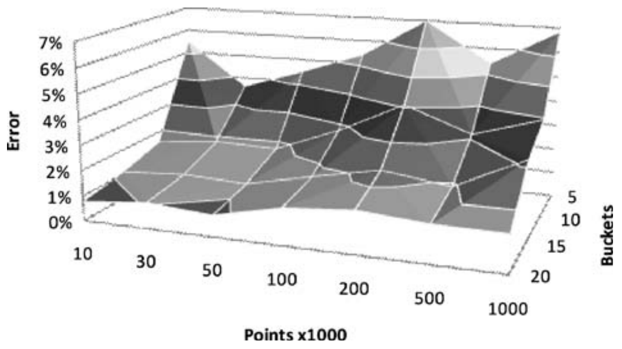


Fig. 10 MaxCount error

Fig. 11 CountRange error



The exact MaxCount provided the values against which our estimation algorithm was tested for accuracy. Since the method does not rely on buckets, and has zero error, we note only that on queries with small result sizes, this method performs as well, or better than the estimation algorithm.

5.6 CountRange

Other CountRange algorithms have achieved error values between 2% and 3%. Using our method we conjecture that we could reduce the error because our method of approximation, although much more complicated, theoretically adapts to skewed distributions better than other methods. Figure 11 shows that we achieved errors under 2% for 20 buckets across all the data sets, and in some cases, under 1%.

Count range also performs about the same speed as the threshold operators due to its similar implementation.

5.7 ThresholdRange

Figures 12 and 13 give the ThresholdRange error and ThresholdRange excess error respectively for $T = 10$. ThresholdRange error gives the percentage of the exact intervals not covered by the estimation value, and ThresholdRange excess error gives the percentage of the estimation not covering the exact. These figures show that our method acts conservatively in covering more than is needed. However,

Fig. 12 ThresholdRange error

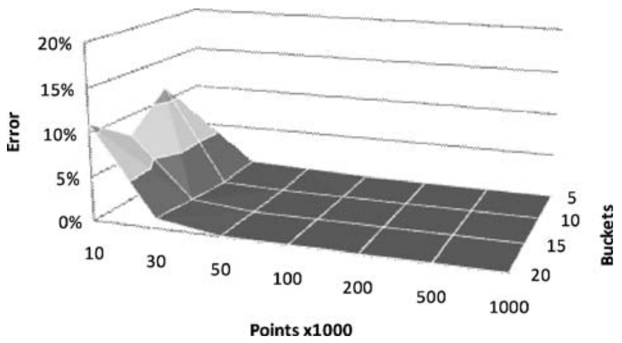
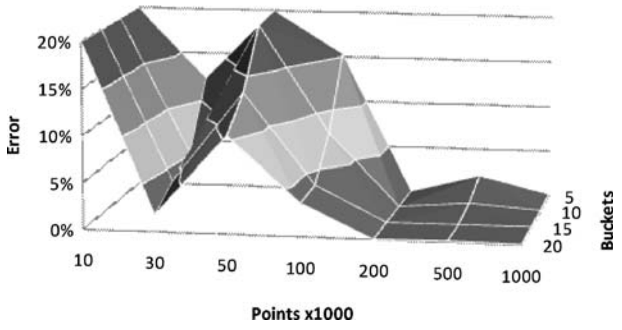


Fig. 13 ThresholdRange error



at larger point-set sizes, we still achieve under 5% error. Figure 12 shows 0% error caused by the point count staying above 10% in data sets containing more than 30,000 points. Figure 13 shows that we covered at least 10% more time in the query time interval than needed until we reach larger point sets. Still, we showed improvement with more buckets.

At $T = 1{,}000$, we see 0% error until we reach point sets of 500,000 and greater. Figure 14 shows excellent results with buckets above 10. Also, Fig. 15 shows that the excess error drops to near 0% as well.

