Continuous Medoid Queries over Moving Objects

Stavros Papadopoulos [1], Dimitris Sacharidis [2], and Kyriakos Mouratidis [3]

[1] Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
stavros@cse.ust.hk

[2] School of Electrical and Computer Engineering
National Technical University of Athens
Greece, 15780
dsachar@dblab.ntua.gr

[3] School of Information Systems
Singapore Management University
Singapore, 178902
kyriakos@smu.edu.sg

Abstract. In the *k*-medoid problem, given a dataset *P*, we are asked to choose *k* points in *P* as the *medoids*. The optimal medoid set minimizes the average Euclidean distance between the points in *P* and their closest medoid. Finding the optimal *k* medoids is NP hard, and existing algorithms aim at approximate answers, i.e., they compute medoids that achieve a small, yet not minimal, average distance. Similarly in this paper, we also aim at approximate solutions. We consider, however, the continuous version of the problem, where the points in *P* move and our task is to maintain the medoid set on-the-fly (trying to keep the average distance small). To the best of our knowledge, this work constitutes the first attempt on continuous medoid queries. First, we consider *centralized* monitoring, where the points issue location updates whenever they move. A server processes the stream of generated updates and constantly reports the current medoid set. Next, we address *distributed* monitoring, where we assume that the data points have some computational capabilities, and they take over part of the monitoring task. In particular, the server installs adaptive filters (i.e., permissible spatial ranges, called *safe regions*) to the points, which report their location only when they move outside their filters. The distributed techniques reduce the frequency of location updates (and, thus, the network overhead and the server load), at the cost of a slightly higher average distance, compared to the centralized methods. Both our centralized and distributed methods do not make any assumption about the data moving patterns (e.g., velocity vectors, trajectories, etc) and can be applied to an arbitrary number of medoids *k*. We demonstrate the efficiency and efficacy of our techniques through extensive experiments.

Keywords: Medoid Queries, Continuous Query Processing, Moving Object Databases.

1 Introduction

Given a dataset P and a user-specified parameter k, a k-medoid query returns a subset of P consisting of k points. These points are called the *medoids* and are selected so that the average distance between the points in P and their closest medoid is minimized. The k-medoid problem arises in many fields and application domains, including resource allocation, data mining, spatial decision making, etc. Consider the example in Figure 1.1, where $P = \{p_1, ..., p_{24}\}$ is the set of residential blocks in a city, and fire stations are to be opened at three of them. To achieve the shortest average response time to emergency calls, we should minimize the average distance between residential blocks and their closest station. In this case, the best blocks to open fire stations at are the $k = 3$ medoids of P. In our example, the medoids are blocks p_6, p_{15} and p_{22}, shown in grey. The lines in the figure signify the assignment of the residential blocks to their responsible (i.e., closest) fire station. Due to this implicit assignment, k-medoids have also been used in different contexts for partitioning clustering.

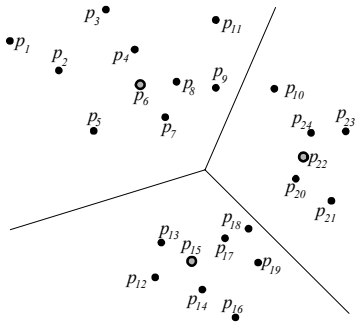


Figure 1.1: A 3-medoid example

Computing an optimal medoid set is NP hard [GJ79], and only approximate answers are possible even for relatively small input datasets. To this end, existing methods range from theoretical approximation schemes (e.g., [ARR98]), to hill-climbing approaches for moderate size datasets (e.g., [KR90, NH94]), to heuristic-based algorithms for disk-resident data (e.g., [EKX95a, EKX95b, MPP]). All previous methods assume a static P, i.e., they compute the k medoids once and then terminate. In this paper, we address a dynamic version of the problem, where the points in P send frequent location updates and the medoid set needs to be continuously maintained. In accordance with most real-world scenarios, the points in P move arbitrarily, with unknown motion patterns. We term the problem *continuous medoid monitoring*.

As a medoid monitoring example, consider a number of users accessing a location based service through their mobile devices, e.g., cellular phones or PDAs. To reduce the communication cost (and, thus, energy consumption), a number k of supernodes

are selected among the mobile devices; the supernodes collect, aggregate and forward to the location server messages received from their vicinity. Due to signal attenuation for long distances, the devices should be close to some supernode. In other words, the supernode selection essentially reduces to a *k*-medoid computation over the set of devices. Additionally, the mobile nature of the system requires on-the-fly medoid maintenance. All the devices (supernodes or not) move frequently and arbitrarily, necessitating supernode re-assignment in order to retain the quality of service.

We consider two system models, corresponding to different mobile environments. First, we address *centralized* medoid monitoring. In this setting, the data objects[1] in *P* send updates to a central server whenever they move. The server processes the location updates and computes/reports the new medoid set. We propose two incremental monitoring algorithms that aim at minimizing the processing time for medoid maintenance. In the centralized model, the objects issue frequent location updates. This raises the additional concern about the communication cost. In particular, in many mobile computing applications, the objects have scarce power resources and we wish to preserve battery life by limiting the number of messages transmitted to the server. This motivates our second, *distributed* processing model. In this context, the server assigns *safe regions* to the data objects, which issue location updates only if they move outside their region. We design effective safe region computation strategies and incorporate them to our medoid monitoring framework. We demonstrate that the distributed methods drastically reduce the object communication overhead, while sacrificing minimal medoid quality (i.e., they result in marginally higher average distance compared to their centralized counterparts).

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our two centralized methods, while Section 4 presents their distributed versions. Section 5 experimentally evaluates the performance of our algorithms. Finally, Section 6 concludes the paper.

2 Related Work

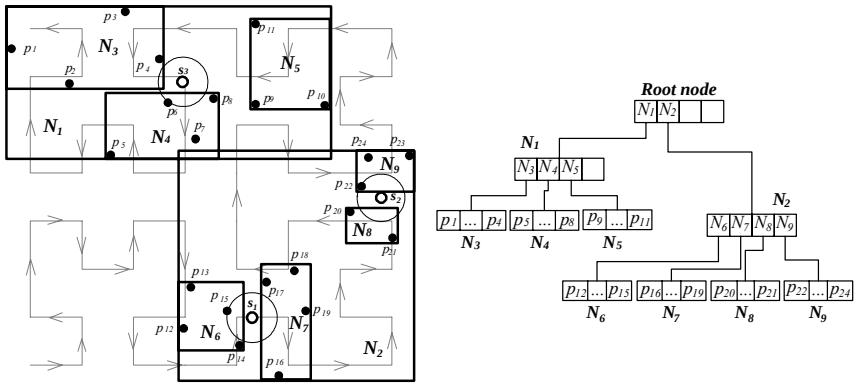
In this section, we survey previous work on medoid queries (in Section 2.1), focusing on solutions targeted at large datasets. We also review spatial query monitoring techniques (in Section 2.2), since we assume a similar system architecture and use related geometric techniques and indexes.

