Exploiting velocity distribution skew to speed up moving object indexing

Thi Nguyen[a,b,1,2], Zhen He[c], Rui Zhang[d], Philip G. D. Ward[e]

[a]*Saw Swee Hock School of Public Health, National University of Singapore, 117549, Singapore*
[b]*National University Health System, Singapore*
[c]*Department of Computer Science and Information Technology, La Trobe University, VIC 3086, Australia*
[d]*Department of Computing and Information Systems, The University of Melbourne, VIC 3052, Australia*
[e]*Clayton School of Information Technology, Faculty of Information Technology, Monash University, VIC 3800, Australia*

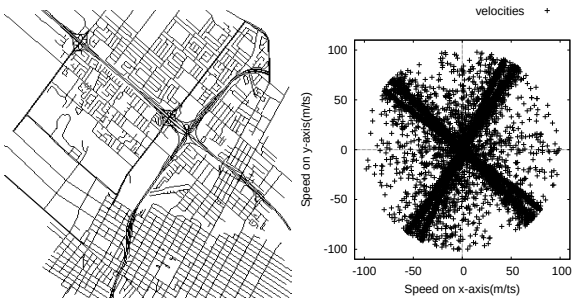
Abstract

There has been intense research interest in moving object indexing in the past decade. However, existing work did not exploit the important property of skewed velocity distributions. In many real world scenarios, objects travel predominantly in only a few directions. Examples include vehicles on road networks, flights, people walking on the street, etc. The search space for a query is heavily dependent on the velocity distribution of the objects grouped in the nodes of an index tree. Motivated by this observation, we propose the *velocity partitioning (VP)* technique, which exploits the skew in velocity distribution to speed up query processing using moving object indexes. The VP technique first identifies the "dominant velocity axes (DVAs)" using a combination of principal components analysis (PCA) and k-means clustering. Then, a moving object index (e.g., a TPR-tree) is created based on each DVA, using the DVA as an axis of the underlying coordinate system. The object is maintained in the index whose DVA is closest to the object's current moving direction. Thus, all the objects in an index are moving in a near 1-dimensional space instead of a 2-dimensional space. As a result, the expansion of the search space with time is greatly reduced, from a quadratic function of the maximum speed (of the objects in the search range) to a near linear function of the maximum speed. The VP technique can be applied to a wide range of moving object index structures. Moreover, we make use of new hardware, solid-state drives (SSDs) to further improve the performance of the VP technique. To this end, we designed a SSD friendly version of the outlier index, called the *RAM-resident compressed grid (RCG)*. We implemented the VP technique on two representative moving object indexes, the TPR*-tree and the B^x-tree. Extensive experiments validate that the VP technique consistently improves the performance of these index structures.

Keywords: Spatial temporal databases, moving objects, indexing, velocity partitioning

1. Introduction

GPS-enabled mobile devices (phones, car navigators, etc) are ubiquitous these days and it is common for them to report their locations to a server in order to access location-based services. Such services involve querying the current or near future locations of the mobile devices. Many index structures have been proposed over last decade to facilitate efficient query processing on moving objects (e.g.,[1, 2, 3, 4, 5, 6, 7]). However, none of these index structures exploits the important property of skewed velocity distributions. In most real world scenarios, objects travel predominantly along only a few directions due to the fixed underlying traveling infrastructure or routes. Examples include vehicles on road networks, flights, people walking on the streets, etc. Fig. 1(a) shows a portion of the road network of San Francisco, where most of the roads are along two directions. Fig. 1(b) shows a sample of velocity distribution of the cars travelling on the San Francisco road network. Every point (2-dimensional vector) in the figure represents the velocity of a car. It is clear that most of the cars are traveling along two dominant directions (axes).



(a) San Francisco road network (b) Velocity distribution of the cars

Figure 1: San Francisco road network and the cars' velocity distribution

The velocity distribution of objects in an index has a great impact on the rate at which the query search space expands. The search space expansion is either due to the tree nodes' minimum bounding rectangle (MBR) expansion (e.g., the TPR-

Email addresses: thi.nguyen.cs@gmail.com (Thi Nguyen), z.he@latrobe.edu.au (Zhen He), rui.zhang@unimelb.edu.au (Rui Zhang), phillipgdward@gmail.com (Philip G. D. Ward)

[1]Corresponding author.

[2]This work was done while the author was with the Department of Computer Science and Computer Engineering, La Trobe University, Australia.

tree/TPR*-tree [5, 6]) or query expansion (e.g., the B^x-tree [2]). In either case, the search space for a tree node is enlarged during the query time interval using the largest speed of the objects grouped in that tree node. If the velocities of the objects in a node are randomly distributed, then the search space is enlarged along both the x- and y-axes, and therefore there is a quadratic function of the maximum speed of the objects in the node. If the movements of all the objects in a node are largely along the same direction, then the search space is enlarged mainly along one axis and hence the search space expansion is close to a linear function of the maximum speed of the objects in the node.

Motivated by this observation, we propose the *velocity partitioning (VP)* technique, which exploits the skew in velocity distribution to speed up query processing using moving object indexes. The VP technique first identifies the "dominant velocity axes (DVAs)" using a combination of principal components analysis (PCA) and k-means clustering. A DVA is an axis to which the velocities of most of the objects are (almost) parallel. Then, a moving object index (e.g., a TPR*-tree) is created based on each DVA, using the DVA as an axis of the underlying coordinate system. Objects are dynamically moved between DVA indexes when their movement directions change from one DVA to another. Objects with current velocities, which are far from any DVAs, can be put in an outlier index. Thus, except for the outlier index, the objects in each other index are moving in a near 1-dimensional space instead of a 2-dimensional space. As a result, the expansion of the search space with time is greatly reduced, from a quadratic function of the maximum speed (of the objects in the search range) to a near linear function of the maximum speed.

The VP technique is a generic method and can be applied to a wide range of moving object index structures. In this paper, we focus our analysis and implementation of the VP technique on the two most well recognized and representative moving object indexes of different styles, the TPR*-tree [6] and the B^x-tree [2]. These two indexes are the basis for many recent indexing techniques [8, 9, 10, 7]. Our method can be applied to these more recent indexes in similar ways. We perform an extensive set of experiments using various real and synthetic data sets. The results show that the VP technique consistently improves the performance of both index structures.

We further improve the performance of the VP technique by exploiting the characteristics of the solid-state drive (SSD). We found that the system was I/O bound when the system used the hard-disk drive (HDD). However, the system was more CPU bound when fast modern SSDs are used. This meant that we need to redesign the system to be more CPU friendly when the SSD is used. In particular, the small outlier index accounts for a disproportionately high percentage of the CPU time (up to 60 %) when the SSD is used. To address this issue, we created an outlier index that uses significantly less CPU time called the RAM-resident compressed grid (RCG). Our experimental study shows that when using the SSD, the RCG outlier index can reduce the total execution time of the VP technique by up to a factor of 2 compared to the outlier index designed for minimizing disk I/O.

A preliminary version of this paper appeared in [11], in which we focused on improving range query performance by exploiting skewed velocity distribution using traditional HDDs. This paper extends the previous paper [11] in three aspects. First, we present an algorithm for the k-nearest neighbor (k-NN) query based on the VP technique (Sect. 5.5). Second, we propose the RCG index which improves the performance of the outlier index by exploiting the characteristics of SSDs, which are superior to HDDs (Sect. 6). Third, we extend our experimental evaluation to include k-NN queries (Sect. 7.9) and the impact of the RCG outlier index on the performance of the VP technique in a variety of situations (Sect. 7.11).

The contributions of this paper are summarized as follows:

- We analytically show why a moving object index with VP outperforms a moving object index without VP.
- We propose the VP technique, which identifies the dominant velocity axes (DVAs) and maintains the objects in separate indexes based on the DVAs.
- We analytically show how to choose the value of an important parameter that determines which objects belong to the outlier index.
- We propose the RCG outlier index which is specifically designed to achieve high performance when solid-state drives are used.
- We implemented the VP technique on two state-of-the-art moving object indexes, the TPR*-tree and the B^x-tree and performed an extensive experimental study. The results validate the effectiveness of our approach across a large number of real and synthetic data sets.

The remainder of this paper is organized as follows. Sect. 2 provides some preliminaries; Sect. 3 discusses related work; Sect. 4 analyzes how velocity partitioning reduces search space expansion; Sect. 5 details our velocity partitioning technique; Sect. 6 describes our RCG outlier index. Sect. 7 reports our experimental study; finally, Sect. 8 concludes this paper.

2. Preliminaries

In this section, we provide some background on moving objects and briefly review two techniques used in our approach, principal components analysis (PCA) and k-means clustering.

2.1. Moving object representation and querying

A simple way of tracking the location of moving objects is to take location samples periodically. However, this approach requires frequent location updates, which imposes a heavy workload on the system. A popular method to reduce the reporting rate is to use a linear function to describe the near future trajectory of moving objects. The model consists of the initial location of the object and a velocity vector. An update is issued by the object when its velocity changes. An object velocity update consists of a deletion followed by an insertion. This linear model-based approach is used by many studies

[1, 2, 3, 12, 4, 5, 6, 7, 13, 14] on indexing and querying moving objects. We also follow this model in this paper, and the moving objects are modeled as moving points.

We support three different types of range queries: the *time slice range query*, which reports the objects within the query range at a particular time unit; the *time interval range query*, which reports the objects within the query range within a time range; and the *moving range query*, where the query range itself is moving and the query reports the objects that intersect the moving range in a time range. For all three types of range queries, if the query time (or time range) is in the future, the query range is projected (expanded) to that future time to check which objects should be returned.

We also support the k-nearest neighbor (k-NN) query, which retrieves the k objects with the least distances from a given query point at a specified time unit.

2.2. Principal components analysis

Principal components analysis (PCA) is a commonly used method for *dimensionality reduction* [15, 16] and for finding correlations among attributes of data [17]. It examines the variance structure in the data set and determines the directions along which the data exhibits high variance. In our case, if we map the velocity of objects into the 2D velocity space as points, then the axis with high variance is the DVA.

Given a set of k-dimensional data points, PCA finds a ranked set of orthogonal k-dimensional eigenvectors v_1, v_2, ..., v_k (which we call principal component vectors) such that:

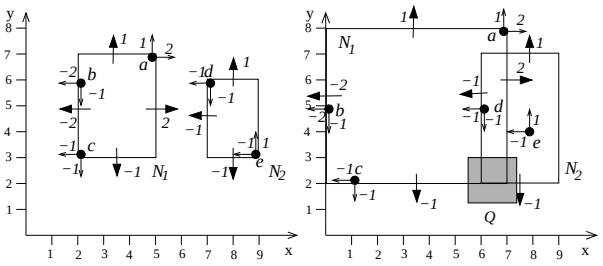
- Each principal component (PC) vector is a unit vector, i.e., $\sqrt{\beta_{i1}^2 + \beta_{i2}^2 + ... + \beta_{ik}^2} = 1$, where β_{ij} (i, j = 1,2, ...,k) is the j^{th} component of the PC vector v_i.
- The first PC v_1 accounts for most of the variability in the data, and each succeeding component accounts for as much of the remaining variability as possible.

2.3. K-means clustering

K-means clustering [18] is a commonly used method to automatically partition a data set into k clusters where each data point belongs to the cluster with the nearest *centroid*. It starts by assigning each object to one of k clusters either randomly or using some heuristic method. The centroid of each cluster is computed and each point is re-assigned to its closest cluster centroid. When all points have been assigned, the k cluster centroids are recomputed. The process is repeated until the centroids no longer move.

3. Related work

In this section, we review existing work on moving object indexes, specifically R-tree [19] based indexes, the B^x-tree [2], and dual transform-based indexes. We also discuss indexing techniques for handling skewed workloads and for handling moving objects on road networks.



(a) MBRs and VBRs at time 0 (b) MBRs and VBRs at time 1

Figure 2: MBRs of a TPR-tree growing with time

3.1. R-tree based moving object indexes

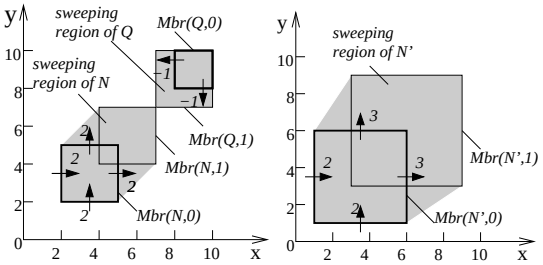
An established approach to index moving objects is to use the R-tree [19] or its more optimized variant the R*-tree [20] to index the extents of objects and their current velocities. These indexes include the TPR-tree [5] and its variant TPR*-tree [6], which optimize some operations of the TPR-tree. They work by grouping object extents at the current time into minimum bounding rectangles (MBRs). Fig. 2(a) shows the objects a, b and c grouped into the same MBR in node N_1. Accompanying the MBRs are the velocity bounding rectangles (VBRs), which represent the expansion of the MBRs with time according to the velocity vectors of the constituent objects. The rate of expansion in each direction is equal to the maximum velocity among the constituent objects in the corresponding direction. A negative velocity value implies that the velocity is towards the negative direction of the axis. For example, in Fig. 2(a) we can see that the solid arrow on the left of node N_1 has a value of -2. This is because the maximum velocity value of the constituent objects in the left direction is 2. Fig. 2(b) shows the expanded MBRs at time 1.

The MBR and VBR structure described can be extended by replacing the constituent object extents with smaller MBRs. This, when recursively applied, creates a hierarchical tree structure. The tree structure is identical to the classic R-tree [20], the only difference being the algorithms used to insert, delete and query the tree also need to take the velocity information into consideration. The TPR-tree and the TPR*-tree modify the R*-tree's insertion/deletion and query algorithms.

In this paper, we use the cost model proposed by Tao *et al.* [6] in the insertion and deletion algorithms of the TPR*-tree but use a different simpler and more general (applied to different types of trees) cost model to analyze the benefits of a partitioned index (see Sect. 4). The cost model of Tao *et al.* [6] reduces the expected number of nodes accessed by a range query Q. We briefly describe this cost model as follows.

Consider a moving tree node N and a moving range query Q for the time interval [0,1] as shown in Fig. 3(a). The MBR (VBR) of N is denoted as $N_R = \{N_{R1-}, N_{R1+}, N_{R2-}, N_{R2+}\}$ ($N_V = \{N_{V1-}, N_{V1+}, N_{V2-}, N_{V2+}\}$), where N_{Ri-} (N_{Vi-}) is the coordinate (velocity) of the lower boundary of N on the i^{th} dimension, where $i \in \{1, 2\}$. Similarly, N_{Ri+} (N_{Vi+}) refers to the upper boundary. MBR (VBR) of Q also can be denoted similar to N.

The *sweeping regions* of N and Q are the regions swept by N and Q during the time interval [0,1] (the grey regions shown in Fig. 3(a)). To determine whether node N intersects Q, we first define the *transformed node* N' with respect to Q as follows: the MBR of N' in the i^{th} dimension is $\langle N_{Ri-} - |Q_{Ri}|/2, N_{Ri+} + |Q_{Ri}|/2\rangle$; the VBR of N' in the i^{th} dimension is $\langle N_{Vi-} - Q_{Vi+}, N_{Vi+} - Q_{Vi-}\rangle$. To check whether node N intersects Q during the time interval [0,1] is equivalent to checking whether the transformed node N' intersects the center of Q (which is a point) during the time interval [0,1]. Therefore, the probability of N intersecting Q (which is the probability of node N being accessed by the query Q) during the time interval [0,1] is the same as the probability of N' intersecting the center of Q during the time interval [0,1], which is equal to the area of the sweeping region of N' in the time interval [0,1] (the grey region shown in Fig. 3(b)). Assume that the MBR of Q uniformly distributes in the data space and the data space has a unit extent in each dimension. Adding up this probability for every node of the tree, we obtain the expected number of node accesses for the range query Q as:



(a) Moving node N, Q (b) Transformed node N'

Figure 3: Sweeping region of moving node

$$\sum_{every\ node\ N\ in\ the\ tree} V_{N'}(q_T), \quad (1)$$

where q_T is the query time interval; $V_{N'}(q_T)$ is the volume of the sweeping region of N' during q_T.