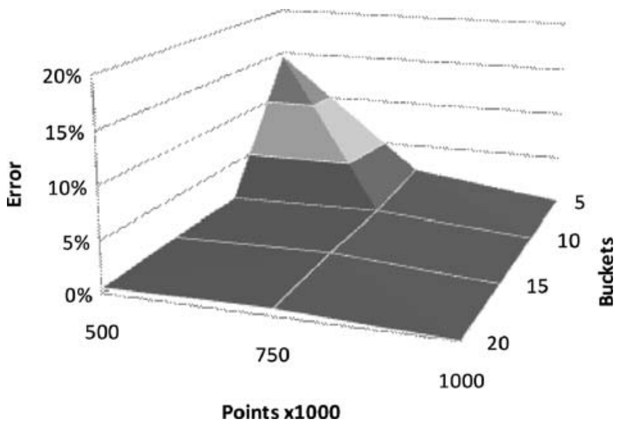
Figures 16 and 17 show what happens when we find an interval near the MaxCount value. The two figures show the consequences of the estimation intervals being offset from the exact intervals by small amounts. The error decreases with more buckets.

5.8 ThresholdCount

This operator is the only operator that does not have relative error measurements. Instead we report the average number of intervals the estimation method differs from the exact method. As you can see, we differ by two from the correct number.

Figure 18 shows the average error at $T = 10$ where the errors are small. Figure 19 ($T = 1{,}000$) looks much worse, but in reality we are still below two intervals off.