2.1 Medoid Queries

Finding the *k*-medoids is a classic problem in Computational Geometry, where it is usually referred to as the *k*-medians problem. Since it is NP hard, several approximation schemes have been proposed for its solution (e.g., [ARR98]). These schemes are of theoretical nature, aiming at graceful asymptotic bounds. More practical solutions include hill-climbing algorithms, such as PAM and CLARA

[1] Henceforth, the terms point and object are used interchangeably.

[KR90]. Starting with a randomly chosen k-medoid set, these methods consider swapping one medoid with another, randomly chosen data point. If the swap leads to a lower average distance, then the resulting medoid set becomes the new candidate answer. This procedure is repeated for a fixed number of possible swaps. It terminates when no considered swap achieves a lower distance than the current medoid set, and returns the latter as the solution. To achieve better scalability than PAM and CLARA, Ng and Han [NH94] propose CLARANS. It builds upon CLARA, examining however a smaller set of possible swaps, and, thus, speeding up the execution (i.e., converging faster to a local minimum). CLARANS is still slow for large problem instances (being restricted to inputs of just a few thousand objects), and it is impractical for disk-resident data. Motivated by this fact, Ester et al. [EKX95a, EKX95b] design FOR. In FOR, dataset P is indexed with an R-tree [G84, BKSS90], and a sample is formed by drawing one data point from each leaf of the R-tree. FOR executes CLARANS on this sample and returns the computed medoids. Focused also on disk-resident data, Mouratidis et al. [MPP] propose TPAQ, a method that solves k-medoid and related problems. TPAQ assumes that P is indexed with an R-tree and exploits its grouping properties to avoid reading the entire dataset, while achieving a low average distance. To exemplify, consider dataset $P = \{p_1, ..., p_{24}\}$ in Figure 2.1a and its R-tree in Figure 2.1b.



(a) Points and node extents (b) The R-tree structure

Figure 2.1: Example R-tree and TPAQ execution for 3-medoid computation

Assume that TPAQ is posed with a 3-medoid query. It descends the R-tree from the root down to the topmost level that contains more than (or equal to) k entries. This level is called the *partitioning level*, and let E denote the set of its entries. In Figure 2.1, the partitioning level is the second one, and its entries are $E = \{N_3, ..., N_9\}$. The entries in E are sorted according to their center's Hilbert value, and the resulting sorted list is divided into k groups S_i of equal cardinality (i.e., $|E|/k$ entries each). The sorted list in our example (as given by the Hilbert curve shown in Figure 2.1a) is N_6, N_7, N_8, N_9, N_5, N_4, N_3, and the 3 groups are $S_1 = \{N_6, N_7\}$, $S_2 = \{N_8, N_9\}$, and $S_3 = \{N_5,$

N_4, N_3}. For each S_i, TPAQ computes the geometric centroid[2] s_i and performs a nearest neighbor (NN) search at s_i among the underlying data points (i.e., among the points corresponding to the sub-trees of entries in S_i). In Figure 2.1a, the centroids of the three groups are points s_1, s_2, and s_3 (appearing hollow). The three medoids returned are their NNs, i.e., points p_{15}, p_{22}, and p_6. TPAQ is shown to achieve lower distance than FOR and exhibits better scalability.

The existing *k*-medoid algorithms are unsuitable for our continuous monitoring setting. All aforementioned methods are designed for static datasets and snapshot queries (i.e., they compute the medoids once and then terminate); their extension to incremental medoid maintenance (in the presence of updates) is non-trivial, if possible at all. On the other hand, the naïve approach of re-computing from scratch the medoids (with some existing algorithm) in each update processing cycle is prohibitively expensive in a highly dynamic scenario, failing to reuse previous results. Additional problems of existing methods are: (i) the hill-climbing approaches (PAM, CLARA, CLARANS, etc.) are very slow for moderate or large input sizes, while (ii) TPAQ and FOR are designed for disk-resident data, with primary objective the minimization of the I/O cost; disk accesses are not an issue in our main memory setting, where CPU time (and communication cost, in the distributed case) is the only concern. On the other hand, an important finding of previous work to our problem is the efficiency and, more so, the efficacy of TPAQ, which motivates us to use a similar Hilbert-based (or, in general, space filling curve-based) approach for our purposes.

Regarding medoid-related problems in dynamic settings, Guha et al. [GMM+03] solve the *k*-medoid problem in a streaming environment. In the assumed model, the points of the input dataset *P* stream into the system. The main memory is not enough to store entire *P*, so the streamed data points are processed once and then discarded as new ones arrive. When the entire input set is seen, the system reports its *k*-medoids. [GMM+03] proposes an one-pass *k*-medoid algorithm that solves the above problem, using a small amount of space. Even though this is a dynamic method, it does not apply to our setting; in our case, (i) the memory does fit the *entire* dataset, but the points therein receive *location updates* in an on-line fashion, and (ii) the system needs to continuously report the *k*-medoid set *at any time*.

A problem related to *k*-medoids is *min-dist optimal-location* (MDOL) computation. The input consists of a set of data points *P*, a set of existing facilities (i.e., a set of existing medoids) and a user-specified spatial region *R*, wherein a new facility should open. The output of an MDOL query is the location in *R* where the new facility should be built in order to minimize the overall average distance between the data points and their closest facility. Zhang et al. [ZDXT06] propose an exact method for this problem. The main differences from the *k*-medoid problem is that (i) MDOL assumes that a set of facilities already exists, (ii) it computes a single point (as opposed to *k*), and (iii) the returned point does not necessarily belong to *P*, but it can be anywhere inside region *R*.

The *k*-medoid problem is related to clustering; essentially, given the medoids, the input dataset can be partitioned into *k* clusters by assigning each point to its closest medoid. The other direction, however, does not work; although there are numerous

[2] The geometric centroid of group S_i is point s_i with coordinates $s_i.x$ and $s_i.y$ equal to the average x- and y- coordinates, respectively, of the entry centers in S_i.

clustering methods for large input sets (e.g., DBSCAN [EKSX96], BIRCH [ZRL96], CURE [GRS98] and OPTICS [ABKS99]), their objective is to create clusters such that the points in any cluster are more similar to each other than to points in other clusters. In addition to addressing a problem of different nature, most clustering algorithms are computationally intensive and unsuitable for the highly dynamic environments we tackle in this work.