3.2. The B^x-tree

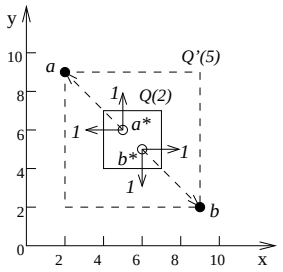


Figure 4: Query enlargement in the B^x-tree

The B^x-tree [2] indexes moving objects using the B$^+$-tree. This is a challenge because the B$^+$-tree indexes 1D space but objects move in a 2D space with associated velocities as well. The B^x-tree addresses the challenge by first partitioning the 2D space using a grid, and then using a space-filing curve (Hilbert-curve or Z-curve) to map the location of each grid cell to a 1D space where 2D proximity is approximately preserved. The locations of the moving objects are indexed relative to a common reference time.

The B^x-tree incorporates the fact that objects are moving by enlarging the query window according to the maximum velocity of the objects. If the query time is far in the future, and therefore very different from the index reference time, then the query may be enlarged significantly. Fig. 4 shows an example of how the window enlargement works. Supposing that the current time is 0, we issue a predictive time slice range query Q at time 2 (the solid rectangle). Given that moving points a and b (the black dots) stored in the B^x-tree, are indexed relative to time unit 5, from their velocities as shown in Fig. 4, we can infer their positions at time unit 2, which are $a*$ and $b*$ (the circles). The window enlargement technique enlarges the range query Q using the reverse velocities of a and b to obtain the query window at time unit 5 (the dashed rectangle). In practice, histograms on a grid-based are maintained for the maximum/minimum velocity of different portions of the data space and the query window is enlarged according to the maximum/minimum velocity in the region it covers. Therefore, a drawback of the B^x-tree is that, if only a few objects have a high speed, they would make the enlarged query window unnecessarily large for most of the objects.

To reduce the amount of query window enlargement, the B^x-tree partitions the index into multiple time buckets, where all objects indexed within the same time bucket are indexed using the same reference time. This results in a smaller difference between the reference time and query time and thus reduces the query window enlargement. When objects are updated, they are moved from the time bucket they are currently residing in to the future time bucket.

3.3. Dual transform-based moving object indexes

The earlier work on dual transform-based moving object indexes [21, 22] was improved upon by more recent indexes such as STRIPES [4], the B^{dual}-tree [7] and [3]. They index objects in the dual space, i.e., a 4-dimensional space consisting of two dimensions for the location of an object and another two dimensions for the velocity of the object. A consequence of indexing the velocity as separate dimensions is that the moving objects are effectively indexed as stationary objects. All objects are indexed based on the same reference time 0. A drawback of indexing all objects at the same reference time is that the query search space continues to grow with time, which is overcome by periodically replacing the old index with a new index with an updated reference time.

Dual transform-based moving object indexes differ from our work by not exploiting velocity distribution skew to index objects traveling along different *dominant velocity axes (DVAs)* separately.

3.4. Indexing techniques that handle skewed workloads

Zhang *et al.* [23] propose the P$^+$-tree, which efficiently handles both range and k-NN queries for different data distributions including skewed distributions. Their work differs from ours in that their index is designed for stationary objects instead of moving objects. Tzoumas *et al.* [10] propose the QU-Trade technique for indexing moving objects that adapts to varying query versus update distributions by building an adaptive layer on top of the R-tree or TPR-tree. Our work differs from this by adapting to velocity distributions instead of query versus update distributions. Chen *et al.* [8] propose the ST^{2}B-tree, which improves the B^x-tree by making it adaptive to data and query distribution. This is done by dynamically adjusting the reference points and grid sizes. Our work differs from this by creating separate indexes according to velocity distributions instead of adjusting the reference points and grid sizes. Our VP technique can be applied in a straightforward manner to the QU-trade technique and ST^{2}B-tree because their underlying structures are the TPR-tree and the B^x-tree, respectively.

Dittrich *et al.* [1] propose a RAM-resident indexing technique called MOVIES for moving objects. MOVIES assumes that the whole data set resides in RAM and the update rate is very high (greater than 5,000,000 per second), whereas our technique does not make such assumptions.

3.5. Indexing techniques for moving objects on networks

There are many existing papers [24, 25, 26, 27] which model the movement of objects along any type of network including road networks. Our paper does not assume that every object must move in a road network, in other words, our technique works for generic scenarios where objects can move freely. Objects moving in road networks is just one of the motivating examples in which case our technique brings great performance gain due to the few dominant directions of object movements.

3.6. Query processing on moving objects

Range and k nearest neighbor (k-NN) queries are two most common queries on moving object databases which have been extensively studied in the literature [2, 3, 4, 5, 7, 13, 28, 29, 30, 31]. In the recent years, new types of queries on moving objects have also been proposed to address many emerging applications. These include the reverse NN (RNN) and the reverse k-NN (Rk-NN) queries [32, 33], the k-NN queries with two predicates [34], the obstructed NN queries [35], the visible k-NN (Vk-NN) query [36], the moving k-NN query [37], and the intersection join query [14, 38]. Our proposed index fully supports range and k-NN queries and other emerging queries can be supported in future work.

4. How velocity partitioning reduces search space expansion

In this section, we analytically show how a velocity partitioned index can reduce the rate of search space expansion. We focus our analysis on the B^x-tree and the TPR-tree variants. We first give an intuitive description of a partitioned index versus unpartitioned index. Second, we define search space expansion. Third, we analytically contrast the rate of search space expansion between an unpartitioned index versus a partitioned index. Finally, we present preliminary experimental verification of our analysis.

Partitioned index. The main idea of the velocity partitioning (VP) technique is to index objects moving along different DVAs (directions) in separate indexes. It is important to note that the VP technique is not restricted to pairs of DVAs that are perpendicular to each other, but rather will work for any number of DVAs separated by any angle. Here we first use a simple example to illustrate the concept of the VP technique. Later in Sect. 5, we provide a detailed description of how the VP technique is performed. Fig. 5 shows an example of objects indexed by an unpartitioned index versus the same objects indexed by a partitioned index. In this example, objects are moving along two DVAs, the x-axis and the y-axis. In the unpartitioned index, all objects are indexed by the same index. In the partitioned index, objects moving along the x-axis are indexed in a separate index from those moving along the y-axis.

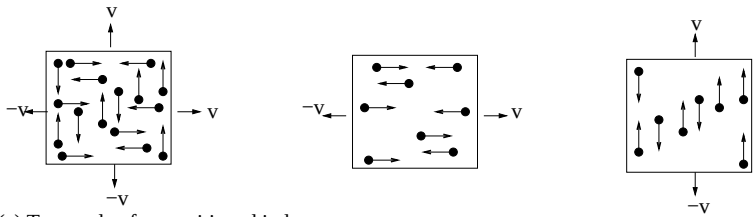
Search space expansion. First, we define what we mean by search space expansion. The search space for a query describes the data space that is covered (accessed) when processing the query. The expansion of the search space is determined by the relative movement between the query and the tree nodes. The size of the search space is proportional to the number of tree nodes accessed by a query Q, which can be estimated using a cost model proposed by Tao *et al.* [6] for the TPR-tree/TPR*-tree. The cost model was described in Sect. 3.1 and given as Eq. 1.

Although the cost model was designed for the TPR-tree, it also applies to the B^x-tree as follows. For the B^x-tree, the query expands but the tree nodes are stationary, which is a special case of the analysis used for Eq. 1 where both the query and the tree node are moving and expanding.

The idea behind the cost model of Eq. 1 is that we can always transform a moving/expanding query into a stationary one by making relative adjustments to tree nodes. For example, an expanding query and a stationary tree node can be transformed into a stationary query by expanding the tree node by the amount the query was supposed to expand. *Following this line of argument, we only consider the expansion of the tree node in the following analysis without loss of generality*.

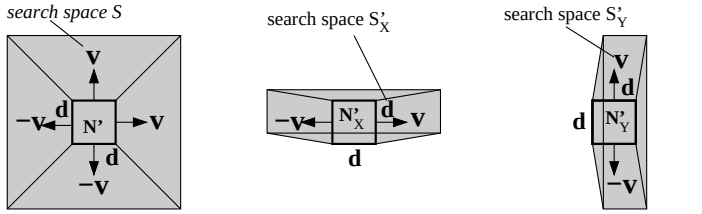
Fig. 6 shows an example of the search space of the example shown in Fig. 5. In the example, S is the search space of the *unpartitioned index*, S'_X and S'_Y are the search spaces of a *partitioned index* in the x- and y-axes, respectively. We also assume that all objects are traveling either along the x- or y-axes, as was the case for Fig. 5. The example shows that the search space expands by a quadratic factor for the unpartitioned index versus a linear factor for the partitioned index.

Analysis of search space expansion of unpartitioned versus partitioned index. We will first analyze a simplified scenario as shown in Fig. 6, and then discuss more general



(a) Tree node of unpartitioned index (b) Tree nodes of partitioned index

Figure 5: Objects indexed by an unpartitioned index versus the same objects indexed by a partitioned index



(a) Search space expansion in both x and y−axis (b) Search space expansion in x−axis (c) Search space expansion in y−axis

Figure 6: Search space of unpartitioned index, S versus search space of partitioned index, S'_X plus S'_Y

situations in Sect. 4.1. In this simplified scenario, we assume that: (i) the velocities of all the objects are exactly along the standard x- or y-axes; (ii) the objects travel at the same speed along all directions; (iii) the extent length of the tree nodes along the x- and y-axes are the same; and (iv) the initial locations of objects are uniformly distributed in the 2D space. The symbols used in Fig. 6 are described as follows. N' is the transformed rectangle of the node N with respect to the query for the unpartitioned index at the initial time 0; N'_X and N'_Y are the transformed rectangles of the node N for the partitioned index for the x- and y-axes, respectively; v is the maximum speed for the objects in S along both the x- and y-axes. The extent length of all the nodes is d. This assumption is reasonable since we are more interested in the rate of expansion of the search space rather than its initial size.

Let S' denote the combined search space of the unpartitioned index in the x-axis, S'_X and the y-axis, S'_Y (as shown in Figs. 6(b) and 6(c), respectively). Our aim is to show that the rate at which the unpartitioned search space, S expands is higher than the rate at which the partitioned search space S' expands. We quantify the search space as the volume created by integrating the search area from time 0 to the query predictive time t_h, where query predictive time refers to the future time of the query. The search area expands with time, therefore we start by expressing the search area of the partitioned index N' as a function of time t, $A_{N'}(t)$ as follows:

$$\begin{aligned} A_{N'}(t) &= (d + 2vt)(d + 2vt) \\ &= d^2 + 4vtd + 4v^2t^2 \end{aligned} \tag{2}$$

We are interested in the total expansion of the search area of the partitioned indexed including both the x-axis index and y-axis index. Therefore, let $A_{CN'}(t)$ be the combined area of N'_X and N'_Y as a function of time t. $A_{CN'}(t)$ can be computed as follows:

$$\begin{aligned} A_{CN'}(t) &= A_{N'_X}(t) + A_{N'_Y}(t) \\ &= (d + 2vt)d + d(d + 2vt) \\ &= 2d^2 + 4dvt \end{aligned} \tag{3}$$

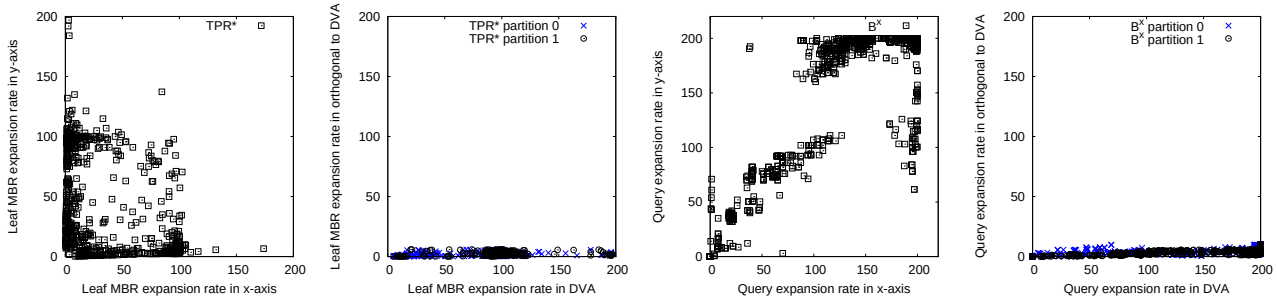
We next compute the search volume of S. It is important to compute the search volume rather than just the expanded search area since the volume includes the cumulative expansion of the area from time 0 to t_h. We compute the search volume V_S of S by integrating the search area $A_{N'}$ from time 0 to t_h as follows:

$$\begin{aligned} V_S(t_h) &= \int_0^{t_h} A_{N'}(t)\, dt \\ &= \int_0^{t_h} (d^2 + 4vtd + 4v^2t^2)\, dt \\ &= d^2t_h + 2dvt_h{}^2 + \frac{4}{3}v^2t_h{}^3 \end{aligned} \tag{4}$$

Similarly, the search space volume from time 0 to t_h of S', $V_{S'}$ can be computed as follows:

$$\begin{aligned} V_{S'}(t_h) &= \int_0^{t_h} A_{CN'}(t)\, dt \\ &= \int_0^{t_h} (2d^2 + 4dvt)\, dt \\ &= 2d^2t_h + 2dvt_h{}^2 \end{aligned} \tag{5}$$

In order to compare the search space of the partitioned index versus the unpartitioned index, we compute the difference between the search space volume of the partitioned search space



(a) Unpartitioned TPR*-tree (b) Partitioned TPR*-tree (c) Unpartitioned B^x-tree (d) Partitioned B^x-tree

Figure 7: Search space expansion of the unpartitioned versus partitioned B^x-tree and TPR*-tree on the Chicago data set. Note that the units for both the x-axis and the y-axis are m^2/s

S' versus the unpartitioned search space S as a function of time, $\Delta V(t_h)$ as follows:

$$\begin{aligned}\Delta V(t_h) &= V_{S'}(t_h) - V_S(t_h) \\ &= 2d^2t_h + 2dvt_h{}^2 - (d^2t_h + 2dvt_h{}^2 + \frac{4}{3}v^2t_h{}^3) \\ &= d^2t_h - \frac{4}{3}v^2t_h{}^3 \end{aligned} \quad (6)$$

From Eq. 6, we can see that as time increases the search volume of the unpartitioned space V_S becomes increasingly larger than the search volume of the partitioned space, $V_{S'}$. This can be seen by the fact $\Delta V(t_h)$ is negative when t_h is greater than $\frac{d\sqrt{3}}{2v}$. Therefore, when time t_h passes the $\frac{d\sqrt{3}}{2v}$ threshold the search volume of the unpartitioned search volume V_S becomes larger than the partitioned search volume $V_{S'}$.

Next, we analyze the rate of change in the search space, by taking the derivative of Eq. 6. This is stated as follows:

$$\frac{d\Delta V(t_h)}{dt_h} = d^2 - 4v^2t_h{}^2 \quad (7)$$

Eq. 7 shows that the search volume of the unpartitioned index expands at a much faster rate than the partitioned index. This can be seen by the fact the rate at which the search volume of the unpartitioned index increases above the partitioned index is a squared factor of both v and t_h because $\frac{d\Delta V(t_h)}{dt_h}$ is a squared factor of both v and t_h.

The above analysis is with respect to a single node. It obviously applies to any node in the tree and when summing up the search space for all the tree nodes, we reach the conclusion that the query search space on a partitioned index grows much slower with time than the query search space on an unpartitioned index. The following experiment on a real data set validates this result.