Fig. 14 ThresholdRange error, $T = 1{,}000$



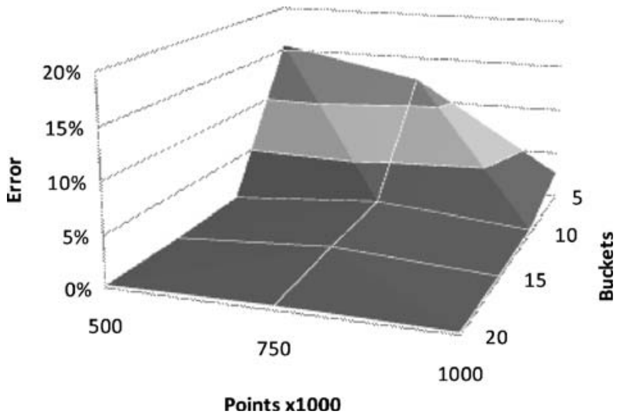


Fig. 15 ThresholdRange excess error, $T = 1{,}000$

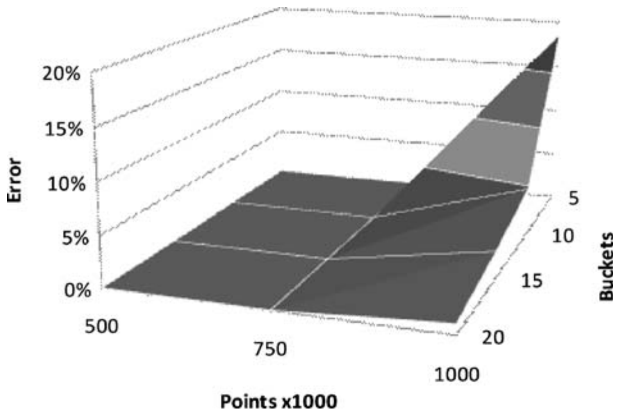


Fig. 16 ThresholdRange error, $T = 100{,}000$

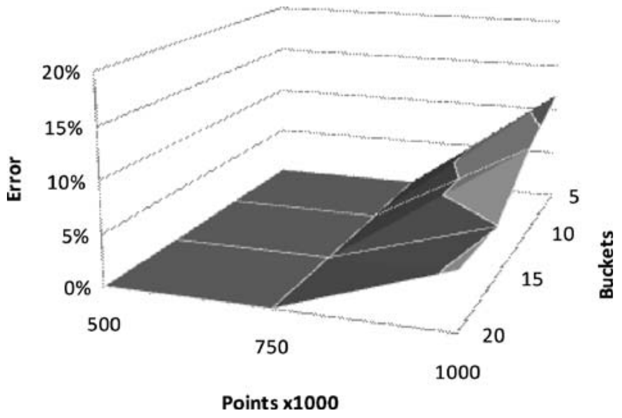


Fig. 17 ThresholdRange excess error, $T = 100{,}000$

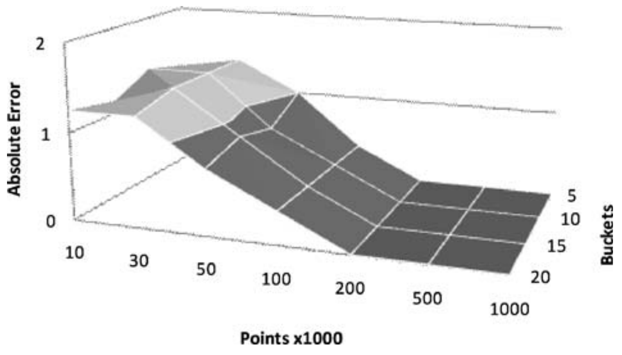


Fig. 18 ThresholdCount error, $T = 10$

We also note that the estimation may split or combine an interval incorrectly when the intervals are very close together without greatly affecting the error of other operators. Given this possibility, the results are excellent.

5.9 ThresholdSum

ThresholdSum gives the total time above the threshold T. As one can see in Fig. 20, at higher bucket counts we have excellent error rates at $T = 10$. We didn't always expect great results at this threshold level across all data sets, but ThresholdSum gives this result consistently all the way across.

We do note that when the threshold approaches MaxCount, we see extremely good accuracy as shown in Fig. 21.

5.10 ThresholdAverage

ThresholdAverage gives the average length of each time interval. Figure 22 shows the now familiar mountains descending below 5% error at 20 buckets for $T = 10$. The Figure also shows that even though a few of the data sets tended to have good results at five and ten buckets, these results are not guaranteed in general. In Fig. 23, the error reaches a plateau below 5% with only small bumps in the data.

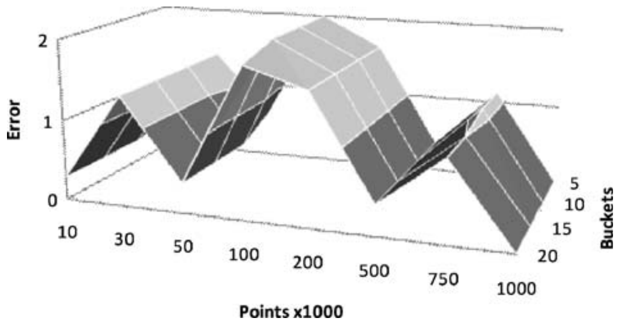


Fig. 19 ThresholdCount error, $T = 100$

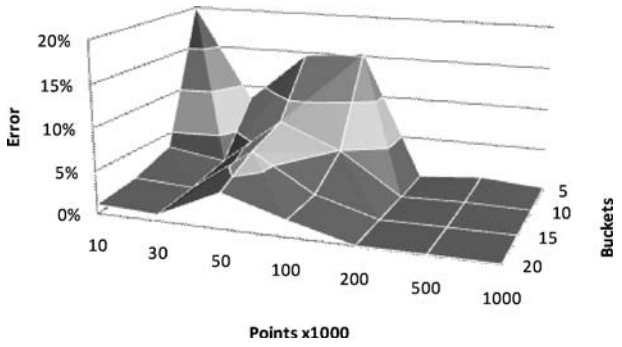


Fig. 20 ThresholdSum error, $T = 10$

6 Related work

Our research fits into the general area of moving object and spatio-temporal operators, which attracted a huge interest in recent years, shown by the recent textbooks [12], [23], [26], and [29]. However, our research is still unique in several respects. Section 6.1 discusses the background of the MaxCount operator, which was developed by our research group. Section 6.2 discusses previous work that is related to CountRange and the threshold operators.

6.1 The uniqueness of the MaxCount operator

To our knowledge, the MaxCount operator is a unique idea of the second author of the present paper. The MaxCount operator was first described in [25] by the second author and Yi Chen, a graduate student at the University of Nebraska-Lincoln. Revesz and Chen [25] considered only 1-dimensional moving point objects, which were represented in a 2-dimensional static point in a dual space. Their algorithm generated for any N points in this dual space, an $O(N^2)$ size data structure that divided the dual space into N regions, such that in each region all the possible