2.2 Continuous Spatial Queries

The first spatial monitoring techniques were targeted at range queries, where the data objects send location updates to a central server, and the latter continuously reports the objects that fall in each monitored range. *Q-index* [PXK+02] processes static range queries. It indexes the ranges using an R-tree and probes moving objects against the index in order to determine the affected queries and update their results. SINA [MXA04] monitors (potentially moving) range queries using a three-step spatial join between moving objects and ranges. *Mobieyes* [GL04] and MQM [CHC04] follow a distributed processing approach, where the objects utilize their computational capabilities and suppress some location updates. In particular, all of *Q-index*, *Mobieyes* and MQM utilize the concept of *safe regions*, according to which each object p is assigned a circular or rectangular region, such that p needs to issue an update only if it exits this area (because, otherwise, it does not influence the result of any query). Figure 2.2 shows a range monitoring example, where the current result of query Q_1 is object p_1, of Q_2 is object p_2, while no object qualifies queries Q_3, Q_4, Q_5. The safe regions for p_1 and p_4 are circular, while for p_2 and p_3 they are rectangular, as shown in the figure (the safe rectangle for p_2 coincides with the boundary of Q_2). Note that even if the objects move, unless they fall outside their assigned safe regions, no query result can change.

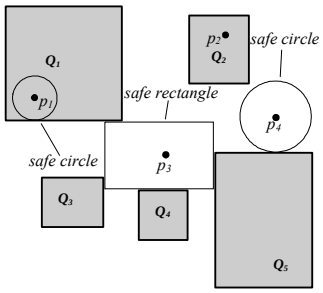


Figure 2.2: The safe regions concept

In addition to rage queries, several methods have been recently proposed for k Nearest Neighbor (k-NN) monitoring. Koudas et al. [KOTZ04] present a system for approximate k-NN queries over streams of multidimensional points. Yu et al. [YPK05], Xiong et al. [XMA05] and Mouratidis et al. [MHP05] describe algorithms

for exact k-NN queries; all three methods index the data with a regular grid and maintain the k-NN results by considering only object movements that may influence some query. The aforementioned techniques aim at low processing time. There exist, however, methods designed for network cost minimization [MPBT05, HXL05] by exploitation of the objects' computational resources; their rationale is similar to that of the *safe regions* explained in Figure 2.2.

3 Centralized Medoid Monitoring

In this section we present our centralized methods. We assume that dataset P consists of $|P|$ two-dimensional points. Although our methods are applicable to higher dimensions, in accordance with most real-world mobile environments, we focus on two dimensions. Furthermore, for ease of presentation, we consider a unit dataspace, i.e., all data fall in $[0,1]^2$. Every point p in P is a tuple of the form $<p.id, p.x, p.y>$, where $p.id$ is a unique identifier and $(p.x, p.y)$ are p's coordinates. Whenever p moves, it issues an update to the monitoring server; the update has the form $<p.id, p.x_{old}, p.y_{old}, p.x_{new}, p.y_{new}>$[3], implying that p moves from $(p.x_{old}, p.y_{old})$ to $(p.x_{new}, p.y_{new})$. The objects move frequently and arbitrarily.

We present two centralized medoid monitoring algorithms, based on a common intuition exemplified in Figure 3.1. Dataset P contains two clusters C_1 and C_2. Suppose that a 2-medoid query returns one medoid in C_1 and another in C_2. Now consider that we wish to compute three medoids. Observe that, although C_1 has a smaller *diameter* than C_2, it contains more points. Due to the larger cardinality of C_1, the distances of its points from its medoid affect the global average distance to a greater extent than that of the points in C_2. Therefore, placing the third medoid in C_1 leads to a larger distance reduction than placing it in C_2. Intuitively, more medoids must be assigned to denser areas of the dataspace.

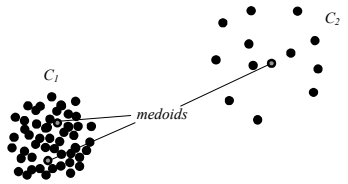


Figure 3.1: The three medoids of a dataset consisting of two clusters

Motivated by this observation, our algorithms (i) partition the points in P into k groups of (roughly) equal cardinality and, then, (ii) select the most centrally located object from each group as the corresponding medoid. To quickly perform step (i) we project the points on a one-dimensional space using a space filling curve. We employ the Hilbert curve since it is shown to best preserve locality compared to alternatives [MJFS01]. Next, we partition the Hilbert-sorted list of points into k groups of equal

[3] If the update is an insertion (deletion), $p.x_{old}$, $p.y_{old}$ ($p.x_{new}$, $p.y_{new}$) are set to a negative value.

cardinality (i.e., $|P|/k$). Due to the locality preservation of the Hilbert curve, the resulting groups can be regarded as well-defined partitions of P in the two-dimensional space. Finally, we extract a medoid from each group; the medoid is the point in the group with the median Hilbert value, as it is expected to be the most centrally located. The above rationale underlies both modules of our algorithms, namely, the initial medoid computation and their maintenance.

3.1 The HBM algorithm

Our first method is *Hilbert-based Monitoring* (*HBM*). It indexes the data objects with an in-memory 2-3 B^+-Tree [C79] (i.e., a B^+-Tree where each internal node has two or three children), using their Hilbert values as search keys. We denote this tree by *BT*. At the leaf level, except for the standard *right sibling* pointers, *BT* is modified to also accommodate *left sibling* pointers. In other words, the leaves are organized as a doubly connected linked list. When the continuous medoid query is installed at the server for the first time and *BT* is built, every entry E in an internal node N *temporarily* stores aggregate information about the number of points $E.a$ contained in its subtree. $E.a$ facilitates the initial medoid computation and is discarded afterwards.

In particular, according to our general approach, the i-th medoid of P is the $[(i\text{-}0.5)\cdot|P|/k]$-th object in the linear order imposed by the Hilbert values. HBM locates the k medoids by performing k traversals in *BT*, at a total cost of $O(k\cdot\log|P|)$. Before each traversal i, an auxiliary variable V is initialized to zero. The traversal starts from root N_R and it checks whether $V+E_1.a$ is larger than or equal to $(i\text{-}0.5)\cdot|P|/k$, where E_1 is N_R's first entry. If that is the case, the medoid is located in E_1's subtree and, therefore, the traversal continues by visiting E_1's child. Otherwise, $E_1.a$ is added to V and the algorithm continues similarly by checking $V+E_2.a$ against $(i\text{-}0.5)\cdot|P|/k$ (E_2 is N_R's second entry). V always keeps the number of points preceding (in the Hilbert order) the point with the smallest search key that is reachable by the traversal. Finally, the algorithm reaches the leaf node containing the i-th medoid. For every computed medoid m, an array M of size k stores a tuple of the form <*m.id*, *m.hv*, *m.ptr*, *m.off*>, where *m.id* is the identifier of the point selected as m, *m.hv* is m's Hilbert value, *m.ptr* points to the leaf node of *BT* that accommodates m, and *m.off* is an integer (initialized to zero) used by the maintenance module and whose functionality is explained later. The temporary *E.a* values are discarded after the end of the initial computation step. Figure 3.2 summarizes the data structures in HBM.