Experimental verification of the analysis. Fig. 7 shows the results of an experiment, which illustrates the 2D search space expansion for an unpartitioned TPR*-tree and an unpartitioned B^x-tree versus a near 1D search space expansion for their partitioned counterparts. The indexes are partitioned using our VP technique (detailed in Sect. 5). The experiment uses data generated from a portion of the road network of Chicago shown in

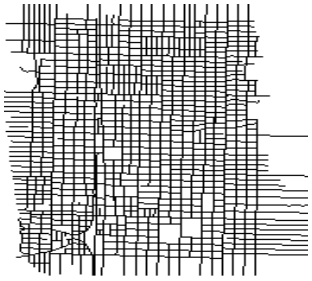


Figure 8: Chicago road network

Fig. 8.The experiment involved 100,000 moving objects, with maximum speed of 100 meters per second, with a query predictive time of 60 seconds. Details of other parameters of the experiment are the default parameters described in the experimental study (Sect. 7).

Figs. 7(a) and 7(b) show the velocity expansion rate of the leaf MBRs for the unpartitioned TPR*-tree and partitioned TPR*-tree, respectively. The results show that the leaf nodes of the unpartitioned TPR*-tree expand in a 2D space, whereas the partitioned TPR*-tree expands in a near 1D space. Similarly, Figs. 7(c) and 7(d) show the query expansion rate of the unpartitioned B^x-tree and partitioned B^x-tree, respectively. Again, the query of the unpartitioned B^x-tree expands in a 2D space, whereas the partitioned B^x-tree expands in a near 1D space.

4.1. Discussion of general cases

In the analysis of the simplified scenario, we have made several assumptions. To lift the first assumption, when the velocities of objects are not exactly along the standard x- or y-axes, as long as their directions are close to the standard x- or y-axes, the previous analysis still holds since a small deviation from the *dominant velocity axis (DVA)* incurs a small search space expansion. However, if some objects' directions are not close to any of the DVAs, we will put these objects into an outlier partition. Details of the outlier partition will be discussed in Sect. 5.2.

An implicit assumption we also made in the previous analysis is that there are two DVAs, one is vertical and the other is horizontal. This assumption may not hold in practice. Therefore,

in our *VP* technique, we first find out the actual DVAs (through a combination of PCA and k-means clustering). Then, the previous analysis still holds when we replace the x- and y-axes with the actual DVAs. Details of how to find the DVAs will be discussed in Sect. 5.1.

5. The velocity partitioning technique

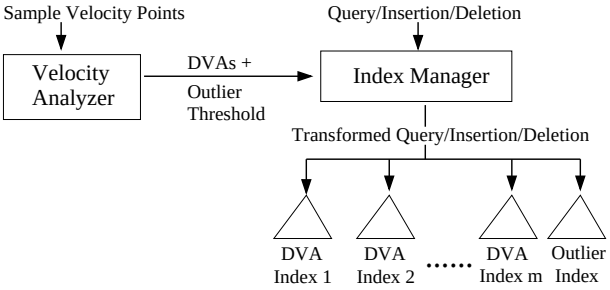


Figure 9: The system architecture of the VP technique

We present our *velocity partitioning (VP)* technique. Fig. 9 shows the system architecture for the VP technique. The system has two main components, a *velocity analyzer* and an *index manager*. The velocity analyzer *partitions a sample* of the velocity of objects from the current workload in order to find the DVAs and an outlier threshold (used to determine which objects belong to the outlier partition). Velocity is a 2D point in the velocity space, so we refer to the velocity of an object as a *velocity point*. The index manager takes the output of the velocity analyzer to transform the query, insertion and deletion operations to operate on the DVA indexes and outlier index. A DVA index is the same as a traditional moving object index such as the TPR-tree or the B^x-tree except objects are indexed using a transformed coordinate space according to the DVA. The index manager inserts an object into the closest DVA index unless it is far from all DVAs, in which case, the object is inserted into the outlier index. If an object update causes its direction of travel to change sufficiently, it may be moved from one index to another. Processing a query involves transforming the query into the coordinate space of each index, and then querying all the indexes and combining the results.

We provide a more detailed description of the velocity analyzer in this section since it is the key component of the system. The velocity analyzer analyzes the sample of velocity points to determine the partition boundaries for future object insertions and querying. The partition boundaries are determined by the DVAs in the data set and an outlier threshold τ. We observe that when there are multiple DVAs in the data set, using only PCA may not be sufficient to identify the DVAs correctly. Therefore, we propose to use a combination of PCA and k-means clustering on the sample velocity points to determine the DVAs. Here m is an input value given by the user based on observation of the data set or experience. For example, most road networks have two dominant traffic directions and we can set m to 2. Once the DVAs are determined, the objects can be partitioned based on the closeness of their velocity directions to the directions of the DVAs. However, some velocity points may not be close to any DVA. These objects are placed in an outlier partition. We determine the boundary of the outlier partition using a threshold τ, which defines an upper bound on what a DVA partition will accept. We choose the τ value for every partition by analyzing the sample data set using a search space-based cost function.

Algorithm 1 summarizes the *VP* algorithm used by the velocity analyzer. It starts by finding the DVAs using a combination of PCA and k-means clustering on the representative sample data (Line 2). Specifically, we integrate PCA into the clustering process itself by using PCA to guide the formation and refinement of clusters. At the end of the clustering process, each cluster contains the velocity points that form one DVA partition. *The 1st PC of each partition is the DVA for the partition.* The partitioning algorithm minimizes the perpendicular distance from each velocity point to the DVAs. The reason we minimize the perpendicular distance is that if all velocity points within one partition have a small perpendicular distance to the DVA, then those velocity points occupy a near 1D space.

We define a *threshold* τ for every DVA to determine whether an object can be accepted to its partition (Line 4). We determine the optimal τ by minimizing the combined rate of search area expansion of the DVA partition and the outlier partition. Objects whose perpendicular velocity is not within the threshold, τ , of any DVA, are placed in the outlier partition (Line 5). Once all the outlier velocity points have been removed from the DVA partition we recompute the DVA using the remaining velocity points (Line 6). This updated DVA will be a more precise representation of the velocity points now remaining in the DVA partition. The final DVAs and their associated τ thresholds are used by the index manager for future insertions and query processing.

Algorithm 1: VelocityPartitioning(A,m)

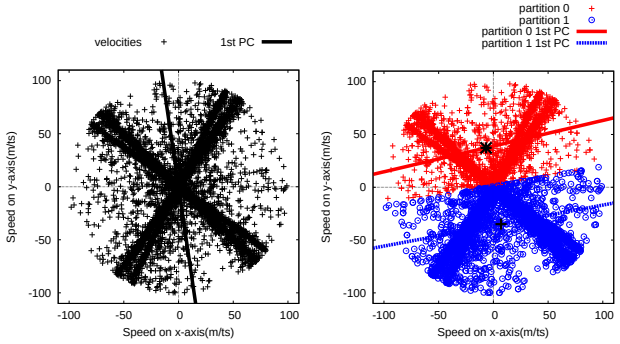
Input: A: sample set of velocity points, m: number of DVA partitions
Output: D: set of DVAs with associated outlier thresholds τ

1 let P be the set of m DVA partitions with their associated DVAs
2 P = Find DVAs(A, m) // See Algorithm 2
3 **for** *each* $p \in P$ **do**
4 **compute** the maximum perpendicular distance threshold τ for p according to Sect. 5.2
5 **move** the velocity points from p whose perpendicular distance is greater than τ from the DVA of p into the outlier partition
6 **recompute** the DVA for the remaining velocity points in p
7 let D be the set of DVAs and associated τ thresholds of P
8 **return** D

In Sect. 5.1, we describe how our velocity analyzer finds DVAs. In Sect. 5.2, we describe how our velocity analyzer determines the threshold τ to decide which objects should be placed in the outlier partition. In Sect. 5.3, we show how our

index manager handles insertion, deletion and update operations. In Sect. 5.4, we show how our index manager performs the range query. Sect 5.5 describes how the index manager handles k-NN queries. Finally, in Sect. 5.6, we discuss the issue of changing velocity distributions.

5.1. *Velocity analyzer: finding dominant velocity axes (DVAs)*



(a) Apply PCA to all data

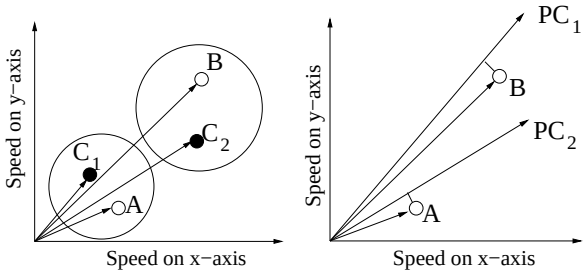
(b) Apply k-means (based on distance to centroid) to find clusters

Figure 10: Result of applying the two naïve approaches to finding the DVAs for the San Francisco data set

In this subsection, we will first examine two naïve approaches to finding DVAs, and then present our approach for finding DVAs.

Naïve approach I: PCA. The first naïve approach is to apply PCA on a sample set of velocity points to find the DVAs. Using PCA to find DVAs is intuitive, since the 1st PC (as described in Sect. 2.2) represents the principal axis along which the data points lay. In our case, the data points are *velocity* points, therefore, the 1st PC represents the principal axis along which objects travel. However, this approach effectively combines the multiple DVAs in the data set into one average velocity axis, which does not represent any of the individual DVAs. PCA is only useful for finding the DVA when there is only one DVA in the data set. Fig. 10(a) shows the results of applying PCA on a sample of 10,000 velocity points of cars traveling on San Francisco road network (shown in Fig. 1). In this case, the data set has two DVAs but the 1st PC is the average of the two, instead of the two individual DVAs. The 1st PC is far from either of the DVAs. The 2nd PC is orthogonal to the 1st PC and also does not correspond to any of the DVAs.

Naïve approach II: k-means clustering based on distance to centroid followed by PCA on each cluster. The second naïve approach applies k-means clustering to the *velocity* points based on distance to a cluster centroid and then uses PCA on each resultant cluster to create one DVA per cluster. This does not work well since it groups objects based on their closeness to a *point* (cluster centroid) rather than closeness to an *axis* (dominant axis). Fig. 12(a) shows an example of clustering based on distance to centroid. In the example, there



(a) Clustering using naïve approach II

(b) Clustering using our approach

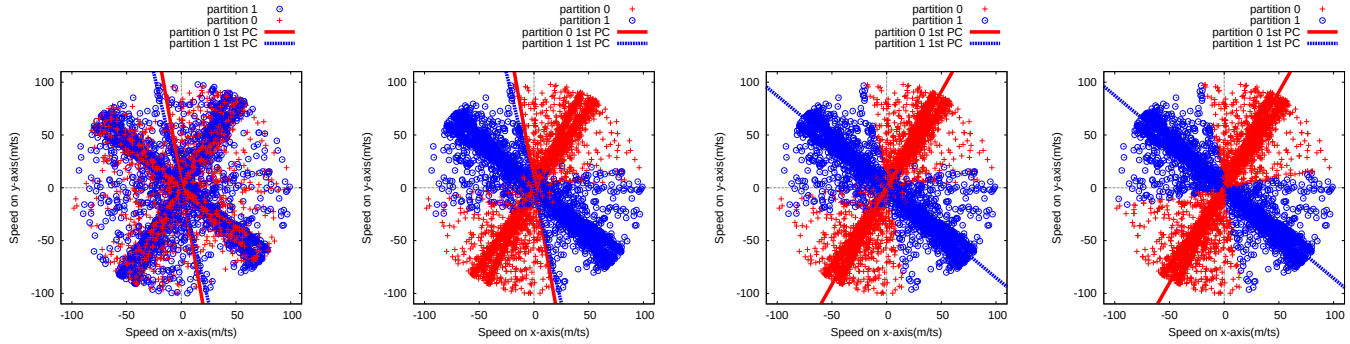
Figure 12: Naïve approach II versus our approach

are two cluster centroids C_1 and C_2 and two objects A and B. The direction of travel of object B is more aligned to C_1 than C_2, however the clustering algorithm groups object B with C_2 since B is closer to C_2. Similar observations can be made for object A. Fig. 10(b) shows the resultant clusters and corresponding DVAs found on the San Francisco dataset when using k-means clustering, where distance to centroid is used as the distance measure. Note that the two DVAs found (two parallel lines in Fig. 10(b) labeled as 1st PC of partition 0 and 1) by this technique do not resemble the two dominant axes (two axes with the highest concentration of data points) of the data set. The reason is the clusters created center around the cluster centroids shown in Fig. 10(b) instead of the dominant axes.

Our approach: k-means clustering based on distance to the 1st PC of each cluster. In our approach, we use k-means clustering on the velocity points, like the naïve approach II, but we use the perpendicular distance to the 1st PC of each cluster (partition) as the distance measure, instead of distance to a centroid. This allows objects to be clustered based on their direction of travel. Fig. 12(b) shows an example of using our clustering approach, where there are two clusters with their 1st PCs being PC_1 and PC_2, respectively. Our algorithm allocates object A to the cluster corresponding to PC_2 because A has a shorter perpendicular distance to PC_2. Similarly, object B is placed in the cluster corresponding to PC_1. This assignment of objects to clusters makes sense since the direction of travel for object A is more aligned to PC_2 than PC_1, similarly for object B.

Algorithm 2 shows precisely how our k-means clustering algorithm based on distance to the 1st PC is used to find DVAs.

Fig. 11 shows an example of applying the FindDVAs algorithm with m = 2 to the San Francisco data set of Fig. 1. Fig. 11(a) shows the initial random partitions and their corresponding 1st PCs (Lines 3-4 and 6). Note that although the two initial partitions are randomly created, their two 1st PCs are slightly apart. Next, Fig. 11(b) shows the partitions created after reassigning velocity points to their closest 1st PCs. Note that after this 1st reassignment iteration, the partitions already closely resemble the final partitions shown in Fig. 11(d). The reason for this is the reassignment of points amplifies the



(a) Partitions after initial random cluster assignment of points, and the 1st PC of each cluster.

(b) Partitions after the first iteration of clustering based on the distance to the 1st PC a cluster.

(c) Partitions and their 1st PCs after recomputing 1st PCs using PCA

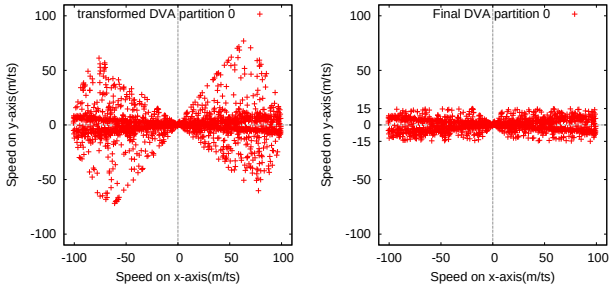
(d) Final partitions and DVAs

Figure 11: Our partitioning algorithm being applied to the San Francisco data set shown in Fig. 1

Algorithm 2: FindDVAs(A, m)

Input: A: set of velocity points, m: number of partitions
Output: P: set of partitions with associated 1st PC

1 let P be the set of m partitions
2 **initialize** each partition $p \in P$ to be empty
3 **for** *each velocity point* $a \in A$ **do**
4 randomly **assign** a into a partition $p \in P$
5 **while** *at least one velocity point has moved into a different partition* **do**
6 **compute** the 1st PC for each partition in P using PCA
7 **for** *each velocity point* $a \in A$ **do**
8 **if** *a is not currently in the partition whose 1st PC has the shortest distance from a* **then**
9 **move** a into partition whose 1st PC has the shortest distance from a
10 **return** P and associated 1st PC as the DVA partitions and their associated DVAs



(a) Transformed DVA partition 0

(b) Final DVA partition 0 after removing the outliers

Figure 13: The transformed DVA partition 0 and its final DVA partition after removing outliers

difference between the two 1st PCs by putting points that are slightly closer to one of the 1st PCs in the partition of that 1st PC. Fig. 11(c) shows the updated 1st PC of the partitions after reassigning velocity points (Line 6). The algorithm continues refining velocity points until they converge to the final partitions with their corresponding 1st PC (DVAs) as shown in Fig. 11(d).