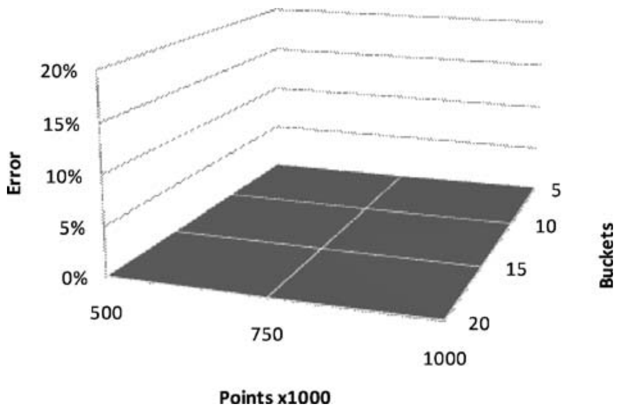


Fig. 21 ThresholdSum error, $T = 100{,}000$

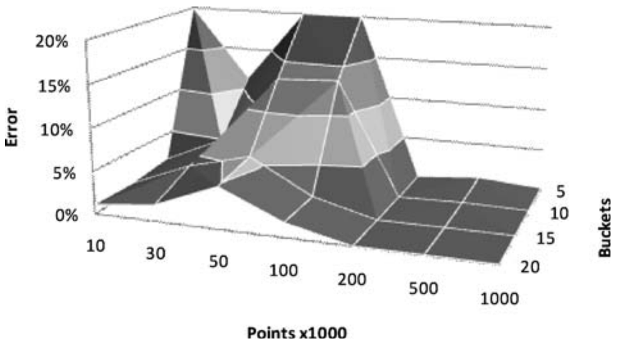


Fig. 22 ThresholdAverage error, $T = 10$

query points had the same MaxCount. Therefore the problem of MaxCount was reduced to the problem of point location in a subdivision of the plane. It was not a practical solution because of the quadratic size of the data structure. Moreover, the data structure was static, that is, it had to be regenerated every time a point was inserted into the database. Chen and Revesz [5] presented a second approach to the MaxCount problem. This approach also was limited to 1-dimensional moving objects, but it made a brake from an exact solution and was the first MaxCount algorithm to consider estimation. Chen and Revesz [5] assumed that the points within each bucket had a uniform distribution.

Later the first author, while a graduate student at the University of Nebraska—Lincoln, and the second author developed a new approach to the MaxCount operator. The distribution of the buckets was extended from uniform to the one defined by trend functions as in Definition 14. A preliminary result for 2-dimensional moving objects was presented in [1, 2]. That preliminary result did not consider higher dimensions, and it also had only a static index structure.

The present work, which is a major extension of the techniques in [1] to make it a dynamic algorithm and applicable to d-dimensional moving points, is the final and

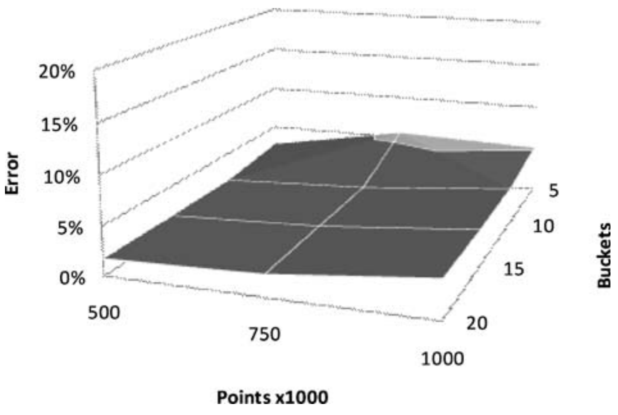


Fig. 23 ThresholdAverage error, $T = 1{,}000$

Table 2 MaxCount operator summary

Max. dim.	Worst case time	Space	Exact or est.	Static or dynamic	Reference
1	$O(log\ N)$	$O(N^2)$	Exact	Static	Revesz and Chen [25]
1	$O(B \log B)$	$O(B)$	Est.	Static	Chen and Revesz [5]
2	$O(B \log B)$	$O(B)$	Est.	Static	Anderson [1]
d	$O(B)$	$O(B)$	Est.	Dynamic	present paper

N is the total number of moving points and B is the number of buckets in the index.

only journal article on MaxCount. The prior works on MaxCount were presented only in conferences. Table 2 summarizes the history of the MaxCount operator.