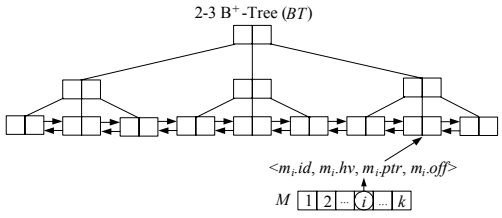


Figure 3.2: The data structures of the HBM method

The server periodically receives updates from the objects in batches. HBM accordingly updates *BT*, after computing the necessary Hilbert values of the inserted, deleted or moving points. Note that the movement of an object involves its deletion from the index followed by its subsequent re-insertion with the new Hilbert value. Whenever a split or merge operation moves a medoid m to a different leaf node, the corresponding *m.ptr* must also be altered in *M*. While updates are reflected in *BT*, HBM stores some book-keeping information, to be used for result maintenance according to its medoid selection strategy. In particular, after processing the insertion/deletion of a point p, HBM performs a binary search in array *M* to locate the leftmost medoid m_u with Hilbert value greater than (or equal to) $p.hv$. In case p initiated an insertion (deletion), the algorithm increases (decreases) $m_u.off$ by one. Particular care must be taken when a medoid m is deleted. In this case, HBM substitutes it with its predecessor in the Hilbert order and decreases $m.off$ by one.

After processing all updates, HBM computes the new medoids as follows. The i-th medoid m_i was formerly data point p_{old} at position $(i\text{-}0.5)\cdot|P|/k$. After the updates, p_{old} moves to position $(i\text{-}0.5)\cdot|P|/k + \Sigma_{j=1}^{i-1} m_j.off$. The actual medoid must be located at position $(i\text{-}0.5)\cdot|P'|/k$, where P' is the updated version of dataset P (which may have different cardinality if new objects were inserted or existing ones deleted). Therefore, the new medoid m_i can be found $OFF_i = (i\text{-}0.5)\cdot|P'|/k - (i\text{-}0.5)\cdot|P|/k - \Sigma_{j=1}^{i-1} m_j.off$ positions to the right or left of p_{old} in the linear order, depending on whether OFF_i is positive or negative, respectively. For every medoid m_i in *M*, HBM first visits the leaf node pointed by $m_i.ptr$ to find its old corresponding point p_{old}. Then, using the left/right sibling pointers of *BT*, it locates the new medoid and properly updates m_i's entry in *M*. The pseudocode of the maintenance procedure is given in Figure 3.3.

```
Function updateMedoids (array M, Tree T)
1.  Initialize V to 0
2.  For i=1 to k
3.      Locate medoid m_i in leaf M[i].ptr of T
4.      OFF_i = (i-0.5)·|P'|/k - (i-0.5)·|P|/k - V
5.      If OFF_i = = 0, continue
6.      Else if OFF_i > 0, find point p located |OFF_i| positions to the right of m_i
7.      Else if OFF_i < 0, find point p located |OFF_i| positions to the left of m_i
8.      V += M[i].off;
9.      Assign p.id, p.hv, the pointer of the leaf of T that accommodates p and 0 to M[i].id,
        M[i].hv, M[i].ptr and M[i].off, respectively
```

Figure 3.3: The maintenance module of HBM

Figure 3.4 illustrates the initial computation and maintenance of $k = 2$ medoids in a set of points, which at timestamp T_1 has cardinality 14. For ease of demonstration, we omit the *BT* operations and focus on the leaf level of the tree, which constitutes a doubly connected linked list of points sorted on their Hilbert values. At timestamp T_1, the set is subdivided into two subsets of seven points each. The medians of the subsets (p_4 and p_{12}) are selected as the medoids (m_1 and m_2, respectively). At timestamp T_2, four updates occur; p_1 and p_{13} are deleted, and p_3 and p_5 move to new positions. Due to p_1's deletion, $m_1.off$ is decreased by one. On the contrary, the deletion of p_{13} does not affect any *off* value because there is no medoid with higher (or equal) Hilbert

value. Regarding p_3 and p_5, recall that a point movement is handled as a deletion followed by an insertion. Upon p_3's deletion, the algorithm decreases $m_1.off$. Subsequently, the point is re-inserted in a position between m_1 and m_2 and, therefore, $m_2.off$ is increased by one. Finally, p_5's movement causes $m_2.off$ to decrease (due to its deletion) and immediately increase (due to its re-insertion) by one, because both its old and new Hilbert values are between $m_1.hv$ and $m_2.hv$. Let old_pos_i be the position (in the Hilbert order) of the point that was selected as medoid m_i at timestamp T_1. Also let $curr\_pos_i$ be the position of the new point to become m_i at timestamp T_2. For m_1, $old\_pos_1 = 4$, $curr\_pos_1 = 3$, and $OFF_1 = 1$. Similarly for m_2, $old\_pos_2 = 11$, $curr\_pos_2 = 9$, and $OFF_2 = -1$. The algorithm locates the new medoids p_3 and p_{11}, by moving one position to the right and one to the left from old medoids p_4 and p_{12}, respectively.