5.2. *Velocity analyzer: the outlier partition*

Our aim is to have all objects within each partition travelling in a near 1D space. However, from Fig. 13(a) we can see that the data points, when transformed into the coordinate space formed by DVA 0 of Fig. 11, do not travel in a near 1D space due to the presence of outlier objects. To moderate the influence of these objects, we place these data points with a perpendicular distance above a *threshold* τ from their DVAs into the outlier partition. A cost analysis is performed upon each DVA partition separately to assign individual τ values to each DVA partition. The outlier partition is indexed in the standard coordinate system since the objects in it have little correlation with any DVAs.

We determine the optimal τ value using a slightly simplified version of the search space metric defined at the beginning of Sect. 4. More specifically, we use the minimum total rate of expansion of the area of the transformed leaf nodes $A_{N'_d}$ and $A_{N'_o}$ of the DVA and outlier partitions, respectively. We use the same process as that shown at the beginning of Sect. 4 to transform the velocities of the queries into the tree nodes. This minimization metric captures the change in the search area as a function of time. We focus our analysis on leaf nodes since non-leaf nodes are typically cached in the RAM buffer, the majority of RAM buffer misses being due to leaf node accesses.

In this section, we use an approximate cost model for deriving the optimal value of τ. Specifically, we assume the dimensions d of the leaf nodes of the DVA indexes are independent of the number of objects n_d in the DVA partition. In addition, we also assume that the maximum velocity v_{xmax} of the objects in the outlier partition is independent of n_d. These assumptions do not hold in general for the TPR-tree. However, the assumption that v_{xmax} is independent of n_d is a good approximation because our algorithm only takes objects that have high v_{y_d} (velocity of object in y-axis), not high v_{x_d} (velocity of ob-

ject in x-axis), from the DVA index into the outlier index. This has a negligible impact upon the v_{xmax} of the remaining objects since we are effectively taking a uniform random sample in terms of v_{x_d}. The assumption that d is independent of n_d is a coarser approximation. As we keep the average number of objects in each leaf node (n_l) constant, whilst removing objects (n_d), in general the MBRs will be larger as the remaining objects will be less dense. In general, we made these assumptions to keep our model simple and easy to be used for both TPR*-trees and B^x-trees. We found that in practice our simplified cost model performed well for both the TPR-tree and B^x-tree DVA indexes. Finally, we model v_{y_d} as a function of n_d because our algorithm is designed to move objects with the highest v_{y_d} into the outlier index.

For a given DVA partition and an outlier partition, we define the total rate of expansion of the area of the transformed leaf nodes of the two partitions as follows:

$$\begin{aligned} TA(t, n_d) &= L_d A_{N'_d}(t) + L_o A_{N'_o}(t) \\ &= \frac{n_d}{n_l}(d + 2v_{xmax}t)(d + 2v_{y_d}(n_d)t) \\ &+ \frac{(n - n_d)}{n_l}(d + 2v_{xmax}t)(d + 2v_{ymax}t) \end{aligned} \quad (8)$$

where L_d and L_o are the number of leaf nodes in the DVA and outlier partitions, respectively, n is the total number of objects in both partitions, n_d is the number of objects in the DVA partition and n_l is the average number of objects per leaf node. Fig. 14 illustrates the other terms used on the equation diagrammatically. The most important term is $v_{y_d}(n_d)$, since this is the term that corresponds to the threshold value τ. $v_{y_d}(n_d)$ is the maximum speed along the y-axis in the DVA partition. $v_{y_d}(n_d)$ is a function of n_d as we adjust $v_{y_d}(n_d)$ by removing from the DVA partition the objects whose y component speed is the highest. The remaining terms are described as follows. d is the length along both the x- and y-axes of both N'_d and N'_o. We use the same d for all side lengths because we assume uniform distribution of object locations. v_{xmax} and v_{ymax} are the maximum speed of N'_o along the x- and y-axes, respectively. For simplicity, we also suppose that the maximum speed of N'_d along the x-axis is also v_{xmax}. This approximation is reasonable since we partition solely based on the y-axis maximum speed and therefore we assume that the maximum speed of object movements along the x-axis is approximately the same for all partitions.

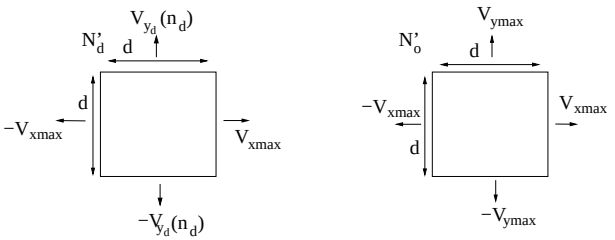


Figure 14: Diagram used to illustrate the terms used in Eq. 8

Next, we take the derivative of $TA(t, n_d)$ with respect to t to quantify the rate of expansion of $TA(t, n_d)$:

$$\begin{aligned} \frac{d\,TA(t, n_d)}{dt} &= \frac{2n_d}{n_l}((v_{y_d}(n_d) - v_{ymax})(d + 4v_{xmax}t)) \\ &+ \frac{2n}{n_l}(dv_{ymax} + v_{xmax}(d + 4v_{ymax}t)) \end{aligned} \quad (9)$$

We need to minimize Eq. 9 in order to minimize the rate of $TA(t, n_d)$ expansion. The only components of the equation that are not constant are n_d and $v_{y_d}(n_d)$. Therefore, minimizing Eq. 9 is same as minimizing the following expression:

$$n_d(v_{y_d}(n_d) - v_{ymax}) \quad (10)$$

Algorithm for determining optimal τ value. To find the n_d value that minimizes Eq. 10 analytically, we would need to have an equation describing $v_{y_d}(n_d)$. However, it is hard to find a general form for the $v_{y_d}(n_d)$ equation because it is data distribution dependent. Therefore, we use an equal width cumulative frequency histogram, per DVA partition, to capture the data distribution of $v_{y_d}(n_d)$. Each bucket of the histogram stores the number of velocity points in the DVA whose maximum y speed is the corresponding y speed of the bucket.

Our algorithm finds the τ threshold for each DVA partition by taking an uniform sample of $v_{y_d}(n_d)$ values and computing the corresponding Eq. 10 value. The $v_{y_d}(n_d)$ value giving the minimum value for Eq. 10 is used as τ. This approach incurs a small computational cost since Eq. 10 is simple and can be computed cheaply. Fig. 13(b) shows the final DVA partition 0 after removing outliers from the transformed partition shown in Fig. 13(a).

Our experimental study (Sect. 7.1) shows that the algorithm proposed above is able to find a close to optimal perpendicular distance τ value for both the B^x-tree and the TPR*-tree.

5.3. Index manager: insertion, deletion and update

The insertion algorithm is relatively straightforward. First, the algorithm finds the DVA index i_{min} whose perpendicular distance from the object o is the smallest. Then, if the perpendicular distance of o to i_{min} is larger than τ, then o is inserted into the outlier index otherwise o is inserted into i_{min}. Before an object is inserted into i_{min}, o is first transformed into the coordinate space of i_{min} using i_{min}'s 1st PC. The transformation process involves a simple matrix multiplication between the coordinates of o and the 1st PC of i_{min}.

When performing deletion, the algorithm first finds the partition object o resides in via a simple lookup table, and then uses the base index structure's deletion algorithm to delete the object from its partition. When an object changes its velocity, an update is performed on the index.

An update consists of a deletion followed by an insertion. The updated object will be inserted into the closest DVA index which may be different from its original DVA index. If an update involves moving an object from one DVA index to another, then both indexes need to be locked at the beginning of

the update to ensure a concurrent query on the destination index does not miss the inserted object. This may slightly increase the locking overhead.

5.4. Index manager: range queries

Algorithm 3: RangeQuery(I, q)

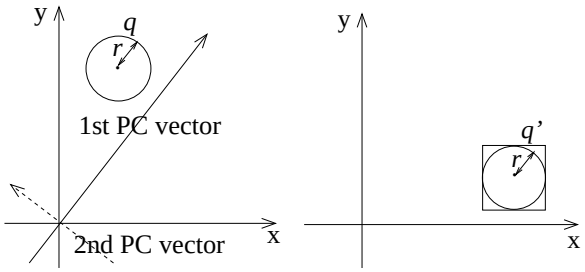
Input: I: set of all indexes including both DVA indexes and the outlier index, q: range query
Output: RS: result set

1 **for** *each index* $i \in I$ **do**
2 **if** i *is a DVA index* **then**
3 **transform** the range of q to the coordinate space of index i using the 1st PC of i
4 **create** transformed query q' consisting of a rectangular axis-aligned MBR of the transformed range of q
5 **else**
6 $q' = q$ // index i is the outlier index
7 **execute** range query q' on index i and store results in URS
8 **filter out** the objects in URS, which are not contained in q and **add** the remaining objects into RS
9 **return** RS

We present the range query algorithm, which can be used for both circular and rectangular range queries. Algorithm 3 details the steps the index manager uses to execute the range query. The index manager needs to query each of the indexes separately and merge the results as the query region may encompass objects from different indexes. Before querying each DVA index, we need to first transform the query range to the coordinate space of the DVA index using the 1st PCs of the DVA index (Line 3). The transformation process involves simple matrix multiplication between the coordinates of the query range and that of the 1st PCs. The transformed ranges are bounded by a rectangular minimum bounding region (MBR), which is axis-aligned with the coordinate space of the DVA indexes (Line 4). The transformed query is then executed on the indexes using the query algorithm of the underlying index, such as the B^x-tree and the TPR*-tree (Line 7). Finally, the objects in the result are filtered to remove any objects, which are in the MBR of the transformed query but not in the original query region (Line 8). Note that when querying the outlier index, there is no query transformation needed since the outlier index uses the standard coordinate system (Line 6).

Fig. 15(a) shows an example of a circular range query q with radius r before transforming into the coordinate space of a DVA index. It also represents the 1st and the 2nd PCs of the DVA index. Fig. 15(b) shows the transformed query q', which is bounded by an axis aligned MBR in the coordinate space of the DVA index formed by the 1st PCs.

Our system supports all three query types described in Sect. 2.1, namely, the time slice range query, time interval range query, and moving range query. We discuss the moving range query since it is the most general form of the three query types. After transforming the range query into the transformed coordinate system and applying the filtering step (Line 9 of Algorithm 3), the same object containment relationship with the original query is retained. The query velocity can also be transformed into the new coordinate system and the query can be executed in the standard way. Thus, our system supports the same query types as the underlying indexes (the B^x-tree/the TPR*-tree), including the three query types discussed in Sect. 2.1.



(a) Before transformation (b) After transformation

Figure 15: Circular range query before and after transforming into a DVA index's coordinate space

5.5. Index manager: k-NN queries

We describe how the index manager handles k-NN queries on DVA and outlier indexes. Assuming a set of objects $N > k$, given a moving point q_p, and a query time t_q, a k-NN query retrieves a set of k-nearest objects (RS) from q_p at time t_q. Algorithm 4 presents the details of the algorithm. The index manager retrieves RS by iteratively performing range queries with an incrementally increasing search region on each of the indexes separately until the exact k-nearest objects in RS are found. Similar to the RangeQuery algorithm (Algorithm 3), before querying each DVA index, we first need to transform the query point q_p to the coordinate system of the DVA index (Lines 8 - 9). Then, an initial range query is conducted which is a region centered at q_p with radius $r = D_k/k$ (Line 12), where D_k is the estimated distance between the query object and its k^{th}-nearest neighbor. D_k is estimated using the equation below (the same as that used by the B^x-tree [2]):

$$D_k = \frac{2}{\sqrt{\pi}}[1 - \sqrt{1 - \sqrt{k/N}}] \tag{11}$$

After querying each index, if the index manager has found the exact k objects in RS, then the index manager reduces the search region to the maximum distance of current k objects in RS (Line 15) which helps to reduce the query cost significantly. After querying all indexes, if the number of objects in RS is less than k, the index manager increases the search region by the initial search radius D_k/k (Line 19), then repeats the process of querying all indexes. Otherwise, the index manager returns RS as the results of the query.

Algorithm 4: k-NNQuery(I, q_p, k, t_q)

Input: I: set of all indexes including both DVA indexes and the outlier index, q_p: query point, k: number of neighbors, t_q: query time

Output: RS: result set of k-nearest neighbors with respect of $\langle q_p, k, t_q \rangle$

```
1  Let q'_p be query point q_p in the transformed coordinate space of the DVA index
2  Let r be the estimated search radius
3  Let dist(RS) be maximum distance of all objects in RS
4  r = D_k/k
5  kNNFound = false
6  while kNNFound == false do
7      for each index i ∈ I do
8          if i is a DVA index then
9              transform q_p to the coordinate space of index i using the 1st PC of i denoted as q'_p
10         else
11             q'_p = q_p
12         construct a range query q centered at q'_p with radius r and query time t_q
13         execute range query q on index i and update results in RS
14         if |RS| == k then
15             r = dist(RS) /* reduce search space by maximum distance of RS */
16     if |RS| == k then
17         kNNFound = true
18     else
19         increase r by D_k/k
20 return RS
```

5.6. Handling changing velocity distributions

In theory, if the dominant direction of object travel changes significantly, we would need to rerun the velocity analyzer to determine new DVAs, and then readjust the indexes to align with the new DVAs. However, in real life, we find that the direction component of the velocity distribution changes little since the routes of the moving objects are usually fixed. This is intuitive as velocity distributions are usually dictated by rarely changing environmental factors, such as road networks, flight paths and shipping lanes, etc. Therefore, the dominant direction of object travel is likely to be stable. However, the speed component of the velocity distribution is likely to change with time. For example, during the morning rush hour there will be many cars traveling into the city, resulting in reduced speed. In contrast, during this time, there will be few cars moving out of the city and they will be moving fast. The opposite is true during afternoon rush hour. The speed distribution has no effect on the coordinate system of the DVA indexes since the cars still travel along the same DVA. However, it does affect the value of the threshold τ, since τ is determined by the y-axis speed distribution of objects moving in the transformed coordinate system of the DVA indexes. We handle this situation by continuous updating the histogram used to determine τ, and then periodically computing an updated τ. Computing τ incurs only a small computational overhead because the equation used to derive it is simple.

5.7. Complexity Analysis

In this section, we present the complexity analysis of the VelocityPartitioning algorithm (Algorithm 1). The VelocityPartitioning algorithm first calls the FindDVAs algorithm (Algorithm 2) in Line 2. The complexity of the FindDVAs algorithm is $\mathrm{O}(n + i(m(d^2n + d^3) + nm))$, where n is the number of data points, d is the number of dimensions of the data, m is the number DVA indexes, and i is the number of iterations of the while loop (Lines 5-9) in Algorithm 2. The first n term is for Line 3 and 4 of the algorithm. The complexity of the PCA (Line 6) is $d^2n + d^3$, where the first d^2n term is the complexity of computing the co-variance matrix and second d^3 term is the complexity of the eigenvalues decomposition. Finally, the nm term is for Lines 7-9 of the algorithm. We can simplify the complexity of the FindDVAs algorithm to $\mathrm{O}(i\ m\ n)$. This is because the PCA computation is the dominant cost in the while loop and d equals to 2 since we are working in 2-dimensional data, so d can be considered as a constant.