The MaxCount operator is a new moving object operator that cannot be reduced to other known operators. In particular, the Count and the Max operators have only a titular relationship to the MaxCount operator because one cannot use the Count and Max operators to implement the MaxCount operator with a continuously moving query space. Hence Table 2 does not discuss Count and Max as related work.

6.2 The CountRange and the threshold operators

For the CountRange and the threshold operators we cannot make as strong claim to uniqueness as in the case of the MaxCount operator. CountRange and various Range operators were defined by other authors and studied in many prior papers, which are summarized in Table 3. These operators are discussed in one table

Table 3 CountRange and Range operators summary

Max. dim.	Worst case time	Worst case space	Exact or est.	Reference
2	$O(N^{\frac{3}{4}+\epsilon}+k)$	$O(N)$	Exact	Kollios et al. [15]
2	$O(\log_2 N+k)$	$O(N^2)^3$	Exact	
2	$O(N)$	$O(N)$	Exact	Papadopoulos et al. [19]
3	$O(N)$	$O(N)$	Exact	Saltenis et al. [27]
3	$O(N)$	$O(N)$	Exact	Cai and Revesz [4]
d	$O(N)$	$O(N)$	Exact	Porkaew et al. [22]
d	$O(B^{d-1}\log_B^d N)$	$O(\frac{N}{B}\log_B^{d-1} N)$	Exact	Zhang et al. [36]
1	$O(\log^2 N)$	$O(N \log N)$	Est.	Revesz [24]
2	$O(\log_B N+\frac{C}{B})$	$O(N)$	Est.	Kollios et al. [15]
2	$O(B)$	$O(B)$	Est.	Choi and Chung [6]
d	$O(B)$	$O(B)$	Est.	Tao, Sun and Papadias [31]
d	$O(\sqrt{N})$	$O(N)$	Est.	Tao and Papadias [30]
d	$O(B)$	$O(B)$	Est.	present paper

N is the total number of moving points and B is the number of buckets in the index. All algorithms are dynamic, that is, allow insertions and deletions of moving objects into the index.

because several Spatiotemporal-Range algorithms can be modified to return the CountRange value by simply counting the number of objects returned.

The related works summarized in Table 3 use a variety of different techniques. Kollios et al. [15] uses a simplex range method. Papadopoulos et al. [19] uses an R^*-tree model. Saltenis et al. [27] and [4] define time-parametric R-trees. Cai and Revesz [4] usually gives a tighter parametric bounding box than [27] gives for the same set of moving objects in each node of the time-parametric R-tree. Porkaew et al. [22] is another modification of the R-tree model, while Zhang et al. [36] uses an ECDF-B-tree. Revesz [24] also uses a modified ECDF-B-tree and estimates the points below a line in the dual space by a formula that reduces to a set of ECDF-B-tree queries. Choi and Chung [6] seems to be the first spatio-temporal selectivity estimation method with buckets. Tao and Papadias [30] give an MVRB-tree-based estimation algorithm.

As can be seen, all the above methods differ in technique from our particular approach to the CountRange and various threshold operators. Our approach is related to our MaxCount algorithm, which we defined and solved first, only later exploring the relationship between MaxCount and the other operators. Hence our CountRange and threshold algorithms are novel approaches to known problems. The new approaches yield algorithms that compare well to previous algorithms in the worst case running time and space requirements, as shown in Table 3.

There are some additional related works which are not shown in Table 3 because they use different assumptions than the above related works. These can be grouped into three groups are follows.

The first group of related works restrict the movement of moving objects in some way. Civilis et al. [7], [8] gave indexing methods that use networks, such as roads. Pfoser and Jensen [21] also considered spatiotemporal range queries in networks. Gupta et al. [11] gave a technique for answering spatiotemporal range queries on objects that move along curves in a planar graph. de Almeida and Güting [10] proposed the MON-tree to index moving objects in networks to find the spatiotemporal range and windows queries.