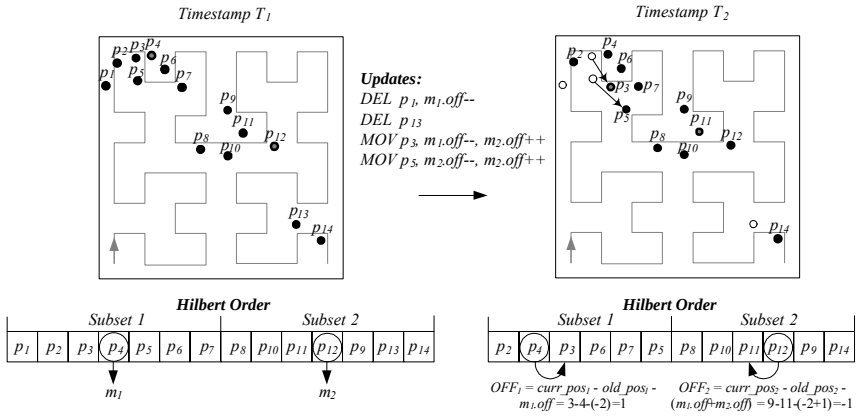


Figure 3.4: A medoid monitoring example in HBM

3.2 The GBM algorithm

The *Grid-based Monitoring* (*GBM*) algorithm utilizes a $C \times C$ regular grid for indexing P. Let δ be the side-length of each cell. A point p in P with coordinates ($p.x$, $p.y$) can be located in constant time in cell $c_{i,j}$ (i.e., the cell in column i and row j, starting from the low-left corner of the grid), where $i = \lfloor p.x/\delta \rfloor$ and $j = \lfloor p.y/\delta \rfloor$. GBM imposes a linear order on the cells by sorting them according to the Hilbert values of their centers. Every cell c is associated with a tuple <$c.n$, $c.prev$, $c.next$, $c.BT$>, where $c.n$ is the cardinality of the set of points contained in c, $c.prev$ and $c.next$ are the cells preceding and succeeding c in the Hilbert order respectively, and $c.BT$ is a BT that indexes the points in c (using their Hilbert values as search keys). Similarly to HBM, the internal nodes in the BTs temporarily incorporate aggregate information, which is discarded after the initial computation of the medoids.

The grouping strategy of GBM is similar to HBM, the difference being in the linear order of the points, which now takes into account firstly the order of the cells. Specifically, the points are considered sorted according to the following rules; (i) a point p_1 in cell c_1 precedes point p_2 in cell c_2, if c_1 precedes c_2 in their Hilbert order, and (ii) the order of the points in the same cell is determined by their Hilbert values. Following similar reasoning as in HBM, the i-th medoid m_i is the $[(i\text{-}0.5)\cdot|P|/k]$-th object in the above order. GBM starts by initializing an auxiliary variable V to zero and scans the linked list of the (sorted) cells. To locate medoid m_i, in every visited cell c_i, it checks whether $V+c_i.n$ is larger than or equal to $(i\text{-}0.5)\cdot|P|/k$. If that is the case, it traverses $c_i.BT$ in order to find the $[V+c_i.n-(i\text{-}0.5)\cdot|P|/k]$-th object in the cell, which is then selected as medoid m_i. Otherwise, it adds $c_i.n$ to V and continues to the next cell. V keeps the number of points encountered by the scan so far. Note that GBM locates all medoids in a single linear scan of the cells, i.e., after finding medoid m_i, it does not restart the scan for finding m_{i+1}; instead, it continues from the cell that contains m_i. Finally, it maintains an array M with functionality identical to that used by HBM. Figure 3.5 depicts the data structures of GBM.

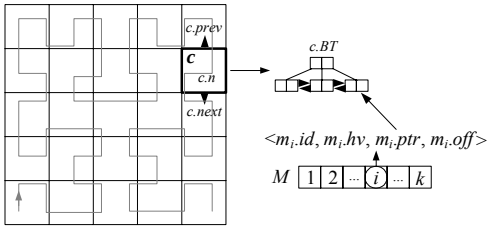


Figure 3.5: The data structures of the GBM method

For every received update, GBM first determines in constant time the cell c where the insertion/deletion takes place, and properly updates $c.BT$. Subsequently, it scans M and updates the *off* value of the leftmost medoid with Hilbert value larger than or equal to that of the object that initiated the update, in a similar fashion to HBM. After processing all the updates, the maintenance module of GBM identifies the points to be selected as the new medoids as follows. It scans M and for every m_i, it computes OFF_i in a fashion similar to Section 3.1. Suppose that m_i lies in cell c. Then, starting from the leaf of $c.BT$ that accommodates m_i and is pointed by $m_i.ptr$, it searches for the point that will be selected as the new m_i. This point lies OFF_i positions to the left or right of old m_i, depending on whether OFF_i is negative or positive, respectively. If the search reaches the leftmost or rightmost (in the Hilbert order) point of cell c, it continues to the cell pointed by $c.prev$ or $c.next$, respectively. Note that the algorithm may skip entire cells (i.e., it may not traverse their BTs at all), since it can always determine whether m_i is located in a visited cell by comparing the cell's cardinality against OFF_i. After finding a new medoid, GBM updates the respective entry in M accordingly.

In Figure 3.6 we exemplify the initial medoid computation and monitoring in a scenario where $k = 2$ and P contains points p_1 to p_{14}. Consider cells $c_{2,2}$ and $c_{1,2}$ at timestamp T_1. The Hilbert curve first passes through $c_{2,2}$ and, thus, p_{11} precedes p_1 in

the GBM order, although it succeeds it in the global Hilbert order (i.e., $p_{11}.hv > p_1.hv$, where $p_{11}.hv$ and $p_1.hv$ are the Hilbert values of p_{11} and p_1, respectively). At timestamp T_1, the medoids are $m_1 = p_4$ and $m_2 = p_{14}$, since they are at positions $0.5 \cdot |P|/k = 4$ and $1.5 \cdot |P|/k = 11$, respectively, in the linear order. At timestamp T_2, objects p_7, p_6 and p_{11} issue updates, as shown in the figure. Their movement leads to $OFF_1 = 1$ and $OFF_2 = 1$, and updates the medoids to $m_1 = p_7$ and $m_2 = p_{11}$.

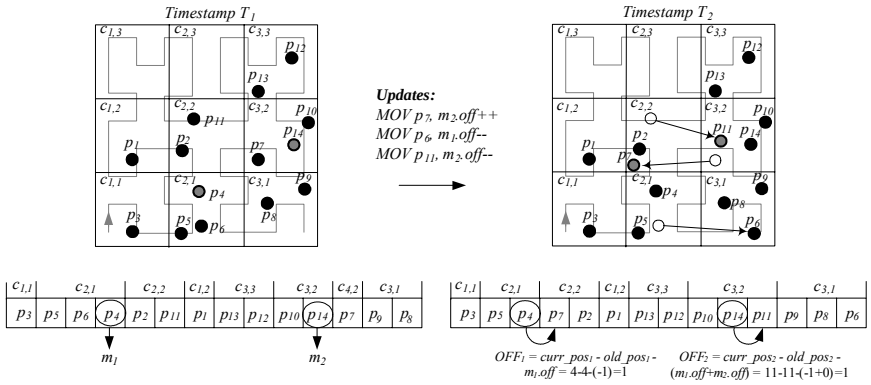


Figure 3.6: A medoid monitoring example in GBM

Compared to HBM, index update and medoid maintenance in GBM are expected to be faster. HBM keeps a common *BT* over all $|P|$ points, which leads to an $O(\log|P|)$ cost for every point insertion or deletion. On the other hand, letting c be the cell of the inserted/deleted point, $c.BT$ contains $c.n$ objects (where $c.n << |P|$), requiring $O(\log|c.n|)$ time per update. Furthermore, maintaining the medoids is also more efficient in GBM, because for large OFF_i values, entire cell contents may be skipped when sliding in the linear point order towards the new medoid position. Another major advantage of GBM over HBM, is the fact that its data index is compatible with existing methods for other spatial query types; most range and nearest neighbor monitoring algorithms use a regular grid index[4]. This allows GBM to be used in conjunction with other methods, in a system that answers general spatial queries over moving objects, utilizing a single data index.