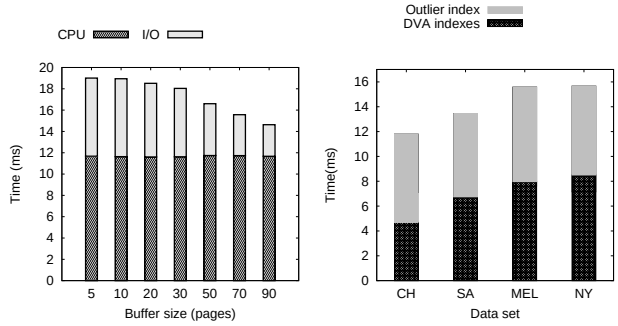
The complexity of the remainder of the VelocityPartitioning algorithm (Lines 3-6) is $\mathrm{O}(m(n + n + (d^2n + d^3))$ The first n term is the complexity of Line 4, the second n is the complexity of Line 5 and finally the final $(d^2n + d^3)$ term is the cost of recomputing the DVA, which involves computing the PCA. So again the complexity of Lines 3-4 can be reduced to the following $\mathrm{O}(m\ n)$ because d is just a constant 2 for our problem.

So the final overall complexity of the VelocityPartitioning algorithm is $\mathrm{O}(i\ m\ n)$. The m term (number of DVA partitions) in usually quite small, in our experiments it is just 2. Hence, the cost is mainly dominated by the number of iterations i and the number of data points n.

6. Improving the performance of the outlier index for SSDs

SSDs have become prevalent in recent years due to their superior performance, reliability and rapidly reducing prices. Our experimental study in Fig. 16(a) shows that, when a SSD is used and the VP technique is applied to the B^x-tree, the CPU time dominates the total execution time (between 61 and 80 %). Furthermore, the outlier index (described in Sect. 5.2) occupies a disproportionately high percentage of that CPU time (up to 60 %) (See Fig. 16(b)). In this section, we improve the performance of the outlier index by making use of SSDs. We have designed an index structure for the outliers which is more suited to the characteristics of SSDs.

The idea is to reduce CPU time by completely caching outliers in a RAM-resident grid structure. The grid structure has been found to yield better query performance for moving object indexes that reside completely in RAM compared to tree-based index structures [28, 39, 40, 29]. Further, we compress



(a) Range query time using the SSD for CH data set

(b) Range query CPU time

Figure 16: (a) Shows the results of breaking down the total time into CPU time and IO time when the SSD is used. (b) Shows CPU time being broken further down between DVA index and outlier index for various real data sets. The above results are for the VP technique applied to the B^x-tree and the default settings in Table 2 were used for these experiments.

the indexed objects to make more efficient use of RAM. To ensure the RCG outlier index does not occupy the entire RAM buffer, we allocate only a portion of the RAM buffer for the RCG outlier index and then adjust the velocity threshold τ to ensure the RCG outlier index stays within the allocated portion of the RAM buffer.

6.1. Index structure

We use a simple grid data structure to store outliers. Each cell of the grid stores compressed object information indexed based on object location. In order to handle the time dimension, we take the same approach as the B^x-tree [2], namely, expanding the query by the maximum velocity of the objects within the grid. In order to minimize the query expansion, we create multiple time buckets in the same way as in the B^x-tree, where each time bucket corresponds to a time interval (T_M). One grid index is created per time bucket. The idea is by keeping the time interval T_M small, the queries will expand less.

6.2. Update, insertion, and deletion

We use the same approach as for the B^x-tree for determining which time bucket an object should be deleted from and inserted into during an update. We give an example to illustrate this as follows.

Fig. 17 shows two time buckets (TB1 and TB2) being used to store the outlier objects. All outlier objects whose latest update time falls within the same time bucket are inserted into the same time bucket. An update involves deleting an object from the time bucket into which the object was inserted and then inserting the updated object to the current time bucket. Initially, during the time bucket [0, T_M), all the insertions and deletions are performed only on TB1, and TB2 remains empty. After time T_M and until time $2T_M$, all insertions are performed in TB2 and deletions remove objects either in TB1 or TB2, depending on whether these deleted objects were inserted to TB1 or TB2. After time unit $2T_M$, TB1 is empty and all outliers are stored in TB2. During the time bucket [$2T_M$, $3T_M$), TB1 and

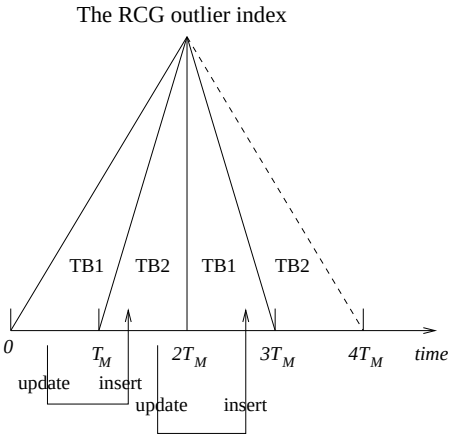


Figure 17: The RCG outlier index

TB2 swap states, i.e., all insertions are performed in TB1 and deletions remove objects either in TB1 or TB2 and so on.

Once we have selected a time bucket for insertion, we find the location of the object at the reference time for the selected time bucket and then store the object in the corresponding cell of the grid, where the reference time for a time bucket is the end of the time interval for the time bucket. For example, in Fig. 17 the reference time for TB1 is T_M.

Fig. 18 shows an example of inserting an object a at update time 2 into a time bucket with reference time 5. First, a is converted to the reference time 5 shown as $a*$. Then, $a*$ is inserted into the grid cell which contains the location of the object at time unit 5 ([2,2] of the 2D grid).

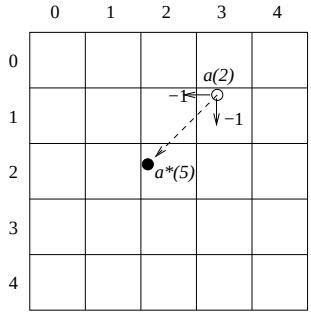


Figure 18: An example of inserting an outlier into a time bucket of a RCG outlier index

6.3. Query

Given a query, all time buckets are searched, and the results are combined to produce the final answer. Before querying each time bucket, we need to expand the query region using the maximum velocity of all objects within the time bucket. Several techniques [2, 41] have been proposed to reduce query expansion by maintaining a velocity histogram for each time bucket. However, for simplicity, we do not adopt these methods in this paper.

This general approach for querying the RCG outlier index can be used for both range and k-NN queries.

ID	x-$location$	y-$location$	x-$velocity$	y-$velocity$
8243	48078.6323	31057.8730	18.2624	-43.2345
13	48053.4567	31011.2460	7.5542	5.8162
432	48032.5434	31094.8437	-35.5001	-5.9782

Table 1: Example of object information stored in each grid cell

6.4. Index compression

We describe a simple approach to compress the information of outlier objects within each grid cell. The basic idea is to store the outlier objects' information column-by-column. Therefore, information on all objects of the same attribute are stored together. Data stored column-by-column is more compressible than data stored row-by-row due to the lower entropy of objects within the same column [42, 43, 44].

Table 1 shows an example of the object information stored in a grid cell before compression. In this example, the minimum and maximum coordinates of the grid cell are [48000, 31000] and [49000, 32000] respectively. The information stored per object includes an ID, a location in x-axis, a location in y-axis, a velocity in x-axis, a velocity in y-axis denoted as ID, x-$location$, y-$location$, x-$velocity$, y-$velocity$, respectively. We assume that IDs are 32-bit integer numbers and locations and velocities are 32-bit floating point numbers. However, our compression technique can be easily extended to work with 64-bit values.

6.4.1. ID compression

We describe our simple compression algorithm for the ID column. Fig. 19 shows the 32-bit binary representation of the ID's column shown in Table 1. We group bits into blocks of four to reduce computation costs. We then look at the number of leading zero blocks of each ID. We remove zb leading zero blocks from all IDs, where zb is the number of leading zero blocks in the ID that has the least number of leading zero blocks.

0000	0000	0000	0000	0010	0000	0011	0011_2 (8243_{10})
0000	0000	0000	0000	0000	0000	0000	1101_2 (13_{10})
0000	0000	0000	0000	0000	0001	1011	0000_2 (432_{10})

Figure 19: Example of applying ID compression on the ID column

6.4.2. Location compression

We store the relative locations of outlier objects in terms of minimum x- and y-coordinates of the containing cell. The relative locations are smaller in magnitude compared to the absolute locations and therefore can be compressed more. Fig. 20 shows the relative values of x-$location$ column after subtracting the grid cell's lower bound and their 32-bit binary representation. Next, we find the longest sequence of leading 4-bit blocks which is the same for all location values and store these values once for all the objects within the same cell.

0100	0010	1001	1101	0100	0011	1011	1101_2 (78.6323_{10})
0100	0010	0101	0101	1101	0011	1010	1001_2 (53.4567_{10})
0100	0010	0000	0010	0010	1100	0111	0001_2 (32.5434_{10})

Figure 20: Example of applying location compression on the x-$location$ column

6.4.3. Velocity compression

We avoid a mixture of negative and positive velocities because we want the first bit of the velocity value to be the same for ease of compression. We achieve this by adding the maximum speed to all velocity values. For example, if the maximum speed of all objects in a data set is 100.0, then velocity values in any dimension range from 0.0 to 200.0. We compress the common leading 4-bit blocks using the same method as location compression.

6.4.4. Lossy compression

In order to further reduce the size of the index, we apply lossy compression on location and velocity values. We use the same method to perform lossy compression on both location and velocity attributes. The user specifies the maximum allowed error in location and velocity attributes using the parameter σ. σ is defined as the maximum allowed percentage loss in value as the result of the lossy compression.

Algorithm 5: LossyCompress(V, σ, $blockSize$)

Input: V: value of location/velocity, σ: compression error, $blockSize$: number of bits each block
Output: tb: the number of trailing blocks can be removed from V with σ

```
let CV be changed value
let M_IL be maximum value loss of V with σ
let B_V and B_CV be bits string of V and CV, respectively
M_IL = V*σ / 100
convert V into B_V
initialize tb = 1, CV = V
while V − CV < M_IL do
    B_CV = B_V >> tb * blockSize
    B_CV = B_CV << tb * blockSize
    convert B_CV into CV
    increase tb by 1;
return tb
```

Algorithm 5 works as follows. First, the algorithm computes the maximum amount (M_{IL}) by which the compressed value

can differ from the original value. The algorithm iteratively replaces a growing number of the least significant bits with zeros and testes if the resulting number differs from the original value by less than M_{IL}. The algorithm stops when the changed value differs from the original value by more than M_{IL}.

The overall lossy compression algorithm works as follows. First, the algorithm runs the lossless compression algorithms described in Sects. 6.4.2 and 6.4.3 to compress the leading bits. Next, we apply Algorithm 5 to count the number of trailing blocks that can be removed for each value. Next, we remove tb trailing blocks from every value, where tb is the minimum value returned by Algorithm 5 for all values.

Fig. 21 gives an example illustrating how Algorithm 5 works. In this example V = 78.6323 and σ = 0.01%, therefore M_{IL} = 0.0079. The example shows the change in value (represented in both binary and decimal format) through each iteration of the algorithm. The number following the + sign represents the difference between the current changed value and the original value.

0100 0010 1001 1101 0100 0011 1011 1101_2 (78.6323_{10})
Iter1: 0100 0010 1001 1101 0100 0011 1011 0000_2 (78.6322_{10})+0.0001
Iter2: 0100 0010 1001 1101 0100 0011 0000 0000_2 (78.6308_{10})+0.0015
Iter3: 0100 0010 1001 1101 0100 0000 0000 0000_2 (78.6250_{10})+0.0073
Iter4: ~~0100 0010 1001 1101 0000 0000 0000 0000_2 (78.5000_{10})+0.1323~~

Figure 21: Example of compressing value V = 78.6323 with σ = 0.01%

Fig. 22 illustrates an example for the entire compression process for lossy compression of the $x\text{-}location$ column. First, we assume the values in the example have already been adjusted to be relative to each cell's minimum x- and y-coordinates. The example shows that after applying Algorithm 5, there is a number of common trailing zeros. We then remove the longest number of common trailing zero blocks, as illustrated by the box containing the zeros. The example also shows the lossless compression algorithm being applied to the common leading blocks.

0100	0010	1001	1101	0100	0000	0000 0000$_2$	(78.6250_{10})
0100	0010	0101	0101	1101	0000	0000 0000$_2$	(53.4531_{10})
0100	0010	0000	0010	0010	1100	0000 0000$_2$	(32.5430_{10})

Figure 22: Example of applying lossy compressing technique on the $x\text{-}location$ column with σ = 0.01%

Setting the σ value involves a trade-off between compression ratio and precision and recall. A larger σ value results in a better compression ratio but lower precision and recall. Our experiments showed when σ was set to a small value 0.01, we achieved a good compression ratio while still achieving high precision and recall blackredscores (between 0.88 and 1.0). This led us to set the value of σ to 0.01% for our experiments.

6.5. Determining threshold τ for RCG outlier index

Due to the fact that the RCG outlier index is RAM resident, we can no longer use the algorithm described in Sect. 5.2 to compute the velocity threshold τ. We instead set the value of τ based the desired size for the RCG outlier index (s_G) which is taken from the user. The value s_G is set as a percentage of the buffer size. Given s_G, we determine τ as follows. We start with the largest possible τ and decrease it until the size of the RCG outlier index is just below s_G. For a given τ, we determine the resultant RCG outlier index size by first determining the number of outlier objects which have velocity in the y-axis which is greater than τ by looking up the velocity histogram (described in Sect. 5.2). The size of the RCG outlier index, cs is estimated as a function of the number of objects n as follows: $cs = os * n * c$, where os is the normal size of an object which is typically 20 bytes for 32-bit values, and c is the current compression ratio of the current RCG outlier index. The compression ratio is defined as the size of the compressed outlier index divided by the size of the uncompressed outlier index. c is initialized to be 1 and is frequently updated with the current measured compression ratio.

In order to handle the changing velocity distribution as discussed in Sect. 5.6, we also continuously update the velocity histogram and periodically compute an updated τ.

6.6. Main Memory Indexing

Until now we have assumed that the index does not fit in memory. This is because often the server that is holding the index may also be running many different applications with competing memory requirements. In addition, some applications may need to query very large data sets which cannot fully fit in memory. However, in the situation that the entire data set does fit in memory, the restricted memory index proposed in Section 6 can be extended to handle this situation. In this case, we recommend using the grid-based index RCG for both the DVA indexes and the outlier index. This is because grid-based indexes have been found to outperform tree-based indexes for main memory moving object indexing [28, 39, 40, 29]. The compression algorithms developed for the RCG outlier index can be used for the in-memory DVA indexes as well. If main memory is abundant we will not need to use the lossy compression algorithm proposed in Sect. 6.4.4. If memory is unrestricted we can again use the method explained in Sect. 5.2 to set the outlier threshold because we do not need to artificially limit the number of objects that can be placed into the outlier index. A full investigation of an indexing technique under the main memory setting is beyond the scope of this paper and we will leave it for future work.

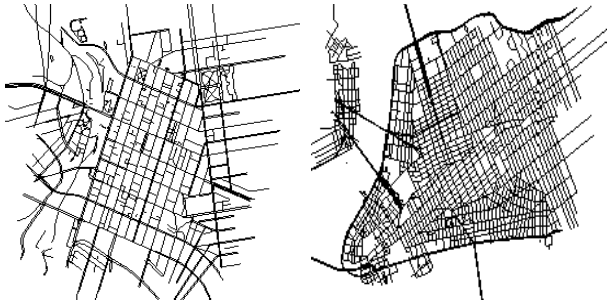
7. Experimental study

In this section, we report the results of experiments illustrating the performance of our VP technique applied to the B^x-tree [2] and the TPR*-tree [6] against their unpartitioned counterparts. We firstly evaluate the ability of our algorithm to find the optimal τ threshold value. Second, we measure the overhead incurred by the velocity analyzer. Third, we compare both the query and update performance of the algorithms across various data sets. Fourth, we compare the range query performance

of the algorithms for various parameter settings including varying data size, maximum speed of objects' movement and query predictive time. Fifth, we show representative results for the rectangular range query. Sixth, we measure the k-NN query performance of the algorithms for varying the number of neighbors k. Finally, we evaluate the effectiveness of our RCG outlier index.