The second group of related works consider only predicting the positions of particular (non-linearly) moving objects at some future time. Mokhtar et al. [17] queries the past, present, and future positions of moving objects in constraint databases. Tayeb et al. [32] adapted the PMR-quadtree [28] for indexing moving objects to answer time-slice queries, which they called instantaneous queries, and infinitely repeated time-slice queries, called continuous queries. Search performance is similar to quadtrees and allows searches in $O(\log N)$ time. Recently, Pelanis et al. [20] proposed the R^{PPF}-tree that indexes past, present and predictive positions of moving points, and extends the previous work on TPR-trees [27] with a partial persistence framework. Related to these papers, Trajcevski et al. [34] present a method for generating pseudo-trajectories of moving objects about which only limited information is known.

Finally, the third group of related works do not allow the query to move or change shape over time. For example, Hadjieleftheriou et al. [14] works with this assumption to find an efficient estimation method for the areas where the density of objects is above a specific threshold during a specific time interval.

7 Future work and conclusion

A practical future work may include implementing our algorithms in a grid computing environment to further decrease the running time. An interesting and deep theoretical open problem is to improve Lemma 32 by proving a constraint on the approximation ratio or some confidence value for the estimation.

We started this research with the MaxCount problem naturally surfacing while we looked at GIS applications regarding congestion, like the example problems of the introduction. Those and related problems will surely play an in increasingly greater role in GIS as mobile phones and wireless computers become even more ubiquous, and there will be a major need to monitor and service them efficiently.

We hope our work inspires further interdisciplinary research and an appreciation of various disciplines, new or old, in fashion or out of fashion, and with large government funding or without. In particular, our work relies more than usual in the indexing area on differential calculus. That is a cautionary tale in an era when computer science departments are dropping introductory calculus courses. Successful interdisciplinary research may involve more than expected.

References

1. Anderson S (2006) Aggregation estimation for 2D moving points. In: Proceedings of the 13th international symposium on temporal representation and reasoning. IEEE Press, pp 137–144
2. Anderson S (2007) Software verification and spatiotemporal aggregation in constraint databases. Ph.D. thesis, University of Nebraska-Lincoln, USA
3. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of ACM SIGMOD international conference on management of data. ACM Press, pp 322–331
4. Cai M, Revesz P (2000) Parametric R-tree: an index structure for moving objects. In: Proceedings of the 10th COMAD international conference on management of data. McGraw-Hill, pp 57–64
5. Chen Y, Revesz P (2004) Max-Count aggregation estimation for moving points. In: Proceedings of the 11th international symposium on temporal representation and reasoning. IEEE Press, pp 103–108
6. Choi Y-J, Chung C-W (2002) Selectivity estimation for spatio-temporal queries to moving objects. In: Proceedings of the ACM SIGMOD international conference on management of data. ACM Press, pp 440–451
7. Civilis A, Jensen CS, Nenortaite J, Pakalnis S (2004) Efficient tracking of moving objects with precision guarantees. First annual international conference on mobile and ubiquitous systems: networking and services. pp 164–173
8. Civilis A, Jensen CS, Pakalnis S (2005) Techniques for efficient road-network-based tracking of moving objects. IEEE Trans Knowl Data Eng 17(5):698–712
9. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. MIT Press, Massachusetts, USA
10. de Almeida VT, Güting RH (2005) Indexing the trajectories of moving objects in networks*. Geoinformatica 9:33–60, MON-tree