A final remark concerns the average distance, which is in general different but similar for GBM and HBM, since their medoid selection rationale is alike. In particular, if the grid granularity in HBM is selected so that C is a power of two (recall that the grid has $C \times C$ cells), their medoids are identical. The reason is that the Hilbert values themselves are computed by definition based on a transparent space partitioning with a grid, whose granularity on each axis is always a power of two (this power is called the *order* of the Hilbert curve). If C is also a power of two, the cells of the object grid contain continuous, non-overlapping intervals of the curve. In other words, if cell c_1 precedes c_2 on the curve, then *any* point p_1 in c_1 precedes *every* p_2 in c_2. In turn, this fact implies that the linear point orders of GBM and HBM are identical and, thus, the medoids are the same.

[4] All methods covered in Section 2.2 use a regular grid, except for [MPBT05] and [HXL05], where processing time minimization is not the main objective.

4 Distributed Medoid Monitoring

The main idea in the distributed version of our methods is to allow objects to move within assigned *safe regions*, without having to transmit updates to the server. Since our general medoid selection strategy relies on a linear point order, the safe regions are defined with respect to the neighboring objects (in the order). Particularly, let *leeway* λ be an integer system parameter. The safe region of the i-th object in the order p_i is a Hilbert interval $SR_i^\lambda = [p_i.sr_L, p_i.sr_R]$. The left boundary $p_i.sr_L$ is the mean of the Hilbert values of p_i and its λ-th left neighbor $p_{i-\lambda}$ (i.e., $p_i.sr_L = \lfloor (p_i.hv + p_{i-\lambda}.hv)/2 \rfloor$). The right boundary $p_i.sr_R$ is set similarly with respect to the λ-th right neighbor (i.e., $p_i.sr_R = \lceil (p_i.hv + p_{i+\lambda}.hv)/2 \rceil$). Object p_i may change location without issuing an update, as long as $p_i.hv \in SR_i^\lambda$. When p_i does move outside SR_i^λ, it sends its new location to the server. The latter updates its index and the medoid set accordingly[5], and assigns a new safe region to p_i. Note that the new SR_i^λ is defined based on the latest point positions reported. Particularly for GBM, the linear point order takes into account the grid cells ordering. Thus, the safe regions are defined within each cell individually (i.e., in the Hilbert order of the objects therein). Whenever an object exits its cell, it sends an update regardless of whether it violates its safe region.

Figure 4.1 demonstrates the safe region function in the case of HBM (the case of GBM is similar, subject to the aforementioned modifications), showing the position of the points on the Hilbert curve. At timestamp T_1, the safe region $SR_3^{\lambda=1}$ ($SR_3^{\lambda=2}$) of p_3 is defined according to p_2 and p_4 (p_1 and p_5) for $\lambda = 1$ ($\lambda = 2$). Similarly, $SR_4^{\lambda=1}$ is determined by p_3 and p_5. Assuming that $\lambda = 1$, at timestamp T_2, points p_3 and p_4 move. However, only p_3 issues an update, because p_4 remains within its safe region. The solid points in the figure correspond to the positions known by the server, the hollow point is p_3's old Hilbert value, while the grey is p_4's actual one. Object p_3 is assigned a new region, based on the Hilbert values of p_2 and p_4. Note that the server is not aware of the new location of p_4 and, thus, uses the last reported one (as of T_1).

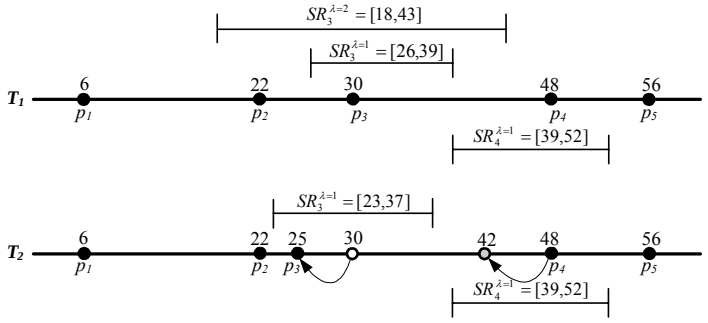


Figure 4.1: Safe regions and update handling

[5] Medoid maintenance at the server side is identical to the centralized case.

5 Experimental Evaluation

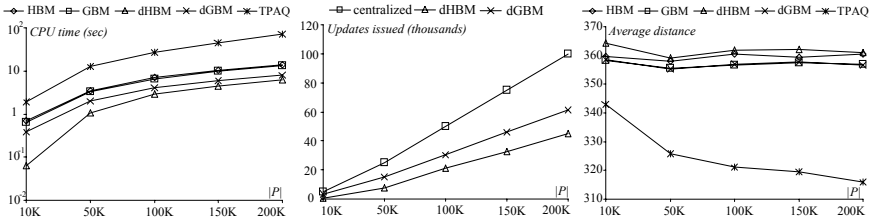
In this section we evaluate the performance of our methods, in terms of processing time (at the server), number of object updates (i.e., communication cost for the objects) and achieved average distance. We generate datasets of cardinality $|P|$ ranging between 10K and 200K objects as follows. For each tested $|P|$, we randomly select the initial position and the destination of each object among the points of a real spatial dataset (North America, available at *www.maproom.psu.edu/dcw*). The object follows a linear trajectory between the two points. Upon reaching the endpoint, a new random destination is selected and the process is repeated. At every timestamp, a percentage a of the objects move towards their endpoint (while the remaining ones remain static), covering a distance v. We refer to a and v as the object *agility* and *velocity*, respectively. The velocity is expressed as a percentage of the dataspace extent on the x axis (we have a $[0,10^4]\times[0,10^4]$ dataspace). The simulation length is 100 timestamps for each setting, and the reported measurements are the average observed values over all timestamps. We process continuous k-medoid queries for k between 2 and 512. We evaluate our four methods HBM, GBM, dHBM, and dGBM (where the latter two are the distributed versions of HBM and GBM). Also, we use as a competitor the TPAQ method with a main memory R-tree, since none of the other existing algorithms works for the large cardinalities tested, even for snapshot queries. To adapt TPAQ to medoid monitoring, we rerun it for the timestamps where (i) some of the medoids move, or (ii) the object updates affect the extents of the R-tree entries at the partitioning level. In each experiment we vary one parameter, while setting the remaining to their default values. The parameter ranges and defaults are shown in Table 5.1. For GBM and dGBM we fine-tuned the grid granularity (with respect to the average distance) for the default settings and use the best one (100×100) in all our experiments. We use a machine with a 3.2 GHz Pentium IV CPU and 1 GB RAM.