The experiments were conducted based on the benchmark defined in Chen *et al.* [45] for evaluating moving object indexes. The road network and synthetic (uniform) data sets used in the experiments were generated using the benchmark's data generator provided by Chen *et al.* [45]. To generate the road network data sets, we fed the road network nodes and edges into the benchmark generator. The road network nodes and edges were all generated using the XML map data from the OpenStreetMap [46]. We generated four road network data sets. Their characteristics can be summarized as follows:

- The Melbourne CBD (MEL) and New York (NY) road networks (Fig. 23) contain the largest number of nodes and edges, and they have average the length of each edge. Therefore, both road networks have the highest update frequency.
- Both the Chicago (CH) (Fig. 8) and the San Francisco (SA) (Fig. 1(a)) road networks contain a smaller number of nodes and edges and hence both have a smaller number of updates compared to the MEL and the NY networks.
- The CH road network's velocity distribution is the most skewed, followed by the SA, the MEL and the NY road networks.



(a) Melbourne CBD (b) New York CBD

Figure 23: Other tested road networks

We focus our experimental study on the circular time slice range query, with a future predictive time ranging from 20 to 120 seconds (s), as described in Table 2. We focus on the circular query because it resembles many real world occurrences and is also used in the filter step of the k-NN query. The circular range query specifies a range which is a certain distance from a point. For example, a taxi driver is interested in potential passengers within 200 meters (m) of itself, or a tank wants to know if there are any other tanks within one kilometer of itself. We use the circular range query as the default query. We performed the same set of experiments for the rectangular range query and the results are similar to those for the circular range query. We show representative results for the rectangular range query in Sect. 7.8.

Parameter	Setting
Space domain (m^2)	**100,000x100,000**
Cardinality of objects	**100K**, ..., 500K
Max. object speed (m/s)	20, ..., **100**, ..., 200
Max update interval (s)	**120**
Range query radius (m)	100,..., **500**,...,1000
Number of neighbors	10, ...,**50**, ... , 100
Query predictive time (s)	20, ..., **60**, ..., 120
Time duration (s)	**240**
RAM buffer size (pages)	5, 10, ..., **50**, ..., 90
Outlier index size (%)	10, 15, ..., **35**, ..., 50
Disk page size	**4KB**
Secondary storage type	**HDD**, SSD
Data set	**CH**, MEL, SA, NY, uniform

Table 2: Parameters and their settings

The parameters used in the experiments are summarized in Table 2, where values in bold denote the default values used.

We compare our VP technique applied on top of two state-of-the-art moving object indexes of contrasting styles: the B^x-tree [2] and the TPR*-tree [6] with their unpartitioned counterparts (indexes that have not been velocity partitioned). We used the source code for the TPR*-tree and the B^x-tree provided by Chen *et al.* [45]. All code was implemented in C++ under Microsoft Visual C++ 2008 running on Microsoft Windows 7 Professional SP1. The algorithms compared are described as follows:

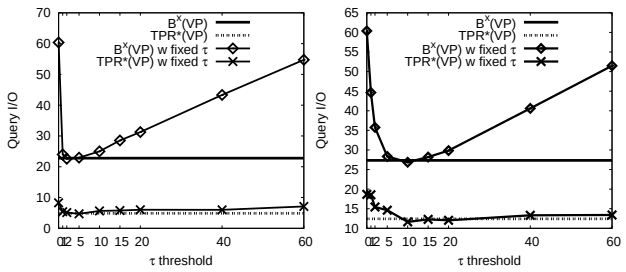
- **B^x-tree**. The B^x-tree [2] has two time buckets and uses the Hilbert curve for space partitioning. We use the improved iterative expanding query algorithm [41] to reduce query enlargement. The histogram used contains 1000x1000 cells.
- **TPR*-tree**. The TPR*-tree [6] is optimized for query size of 1000x1000m^2.
- **B^x(VP)-tree** and **TPR*(VP)-tree**. The VP technique applied to the B^x-tree and the TPR*-tree is denoted as the B^x(VP)-tree and the TPR*(VP)-tree, respectively. We use the disk-based outlier index, where the velocity threshold τ is set using the algorithm described in Sect. 5.1. Both trees use a velocity histogram containing 100 buckets for determining τ value. We set the number of DVA indexes to 2 because we found that in almost all road network data sets, the roads were aligned to two main axes. The settings for the underlying B^x-tree and TPR*-tree are the same as above. The velocity analyzer used for both indexes used 10,000 sample velocity points.
- **B^x(VP:G)-tree** and **TPR*(VP:G)-tree**. The same as the B^x(VP)-tree and the TPR*(VP)-tree except with the RCG outlier index replacing the disk-based outlier index. The

RCG outlier index uses two time buckets. The grid contains 100x100 cells.

Our experiments measure the following metrics: average I/O per query; average I/O per update; average execution time per query; and average execution time per update. The execution time (referred to as time) results include both CPU and I/O time. We use both a magnetic hard disk drive (HDD) and a state-of-the-art solid-state drive (SSD) to measure the I/O time. We use the HDD as the default secondary storage. The SSD is used in Sect. 7.11 because in this section, we test the performance of the RCG outlier index which is designed for the SSD. The update metric results are only reported for one experiment (see Sect. 7.3) because this paper is focused on improving query performance.

All experiments were conducted on a PC powered by Intel Core i7 CPU 2.8GHz with 8GB DDR3 RAM using a 1TB 7200RMS Seagate Barracuda HDD and a 256GB OCZ Vertex 4 SSD.

7.1. Finding optimal τ threshold



(a) CH road network (b) SA road network

Figure 24: τ algorithm versus varying fixed τ threshold

In this experiment, we examine the effectiveness of our algorithm (see Sect. 5.2) at finding the optimal τ threshold for each index. As mentioned before, τ is used to determine which objects should be placed in the outlier index. We compared the B^x(VP)-tree and the TPR*(VP)-tree using different fixed τ thresholds against the B^x(VP)-tree and the TPR*(VP)-tree automatically finding the optimal threshold value according to the algorithm in Sect. 5.2. We used both the CH and SA road network data sets for this experiment. The results are shown in Fig. 24. In Fig. 24, the straight lines represent the B^x(VP)-tree and the TPR*(VP)-tree using the automatic algorithm for determining τ and the curves represent the B^x(VP)-tree and the TPR*(VP)-tree using different fixed τ thresholds. The results show that the VP technique is able to automatically compute a near optimal τ threshold for both real data sets and moving object indexes.

7.2. Velocity analyzer overhead

In this experiment, we measure the overhead of running our velocity analyzer as described in Sects. 5.1 and 5.2. The velocity analyzer partitions the sample velocity points using a combination of PCA and k-means clustering to arrive at the DVA index boundaries. We performed this experiment across the four road networks, CH, SA, MEL, NY and the uniform synthetic data sets. We ran each data set five times and reported the average execution time. The results are shown in Fig. 25. The results show that the overhead of the velocity analyzer over all tested data sets is low, taking between 50 milliseconds and 97 milliseconds.

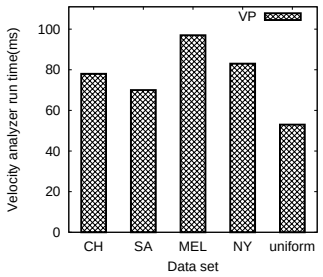


Figure 25: Overhead of velocity analyzer

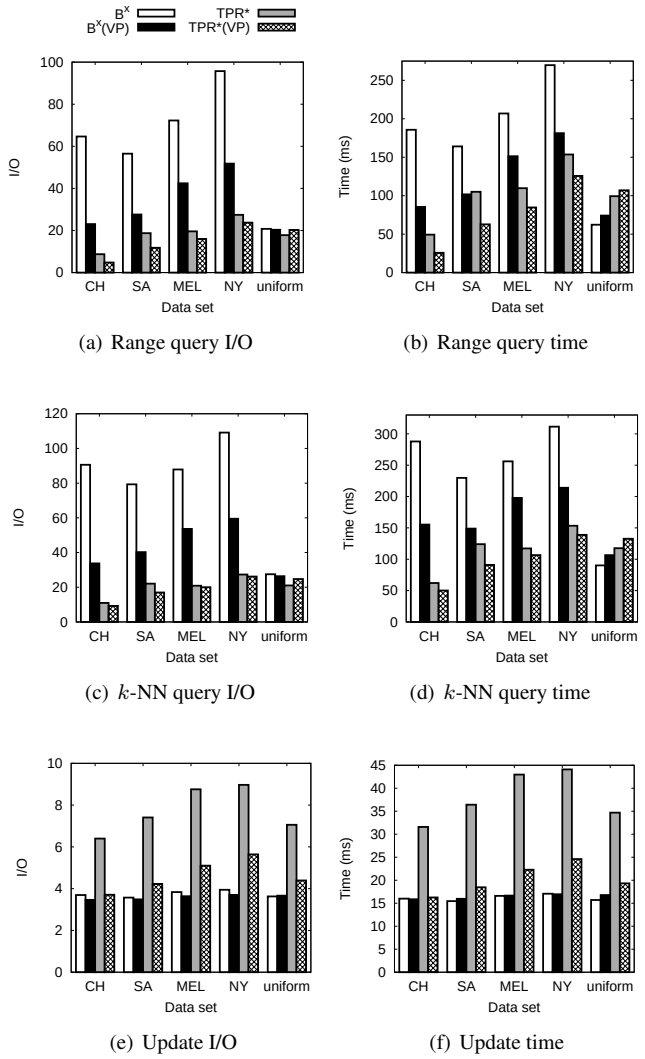
7.3. Effect of varying data sets

In this experiment, we compare the algorithms across the four road networks CH, SA, MEL, NY and the uniform synthetic data sets. The range query I/O and time results are shown in Figs. 26(a) and 26(b), respectively. The k-NN query I/O and time results are shown in Figs 26(c) and 26(d), respectively. The results show that the B^x(VP)-tree and the TPR*(VP)-tree consistently outperform their unpartitioned counterparts for road network data sets by up to a factor of 2.8 for range query I/O, by up to a factor of 2.2 for range query time, by up to a factor of 2.6 for k-NN query I/O and by up to a factor of 1.8 for k-NN query time. Performance improvement is due to the fact that the VP technique is able to exploit the presence of DVAs in these data sets.

In general, the VP technique is able to improve both range and k-NN query performance of the B^x-tree by more than the TPR*-tree because the B^x-tree does not attempt to group objects travelling in similar directions at all. In contrast, the insertion algorithm of the TPR*-tree attempts to group objects travelling in the same direction into the same tree node, albeit in a locally optimized way instead of the globally optimized way of the VP technique. Therefore, for the TPR*-tree, the performance advantage of using the VP technique is diminished.

The results for the uniform data set show that the performance advantage of the B^x(VP)-tree and the TPR*(VP)-tree over their unpartitioned counterparts is removed. This is because in the uniform data set there are no DVAs, and therefore nothing can be gained from partitioning the index by velocity distributions. In some cases, the B^x(VP)-tree performs slightly worse than the unpartitioned counterparts because of the overhead of maintaining multiple indexes and frequently computing an updated τ threshold.

The update I/O and execution time results for this experiment are shown in Figs. 26(e) and 26(f), respectively. The TPR*(VP)-tree outperforms the TPR*-tree by up to a factor of 1.7 for average update I/O cost and by up to a factor of 1.9 for average execution time. This is because both the deletion



(a) Range query I/O (b) Range query time

(c) k-NN query I/O (d) k-NN query time

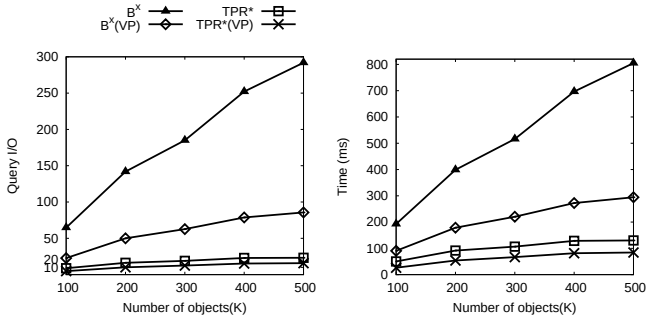
(e) Update I/O (f) Update time

Figure 26: Effect of varying data sets

and insertion algorithms of the TPR*-tree involve traversing the tree in a similar fashion to the query. Our algorithm is better at querying than the unpartitioned TPR*-tree. This fact combined with the fact that each of the partitioned indexes is smaller than the single unpartitioned TPR*-tree, explains the reason for the faster update performance of the TPR*(VP)-tree compared to the unpartitioned TPR*-tree. However, the update performance of the B^x(VP)-tree and the unpartitioned B^x-tree are similar. This is because for the B^x-tree, the update performance is directly proportional to the height of the tree. The height of the B^x(VP)-tree and the unpartitioned B^x-tree are the same in our experiments. In fact, the B^x(VP)-tree is slightly worse than the B^x-tree for update performance due to the fact buffering is more effective when there are less trees and the B^x(VP)-tree needs to frequently compute an updated τ threshold.

For the remaining experiments, we only report query cost results and omit the update results because the technique proposed in this paper is mainly aimed at improving query performance rather than update performance.

7.4. Effect of data size on range query

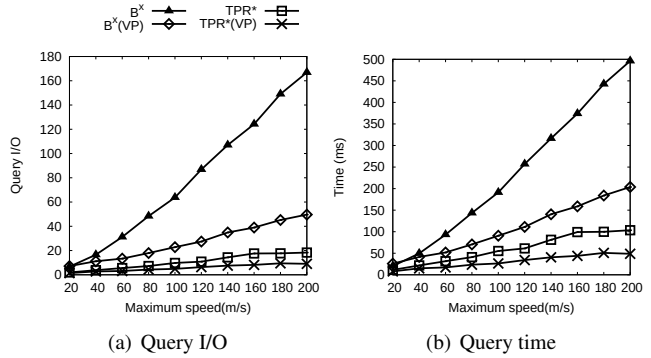


(a) Query I/O (b) Query time

Figure 27: Effect of data size on range query

In this experiment, we examine the query performance of each index while varying the number of objects. As the data size grows, Fig. 27 shows that the query performance increases approximately linearly across all indexes. We observed that the B^x-tree has the worst query performance and scales poorly with increasing number of objects. The results show that the B^x(VP)-tree is effective at improving the performance of the unpartitioned B^x-tree by up to as much as a factor of 3.4 for I/O and a factor of 2.8 for execution time. The performance improvement of the TPR*(VP)-tree over the unpartitioned TPR*-tree is more modest at up to a factor of 1.8 for I/O and 1.9 for execution time. The reason for this is the same as explained in the previous section, namely, the TPR*-tree already attempts to group objects moving in the same direction into the same tree node, whereas the B^x-tree does not.

7.5. Effect of maximum object speed on range query



(a) Query I/O (b) Query time

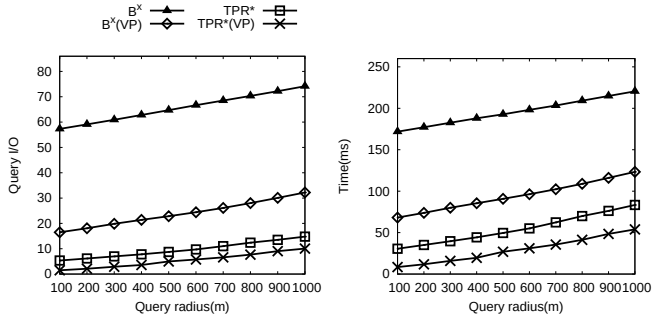
Figure 28: Effect of maximum object speed on range query

In this experiment, we study the effect of varying the maximum object speed on the query performance among all the indexes. Fig. 28 shows that the B^x-tree suffers the most from increasing the maximum object speed and exhibits the steepest increase in both query I/O and query execution time. The reason is that it uses the maximum velocity when expanding queries.