11. Gupta S, Kopparty S, Ravishankar CV (2004) Roads, codes and spatiotemporal queries. In: Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. pp 115–124
12. Guting RH, Schneider M (2005) Moving objects databases. Morgan Kaufmann, San Francisco, CA, USA
13. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Yormark B (ed) Proceedings of the ACM SIGMOD international conference on management of data. ACM Press, pp 47–57
14. Hadjieleftheriou M, Kollios G, Gunopulos D, Tsotras VJ (2003) On-line discovery of dense areas in spatio-temporal databases. In: Advances in spatial and temporal databases, 8th international symposium. Springer, pp 306–324
15. Kollios G, Gunopulos D, Tsotras VJ (1999) On indexing mobile objects. In: Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. pp 261–272
16. Marsaglia G, Tsang W (2000) The ziggurat method for generating random variables. J Stat Software 5(8):1–7
17. Mokhtar H, Su J, Ibarra O (2002) On moving object queries: (extended abstract). In: Proceedings of the 21st symposium on principles of database systems. pp 188–198
18. Nascimento M, Pfoser D, Theodoridis Y (2003) Synthetic and real spatiotemporal datasets. IEEE Data Eng Bull 26(2):26–32
19. Papadopoulos D, Kollios G, Gunopulos D, Tsotras VJ (2002) Indexing mobile objects on the plane. In: Proceedings of the international conference on database and expert systems applications
20. Pelanis M, Saltenis S, Jensen CS (2006) Indexing the past, present, and anticipated future positions of moving objects. ACM Trans Database Syst 31(1):255–298
21. Pfoser D, Jensen CS (2005) Trajectory indexing using movement constraints*. GeoInformatica 9:93–115
22. Porkaew K, Lazaridis I, Mehrotra S (2001) Querying mobile objects in spatio-temporal databases. In: Proceedings of symposium on spatial and temporal databases (SSTD). pp 59–78
23. Revesz P (2002) Introduction to constraint databases, Springer, New York, USA.
24. Revesz P (2005) Efficient rectangle indexing algorithms based on point dominance. In: Proc. of the 12th international symposium on temporal representation and reasoning. IEEE Press, pp 210–212
25. Revesz P, Chen Y (2003) Efficient aggregation over moving objects. In: Proceedings of the 10th international symposium on temporal representation and reasoning. IEEE Press, pp 118–127
26. Rigaux P, Scholl M, Voisard A (2001) Spatial databases: with applications to GIS. Morgan Kaufmann, San Francisco, CA, USA
27. Saltenis S, Jensen CS, Leutenegger ST, Lopez MA (2000) Indexing the positions of continuously moving objects. In: Proc. of the ACM SIGMOD international conference on management of data. ACM Press, pp 331-342
28. Samet H (1990) The design and analysis of spatial data structures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
29. Samet H (2005) Foundations of multidimensional and metric data structures. Morgan Kaufmann, San Francisco, CA, USA
30. Tao Y, Papadias D (2005) Historical spatio-temporal aggregation. ACM Trans Inf Sys 23(1): 61–102
31. Tao Y, Sun J, Papadias D (2003) Selectivity estimation for predictive spatio-temporal queries. In: Proceedings of the 19th international conference on data engineering, pp 417–428
32. Tayeb J, Ulusoy Ö, Wolfson O (1998) A quadtree-based dynamic attribute indexing method. Comput J 41(3):185–200
33. Theodoridis Y, Silva J, Nascimento M (1999) On the generation of spatiotemporal datasets. In: Proc. symposium on spatial databases. Springer, pp 147–164
34. Trajcevski G, Wolfson O, Hinrichs K, Chamberlain S (2004) Managing uncertainty in moving objects databases. ACM Trans Database Syst 29(3):463–507
35. Tzouramanis, T, Vassilakopoulos, M, Manolopoulos, Y. (2002) On the generation of time-evolving regional data*. Geoinformatica 6(3):207–231
36. Zhang D, Gunopulos D, Tsotras VJ, Seeger B (2003) Temporal and spatio-temporal aggregations over data streams using multiple time granularities. Inf Syst 28(1–2):61–84

Scot Anderson obtained his Ph.D. degree in Computer Science from the University of Nebraska—Lincoln in 2007. He is currently an assistant professor at Southern Adventist University. His research interests are geographic information systems, moving objects, and spatio-temporal data.

Peter Revesz holds a Ph.D. degree in Computer Science from Brown University and was a postdoctoral fellow at the University of Toronto before joining the University of Nebraska—Lincoln, where he is currently a full professor in the Department of Computer Science and Engineering. He is well-known as a co-inventor of constraint databases in a highly-cited joint paper with Paris Kanellakis and Gabriel Kuper. He is the author of the book "Introduction to Constraint Databases", which was published by Springer in 2002. His current research interests include geographic information systems and spatio-temporal databases. He has been a visiting professor at the University of Athens in Greece, the University of Hasselt in Belgium and the Max Planck Institute for Computer Science and the University of Freiburg in Germany. He was awarded a Fulbright Award and an Alexander von Humboldt Research Fellowship.