Parameter	Default	Range
Dataset cardinality $\|P\|$	100K	10, 50, 100, 150, 200 (K)
No. of medoids k	32	2, 8, 32, 128, 512
Agility a	50%	10, 30, 50, 70, 100 (%)
Velocity v	0.5%	0.1, 0.3, 0.5, 0.7, 1 (%)
Leeway λ	300	100, 200, 300, 400, 500

Table 5.1: Parameter ranges and default values

In Figure 5.1, we measure the effect of object cardinality $|P|$, varying it from 10K to 200K objects and setting the other parameters to their defaults. Figure 5.1a shows the CPU cost (in logarithmic scale) for medoid maintenance per timestamp, i.e., the time to update the object index and the medoids. We observe that the centralized methods have similar cost (with GBM being slightly faster). The distributed algorithms have shorter running time, because they process fewer updates; dHBM (dGBM) takes less than 45% (60%) of the time of its centralized counterpart. dHBM is faster than dGBM, because the latter's safe regions are practically smaller, as they are bounded by the grid cell boundaries (leading to more reported updates and, thus, higher processing cost). Compared to our methods, TPAQ is slower by an order of

magnitude, mainly due to the excessive update cost of its R-tree index. An important remark about Figure 5.1a (and all remaining CPU time charts) is that we focus on pure maintenance cost, i.e., we exclude the initial k-medoid computation. For the sake of completeness, the first-time medoid extraction for the default setting takes 12.9, 12.4 and 54.4 sec for HBM, GBM and TPAQ, respectively (the times for dHBM and dGBM are identical to HBM and GBM).

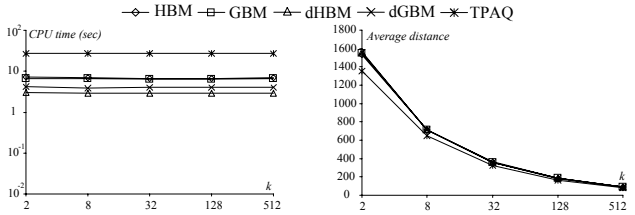


(a) CPU time (b) Number of updates (c) Average distance

Figure 5.1: Performance versus dataset cardinality $|P|$

Figure 5.1b shows the number of updates sent to/processed by the server in the same experimental setup. All centralized methods (i.e., HBM, GBM, TPAQ) have the same communication cost, with the objects reporting their positions whenever they move. On the other hand, the safe regions of dHBM and dGBM save around 55% and 40% of these updates, respectively. dGBM avoids less updates than dHBM, due to the necessary updates required when the objects move to another cell, as explained in the context of Figure 5.1a. Figure 5.1c illustrates the achieved distance for the various cardinalities, expressed in distance units in our $[0,10^4]\times[0,10^4]$ dataspace. We observe that the distributed methods compute only slightly worse medoid sets, verifying their efficacy. Note that both versions of GBM are better than those of HBM. The reason is that HBM is solely based on the one-dimensional Hilbert mapping, while GBM preserves a stronger connection to the original (two-dimensional) space, due to its spatial grid index. For a similar reason, TPAQ achieves 4 to 11% smaller distance than our methods, exploiting the graceful grouping properties of its R-tree. However, this benefit comes to a prohibitive update cost, leading to an excessive processing time (see Figure 5.1a). Another remark for TPAQ is that it improves with $|P|$; for a denser space, the nearest neighbor queries (in its final step) retrieve medoids that lie closer to the "ideal" geometric centroids of the k groups, leading to a lower distance.

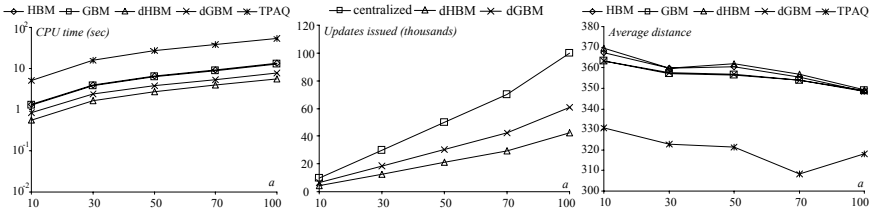
In Figure 5.2 we use the default settings and vary k between 2 and 512. Figure 5.2a shows the CPU time. Again dGBM is the fastest, for the reasons explained above. We observe that the processing cost is almost constant for each method and unaffected by k. The reason is that, in all methods (and especially in TPAQ), the monitoring cost is dominated by the number of processed updates (mainly due to index maintenance), which is irrelevant to k. Furthermore, in our algorithms, for larger k, there are more medoids to maintain, but the offsets (to slide in the linear point order) are smaller. On the other hand, the average distance drops with k for all methods, and our techniques' difference from TPAQ decreases.



(a) CPU time (b) Average distance

Figure 5.2: Performance versus number of medoids *k*

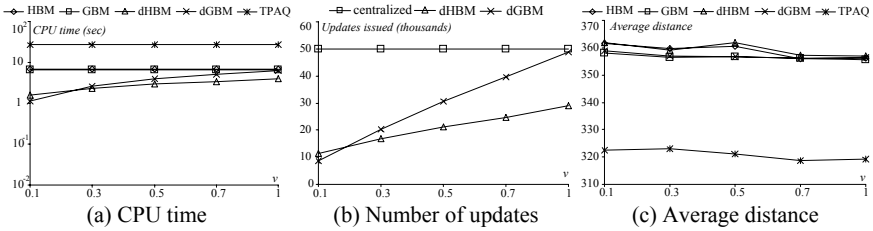
In Figure 5.3 we examine the effect of object agility *a*, with 10% up to 100% of the data points moving at each timestamp. The CPU cost (Figure 5.3a) increases with *a* due to the larger number of updates processed. Figure 5.3b shows the number of issued updates, which, as expected, is linear to *a*. In terms of average distance (Figure 5.3c), there is not much fluctuation; the small differences are due to the randomness of the dataset generation.