The results show that the VP technique is able to improve the performance of the unpartitioned indexes by an increasing margin as the maximum object speeds increase. This matches the analysis of Sect. 4.

The B^x(VP)-tree outperforms the B^x-tree by up to a factor of 3.4 for average query I/O and by up to a factor of 2.8 for query execution time. The TPR*(VP)-tree outperforms the TPR*-tree by up to a factor of 2 for average query I/O and by up to a factor of 2.1 for query execution time.

7.6. *Effect of range query size on range query*



(a) Query I/O (b) Query time

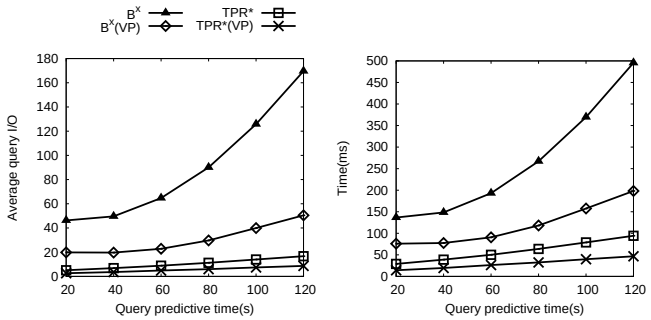
Figure 29: Effect of range query size on range query

In this experiment, we vary the radius of the range query. Results in Fig. 29 again show that the VP technique is more effective at improving the performance of the B^x-tree compared to the TPR*-tree. However, the relative performance difference between the B^x(VP)-tree and the TPR*(VP)-tree and their unpartitioned counterparts become relatively smaller in percentage terms. The reason for this is that as the query window becomes larger, the extent size of the query dominates the query expansion due to the object velocities. The VP technique only reduces query expansion by partitioning the index according to object velocities and does not reduce the query extent size.

More specifically, the results show that for a small query size (radius = 100m) the B^x(VP)-tree outperforms the B^x-tree by up to a factor of 3.5 for query I/O and 2.8 for query execution time and the TPR*(VP)-tree outperforms the TPR*-tree by up to a factor of 3.6 for query I/O and 3.8 for query execution time.

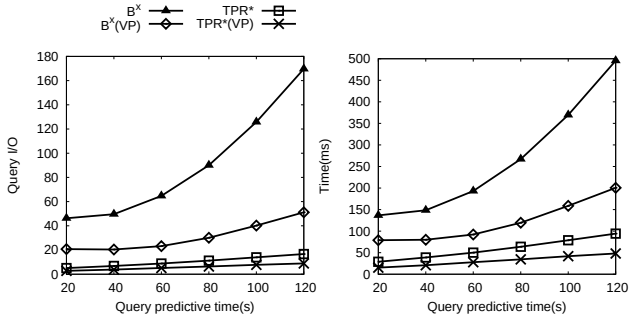
7.7. *Effect of query predictive time on range query*

In this experiment, we vary the query predictive time from 20 to 120 s. This experiment is important since it demonstrates how well we can restrict the expansion of the search space as we query further into the future. The results in Fig. 30 again show that the query performance of the B^x-tree degrades much faster with increasing query predictive time than the other algorithms. Again, the VP technique is able to make a large impact on improving the performance of the B^x-tree compared to the TPR*-tree. The reasons are similar to the previous experiment, namely, the B^x-tree expands the query too much but this time due to a larger time value rather than velocity value.



(a) Query I/O (b) Query time

Figure 30: Effect of query predictive time on range query



(a) Query I/O (b) Query time

Figure 31: Effect of query predictive time on the rectangular range query

7.8. *Effect of query predictive time on rectangular range query*

As mentioned earlier, we conducted the same set of experiments for the rectangular range query as the circular range query and the results were similar. However, due to space limitations, we only show representative results for the rectangular range query. We chose the vary query predictive time experiment because it tests the ability of the algorithms to handle varying rates of query search space expansion.

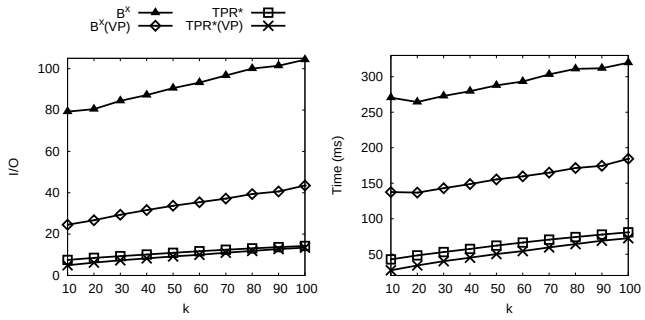
In this experiment, the rectangular range queries have side lengths of 1000x1000m^2. The results shown in Fig. 31 are almost the same as the results for the circular range query.

7.9. *Effect of number of neighbors k on k-NN query*

In this experiment, we examine the k-NN query performance of the algorithms by varying k from 10 to 100. Results in Fig. 32 show that our VP technique applied to the B^x-tree and the TPR*-tree consistently outperform their counterparts. The results indicate that our VP technique can improve the performance of the k-NN query by similar margins compared to the range query results which clearly validates the effectiveness of the VP technique for improving both range and k-NN queries.

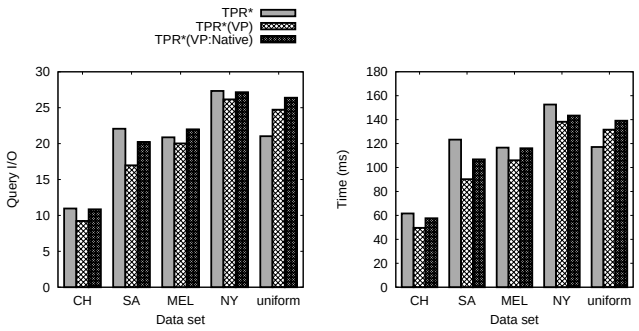
7.10. *Effect of k-NN algorithms when applying VP technique to the TPR*-tree for different data sets*

In this experiment, we compare the performance of our proposed k-NN algorithm with the "native" R-tree best-first traversal when applying the VP technique to the TPR*-tree (denoted



(a) k-NN query I/O (b) k-NN query time

Figure 32: Effect of number of neighbors on k-NN query



(a) k-NN query I/O (b) k-NN query time

Figure 33: Effect of k-NN algorithms when applying VP technique to the TPR*-tree for different data sets

as TPR*(VP:Native)-tree). Specifically, the k-NN query algorithm for TPR*(VP:Native)-tree works as follows. First, we use the native TPR*-tree k-NN query algorithm to query each DVA index. Next, we execute a range query on the oulier index centered at the query point and using r_{max} radius, where r_{max} is the distance to the k^{th} nearest neighbor found in the DVA indexes. The k-NN objects of the whole data set must within r_{max} since within the DVA indexes alone there already exists k objects within r_{max}. We then combine the results from querying the DVA indexes and the outlier index to find the k-NN objects of the entire data set. We just use a simple range query on the outlier index since r_{max} has narrowed the search space a lot and the number of objects in the outlier index is usually quite small.

The result in Fig. 33 shows that actually our original k-NN query algorithm (denoted as TPR*(VP)-tree) slightly outperforms the TPR*(VP:Native)-tree. This is because when the TPR*(VP:Native)-tree searches the multiple DVA indexes it will end up search a larger area than necessary. For example, suppose k is 10 and we have 2 DVA indexes then TPR*(VP:Native)-tree will need to find 10 objects from each DVA index. Whereas, the range based k-NN search of the TPR*(VP)-tree can stop when it finds the range that contains 10 objects within the two DVA indexes altogether. Also as the TPR*(VP)-tree expands the search, the previously visited objects from the previous smaller ranges are normally cached in the RAM buffer and and therefore do not incur any additional I/O. So the TPR*(VP)-tree ends up searching a smaller total search area without incurring repeated I/O.

The proposed native best-first k-NN algorithm of this section uses a straightforward approach of querying each DVA index and the outlier index separately. This leads to larger than necessary search regions. A more sophisticated approach to native k-NN search would use one global queue of nodes for searching across all DVA and outlier indexes simultaneously. This approach is more efficient than our proposed straightforward approach by needing to search over a smaller search area. This sophisticated approach would require the MBRs to be transformed back to the common coordinate system. Implementing and benchmarking this more sophisticated approach is an important area of future work.

7.11. Evaluation of the RCG outlier index

For the remaining experiments, we measure the performance of our VP technique using the RCG RAM resident outlier index (the B^x(VP:G)-tree and the TPR*(VP:G)-tree) compared to using the disk-based outlier index (the B^x(VP)-tree and the TPR*(VP)-tree).

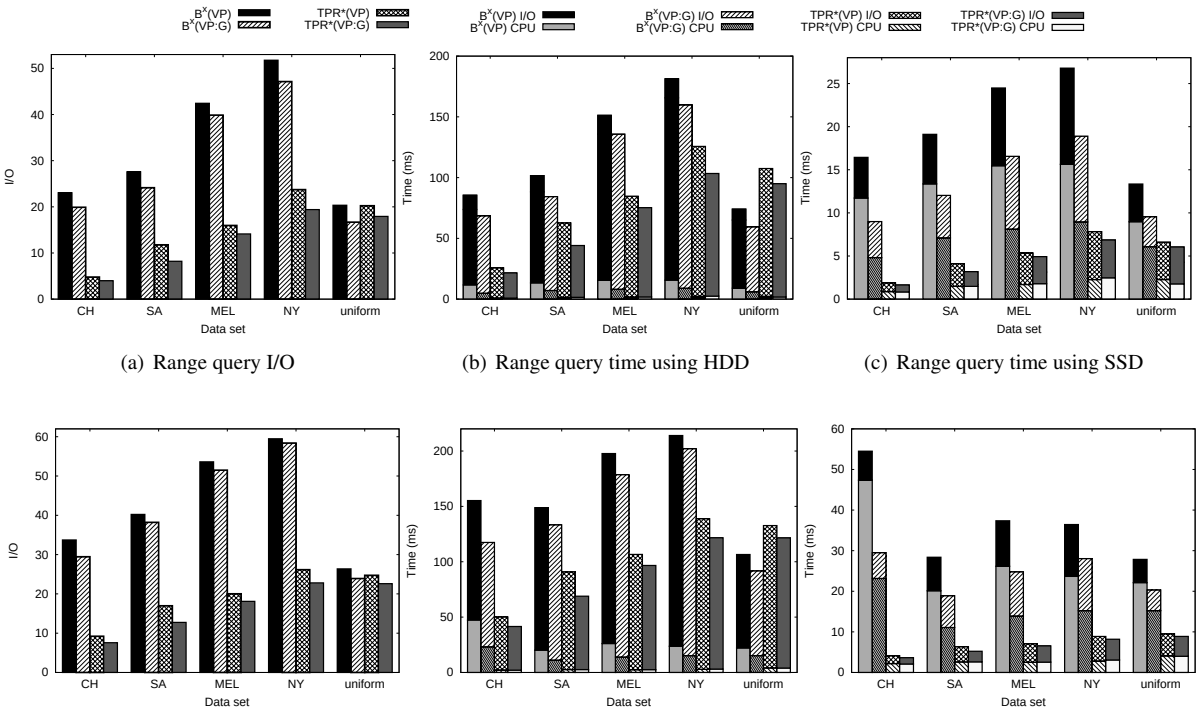
7.11.1. Effect of data sets

In this experiment, we compare the performance of the VP technique using the RCG outlier index versus the disk-based outlier index for different road network data sets. We report the results of using the HDD and the SSD for both the range and k-NN queries. To gain a better understanding of the timing results, we break down the time taken between CPU time and I/O time. The results in Fig. 34 show that using the RCG outlier index consistently outperforms the disk-based index. We notice that the B^x(VP:G)-tree is by up to a factor of 2.4 faster for range query and by up to a factor of 2.0 faster for k-NN query than the B^x(VP)-tree in terms of CPU time. The reason is that the RCG outlier index employed by the B^x(VP:G)-tree uses less CPU time compared to the disk-based outlier index as discussed in Sect. 6. When using the HDD as the secondary storage, the positive effects of using less CPU time is smaller due to the slowness of disk I/O. In contrast, the CPU time spent is more important when using the SSD as secondary storage due to the faster I/O performance of the SSD. The results show that when using the SSD, the B^x(VP:G)-tree outperforms the B^x(VP)-tree by up to a factor of 2.0 for the range query time and by up to a factor of 1.8 for the k-NN query time.

We observe that the performance improvement of the TPR*-tree using the RCG outlier index is much less than the improvement for the B^x-tree. The reason is that the TPR*-tree is already very CPU efficient by grouping objects to reduce the search space. We recommend the use of the disk-based outlier index for the TPR*-tree because the RCG outlier index does not offer significant improvements in performance and can also generate possible errors due to the use of lossy compression.

7.11.2. Effect of RAM buffer size

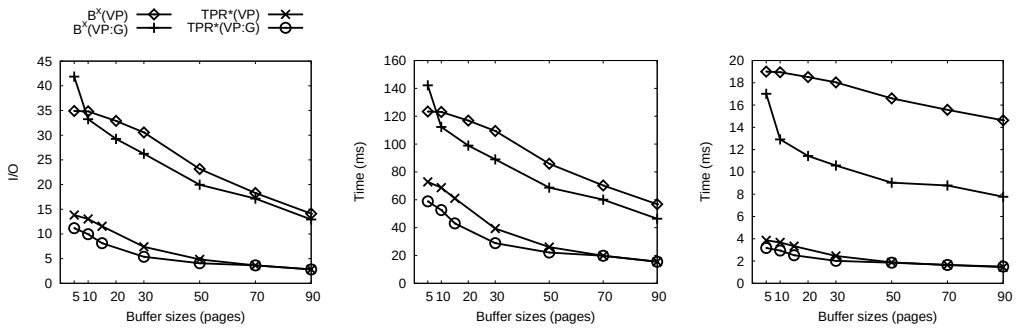
In this experiment, we study the effect of RAM buffer size on the range query performance by varying the RAM buffer



(a) Range query I/O (b) Range query time using HDD (c) Range query time using SSD

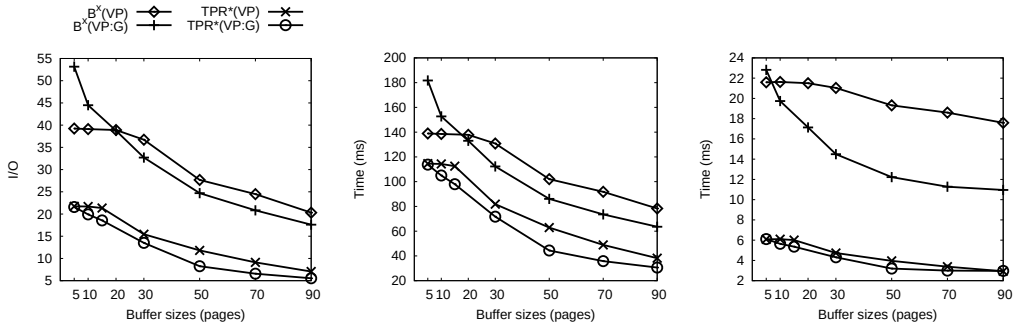
(d) k-NN query I/O (e) k-NN query time using HDD (f) k-NN query time using SSD

Figure 34: Effect of data sets



(a) Range query I/O (b) Range query time using HDD (c) Range query time using SSD

Figure 35: Effect of RAM buffer size for CH data set



(a) Range query I/O (b) Range query time using HDD (c) Range query time using SSD

Figure 36: Effect of RAM buffer size for SA data set

size from 5 to 90 pages. Figs. 35 and 36 show the range query performance in terms of I/O, time using HDD and time using SSD for the CH and SA road network data sets, respectively.

For the CH road network data set, where the velocity distribution is most skewed i.e, there is very limited number of outlier objects in the data set, the results in Fig. 35 show that the indexes which employ the RCG outlier index consistently outperform other indexes which employ the disk-based outlier index except when the buffer size is 5 pages. When the buffer size is 5 pages, the range query I/O of the B^x(VP:G)-tree is worse than the query I/O of the B^x(VP)-tree. The reason is that at very small buffer size, the RCG outlier index could not accommodate many outlier objects, therefore some outlier objects are inserted into the DVA indexes, affecting the query performance of the DVA indexes. When using the SSD as the secondary storage, the CPU time saving of the B^x(VP:G)-tree made from using the grid-based RCG outlier index outweighs the extra I/O time spent on the DVA indexes for the entire range of RAM sizes tested.

For the SA road network data set where the velocity distribution is less skewed, i.e, there are more outlier objects in the data sets, the results in Fig. 36 show that it takes a larger buffer size (30 pages for the HDD and 10 pages for the SSD) before the B^x(VP:G)-tree outperforms the B^x(VP)-tree. This is because for the less skewed data set, the number of outlier objects is greater. However, even for this less skewed data set, it still only takes a small buffer size of 10 pages before the RCG outlier index outperforms the disk-based outlier index on the SSD.

Similar to the observations given in Sect. 7.11.1, the query performance of the TPR*(VP:G)-tree only sightly outperforms the TPR*(VP)-tree when using the HDD and is almost the same when using the SSD. The reasons for this are the same as for Sect. 7.11.1.

7.11.3. *Effect of compression error σ*

In this experiment, we examine the effect of choosing the maximum compression error σ on the compression ratio of the RCG outlier index and the performance of the range query. We define the compression ratio as the size of the compressed outlier index divided by the size of the uncompressed outlier index.

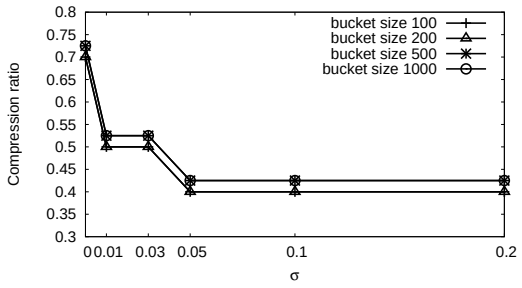
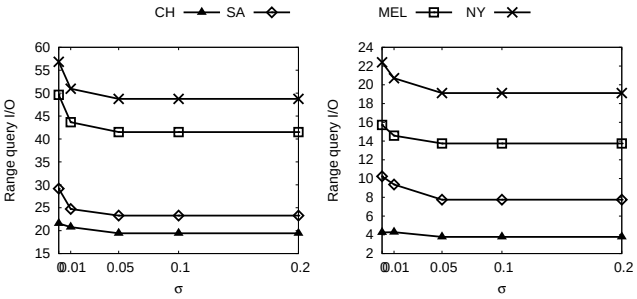


Figure 37: Effect of compression error σ on compression ratio

Fig. 37 shows an experiment which applies the compression technique on a sample of the CH road network data set where we vary the number of objects in one bucket and also vary the maximum allowed error σ. The experiment shows that even at a very small σ of 0.01%, we can still achieve a good compression ratio of 0.5 compared to 0.7 for lossless compression. This is because setting σ to 0.01% already allows us to remove many insignificant bits. The compression ratio between σ = 0.05% and 0.2% is constant because further reducing the compression ratio after σ = 0.05% would require removing very significant bits which would mean a very large σ is required.

Next, we report the results of the range query I/O of the B^x(VP:G)-tree and the TPR*(VP:G)-tree while varying σ from 0 to 2.0. The value of σ = 0 means that the compression is lossless. We found the k-NN query and range query results are very similar and therefore only report the results for the range query. The results in Fig. 38 show that the lossy compression using even a small σ = 0.01% is able to significantly improve query performance compared to the lossless compression. The reason the performance does not change between σ of 0.05% and 0.2% is for similar reasons as the compression ratio results.



(a) B^x(VP:G)-tree (b) TPR*(VP:G)-tree

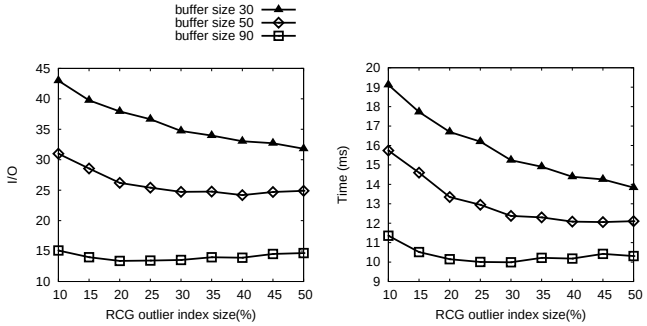
Figure 38: Effect of compression error σ on range query

7.11.4. *Determining the size of RCG outlier index*

In order to determine a good size for the RCG outlier index for a given workload, we run some sample tests with different sized RCG outlier indexes in terms of percentage of the total RAM buffer size. Fig. 39 shows the range query performance of the B^x(VP:G)-tree for the SA road network data set, when the RAM buffer size is varied from 30 to 90 pages (4KB per page). For the default buffer size of 50 pages, we found a RCG outlier index size of 35% of the RAM buffer size gives us the best overall performance. For the CH, MEL, and NY data sets, the good size ranges from 30% to 35%, respectively. Therefore, we have chosen 35% as the default parameter for all data sets.

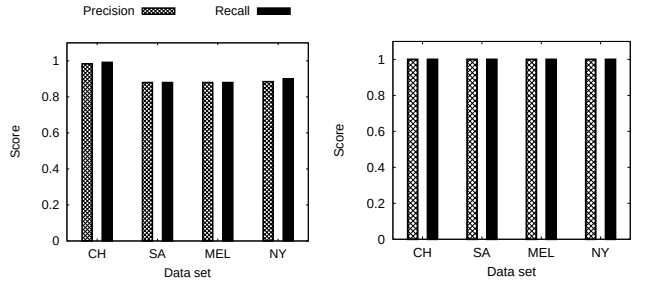
7.11.5. *Effect of data sets on precision and recall*

Finally, we use *precision* and *recall* [47] to measure the correctness of the query result when lossy compression with σ = 0.01% is used for the RCG outlier index. In our context, precision is the fraction of returned objects in the result that actually satisfy the query predicate, whereas recall is the fraction of the returned objects that satisfy the query predicate that are in the query result. Ideally, precision = recall = 1, meaning that the query result returns exactly the objects that satisfy the query.



(a) Query I/O (b) Range query time

Figure 39: Effect of RCG outlier index size on range query of the B^x(VP:G)-tree for the SA road network data set



(a) Range query (b) k-NN query

Figure 40: Effect of data sets on precision and recall for range and k-NN queries

We only report the results of the B^x(VP:G)-tree, because the results for the TPR*(VP:G)-tree are exactly the same. Fig. 40 validates that the lossy compression technique achieves high scores between 0.88 and 1.0 for precision and recall of range queries and perfect precision and recall for k-NN queries over various road network data sets.

8. Conclusion

In this paper, we proposed the VP technique, a novel method that improves the moving object index performance by exploiting velocity distribution skew. The main idea is to partition objects based on their moving directions, and then use separate DVA indexes to index the objects moving along different dominant velocity axes separately. We first provided analysis to show why this idea should work. Then, we proposed several algorithms to achieve effective velocity partitioning. The VP technique can be applied to most moving object index structures. Finally, we implemented it on two representative index structures, the TPR*-tree and the B^x-tree and performed extensive experiments on both real and synthetic data sets. The results showed that these index structures, equipped with the VP technique, outperform their original versions consistently.

We found that, when the SSD was used, the outlier index accounted for a disproportionately high percentage of the total time. We addressed this by replacing the disk-based outlier index with the novel RCG outlier index. For the B^x-tree on the SSD, the results showed that the RCG outlier index produces significantly higher overall performance compared to the outlier index designed for the HDD. However, for the TPR*-tree, we recommend the use of the original disk-based outlier index due to the fact that the RCG outlier index only gives small performance improvements for the TPR*-tree but may introduce small errors due to the use of lossy compression.

For future work, we plan to investigate boosting the performance of other moving object indexes using the VP technique.

Acknowledgments This work is supported under the Australian Research Council's Discovery funding scheme (project numbers DP0985451 and DP0880250).

References

[1] J. Dittrich, L. Blunschi, M. Antonio, V. Salles, Indexing moving objects using short-lived throwaway indexes, in: SSTD, 2009, pp. 189–207.
[2] C. S. Jensen, D. Lin, B. C. Ooi, Query and update efficient B$^+$-tree based indexing of moving objects, in: VLDB, 2004, pp. 768–779.
[3] G. Kollios, D. Papadopoulos, D. Gunopulos, J. Tsotras, Indexing mobile objects using dual transformations, VLDB J. 14 (2) (2005) 238–256.
[4] J. M. Patel, Y. Chen, V. P. Chakka, STRIPES: an efficient index for predicted trajectories, in: ACM SIGMOD, 2004, pp. 635–646.
[5] S. Saltenis, C. Jensen, S. Leutenegger, M. Lopez, Indexing the positions of continuously moving objects, in: ACM SIGMOD, 2000, pp. 331–342.
[6] Y. Tao, D. Papadias, J. Sun, The TPR*-tree: an optimized spatio-temporal access method for predictive queries, in: VLDB, 2003, pp. 790–801.
[7] M. Yiu, Y. Tao, N. Mamoulis, The B^{dual}-tree: Indexing moving objects by space filling curves in the dual space, VLDB J. 17 (3) (2008) 379–400.
[8] S. Chen, B. C. Ooi, K.-L. Tan, M. A. Nascimento, ST^{2}B-tree: A self-tunable spatio-temporal B$^+$-tree index for moving objects, in: ACM SIGMOD, 2008, pp. 29–42.
[9] Y. Tao, C. Faloutsos, D. Papadias, B. Liu, Prediction and indexing of moving objects with unknown motion patterns, in: ACM SIGMOD, 2004, pp. 611–622.
[10] K. Tzoumas, M. L. Yiu, C. S. Jensen, Workload-aware indexing of continuously moving objects, PVLDB 2 (1) (2009) 1186–1197.
[11] T. Nguyen, Z. He, R. Zhang, P. Ward, Boosting moving object indexing through velocity partitioning, PVLDB 5 (9) (2012) 860–871.
[12] S. Nutanong, R. Zhang, E. Tanin, L. Kulik, The V*-diagram: A query dependent approach to moving kNN queries, PVLDB 1 (1) (2008) 1095–1106.
[13] R. Zhang, H. V. Jagadish, B. T. Dai, K. Ramamohanarao, Optimized algorithms for predictive range and kNN queries on moving objects, Inf. Syst. 35 (8) (2010) 911–932.
[14] R. Zhang, J. Qi, D. Lin, W. Wang, R. C.-W. Wong, A highly optimized algorithm for continuous intersection join queries over moving objects, VLDB J. 21 (4) (2012) 561–586.
[15] K. Chakrabarti, S. Mehrotra, Local dimensionality reduction: a new approach to indexing high dimensional spaces, in: VLDB, 2000, pp. 89–100.
[16] J. Hui, B. Ooi, H. Shen, C. Yu, An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing, in: ICDE, 2003, pp. 87–98.
[17] I. Jolliffe, Principal Component Analysis, Springer-Verlag, New York, 1986.
[18] J. B. MacQueen, Some methods for classification and analysis of multivariate observations, in: Berkeley Symp. on Math. Statist. and Prob., 1967, pp. 281–297.
[19] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: ACM SIGMOD, 1990, pp. 322–331.
[20] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: ACM SIGMOD, 1984, pp. 57–47.
[21] P. Agarwal, L. Arge, J. Erickson, Indexing moving points, in: PODS, 2000, pp. 175–186.

[22] D. Kollios, G.and Gunopulos, V. Tsotras, On indexing mobile objects, in: PODS, 1999, pp. 261–272.
[23] R. Zhang, B. C. Ooi, K.-L. Tan, Making the pyramid technique robust to query types and workloads, in: ICDE, 2004, pp. 313–324.
[24] V. Almeida, Indexing the trajectories of moving objects in networks, Geoinformatica 9 (1) (2005) 33–60.
[25] J. Chen, X. Meng, Update-efficient indexing of moving objects in road networks, Geoinformatica 13 (4) (2009) 397–424.
[26] E. Frentzos, Indexing objects moving on fixed networks, in: SSTD, 2003, pp. 289–305.
[27] R. H. Güting, V. T. de Almeida, Z. Ding, Modeling and querying moving objects in networks, VLDB J. 15 (2) (2006) 165–190.
[28] H. D. Chon, D. Agrawal, A. El Abbadi, Range and kNN query processing for moving objects in grid model, Mob. Netw. Appl. 8 (4) (2003) 401–412.
[29] X. Yu, K. Q. Pu, N. Koudas, Monitoring k-nearest neighbor queries over moving objects, in: ICDE, 2005, pp. 631–642.
[30] M. E. Ali, E. Tanin, R. Zhang, L. Kulik, A motion-aware approach for efficient evaluation of continuous queries on 3d object databases, VLDB J. 19 (5) (2010) 603–632.
[31] M. E. Ali, R. Zhang, E. Tanin, L. Kulik, A motion-aware approach to continuous retrieval of 3d objects, in: ICDE, 2008, pp. 843–852.
[32] R. Benetis, S. Jensen, G. Karciauskas, S. Saltenis, Nearest and reverse nearest neighbor queries for moving objects, VLDB J 15 (3) (2006) 229–249.
[33] T. Xia, D. Zhang, Continuous reverse nearest neighbor monitoring, in: ICDE, 2006, pp. 77–88.
[34] A. M. Aly, W. G. Aref, M. Ouzzani, Spatial queries with two knn predicates, PVLDB 5 (11) (2012) 1100–1111.
[35] Y. Gao, B. Zheng, Continuous obstructed nearest neighbor queries in spatial databases, in: ACM SIGMOD, 2009, pp. 577–590.
[36] Y. Wang, R. Zhang, C. Xu, J. Qi, Y. Gu, G. Yu, Continuous visible k nearest neighbor query on moving objects, Information Systems 44 (2014) 1–21.
[37] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, Y. Wang, Processing moving kNN queries using influential neighbor sets, PVLDB 8 (2) 113–124.
[38] P. G. D. Ward, Z. He, R. Zhang, J. Qi, Continuous intersection joins over large sets of moving objects using graphic processing units, VLDB J. 23 (6) (2014) 965–985.
[39] K. Mouratidis, D. Papadias, M. Hadjieleftheriou, Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring, in: ACM SIGMOD, 2005, pp. 634–645.
[40] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, D. Šaulys, Trees or grids?: indexing moving objects in main memory, in: ACM SIGSPATIAL GIS, 2009, pp. 236–245.
[41] C. S. Jensen, D. Tiesyte, N. Tradisauskas, Robust B$^+$-tree-based indexing of moving objects, in: MDM, 2006, pp. 12–20.
[42] D. Abadi, S. Madden, M. Ferreira, Integrating compression and execution in column-oriented database systems, in: ACM SIGMOD, 2006, pp. 671–682.
[43] D. J. Abadi, S. R. Madden, N. Hachem, Column-stores vs. row-stores: how different are they really?, in: ACM SIGMOD, 2008, pp. 967–980.
[44] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, C-store: a column-oriented DBMS, in: VLDB, 2005, pp. 553–564.
[45] S. Chen, C. S. Jensen, D. Lin, A benchmark for evaluating moving object indexes, PVLDB 1 (2) (2008) 1574–1585.
[46] OpenStreetMap [cited 2011]. [link].
URL openstreetmap.org
[47] C. D. Manning, P. Raghavan, H. Schtze, Introduction to Information Retrieval, Cambridge University Press, New York, NY, USA, 2008.