(a) CPU time (b) Number of updates (c) Average distance

Figure 5.3: Performance versus object agility *a*

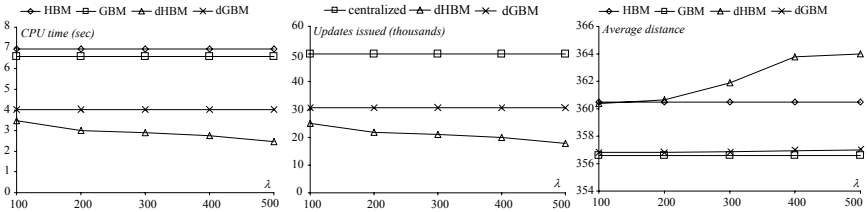
In Figure 5.4 we vary the object velocity *v* from 0.1 to 1% of the dataspace extent on the *x* dimension. Figure 5.4a shows the CPU time. The centralized methods are unaffected by *v*. On the other hand, the cost of the decentralized increases as more objects move outside their safe regions for larger *v*, sending more updates to the server for processing. This is also evident in Figure 5.4b. Interestingly, for $v = 0.1\%$, dGBM incurs less object updates than dHBM (because its cells are large with respect to *v*, without practically limiting the safe regions), while for $v = 1\%$ their number is almost as high as for the centralized methods. The average distance (Figure 5.4c) is similar for all values of *v*.



(a) CPU time (b) Number of updates (c) Average distance

Figure 5.4: Performance versus object velocity *v*

Figure 5.5 investigates the effect of the leeway λ, varying it from 100 to 500. The performance of the centralized methods is identical, because they do not use safe regions. As shown in Figure 5.5b, for λ = 500, dHBM achieves 65% reduction of the location updates. For dGBM, however, there is a marginal decrease, because the safe regions are restricted by the grid cells, rather than by λ. The number of updates has a direct impact on the CPU time and, thus, the trends in Figure 5.5a are similar as in Figure 5.5b. In terms of average distance, λ affects only dHBM, whose performance deteriorates for larger λ. This trend verifies the tradeoff between update cost and medoid quality. On the other hand, dGBM is not affected because the server processes a similar set of updates.



(a) CPU time (b) Number of updates (c) Average distance

Figure 5.5: Performance versus leeway λ

6 Conclusion

In this paper we address the problem of *k*-medoid monitoring. To the best of our knowledge this is the first work on this topic. We consider a central server that continuously receives the locations of frequently moving objects and incrementally maintains their medoid set. Without making any assumption about the data moving patterns, our methods achieve low running times while keeping the medoid quality high. Furthermore, we consider distributed environments, where the data objects have limited power resources and attempt to preserve them by reducing the number of updates they transmit to the server. In this context, the server assigns safe regions to the objects, which report their position only when they exit their region. We evaluate our methods through extensive experiments and investigate tradeoffs between communication cost and medoid quality.

Acknowledgements

This work was supported by grant HKUST 6184/06E from Hong Kong RGC, and by an award from the Lee Foundation.

References

[ARR98] Arora, S., Raghavan, P., Rao, S. Polynomial Time Approximation Schemes for Euclidean k-Medians and Related Problems. *STOC*, 1998.

[ABKS99] Ankerst, M., Breunig, M., Kriegel, H.P., Sander, J. OPTICS: Ordering Points To Identify the Clustering Structure. *SIGMOD*, 1999.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[C79] Comer, D. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2): 121-137, 1979.

[CHC04] Cai, Y., Hua, K., Cao, G. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. *MDM*, 2004.

[EKX95a] Ester, M., Kriegel, H. P., Xu, X. A Database Interface for Clustering in Large Spatial Databases. *KDD*, 1995.

[EKX95b] Ester, M., Kriegel, H. P., Xu, X. Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification. *SSD*, 1995.

[EKSX96] Ester, M., Kriegel, H. P.,Sander, J., Xu, X. A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, *KDD*, 1996.

[G84] R-Trees: A dynamic index structure for spatial searching. *SIGMOD*, 1984.

[GJ79] Garey, M., Johnson, D. Computers and Intractability: A Guide to the Theory of NP-Completeness. *W.H. Freeman*, 1979.

[GL04] Gedik, B., Liu, L. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. *EDBT*, 2004.

[GMM+03] Guha, S., Meyerson, A., Mishra, N., Motwani, R., O'Callaghan, L. Clustering Data Streams: Theory and Practice. *IEEE TKDE*, 15(3): 515-528, 2003.

[GRS98] Guha, S., Rastogi, R., Shim, K. CURE: An Efficient Clustering Algorithm for Large Databases. *SIGMOD*, 1998.

[HXL05] Hu, H., Xu, J., Lee, D. A generic framework for monitoring continuous spatial queries over moving objects. *SIGMOD*, 2005.

[KOTZ04] Koudas, N., Ooi, B., Tan, K., Zhang, R. Approximate NN queries on Streams with Guaranteed Error/performance Bounds. *VLDB*, 2004.

[KR90] Kaufman, L., Rousseeuw, P. Finding Groups in Data. *Wiley-Interscience*, 1990.

[MHP05] Mouratidis, K., Hadjieleftheriou, M., Papadias, D. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. *SIGMOD*, 2005.

[MJFS01] Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TKDE*, 13(1): 124-141, 2001.

[MPBT05] Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y. A Threshold-based Algorithm for Continuous Monitoring of *k* Nearest Neighbors. *IEEE TKDE*, 17(11): 1451-1464, 2005.

[MPP] Mouratidis, K., Papadias, D., Papadimitriou S. Tree-based Partition Querying: A Methodology for Computing Medoids in Large Spatial Datasets. To appear in *VLDBJ*.

[MXA04] Mokbel, M., Xiong, X., Aref, W. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. *SIGMOD*, 2004.

[NH94] Ng, R., Han, J. Efficient and Effective Clustering Methods for Spatial Data Mining. *VLDB*, 1994.

[PXK+02] Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W., Hambrusch, S. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10): 1124-1140, 2002.

[XMA05] Xiong, X., Mokbel, M., Aref, W. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *ICDE*, 2005.

[YPK05] Yu, X., Pu, K., Koudas, N. Monitoring K-Nearest Neighbor Queries Over Moving Objects. *ICDE*, 2005.

[ZDXT06] Zhang, D., Du, Y., Xia, T., Tao, Y. Progressive Computation of the Min-Dist Optimal-Location Query. *VLDB*, 2006.

[ZRL96] Zhang, T., Ramakrishnan, R., Livny, M. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD*, 1996.