Continuous k-Means Monitoring over Moving Objects

Journal:	*Transactions on Knowledge and Data Engineering*
Manuscript ID:	TKDE-2007-10-0510.R1
Manuscript Type:	Regular
Keywords:	H.2.8.d Data mining < H.2.8 Database Applications < H.2 Database Management < H Information Technology and Systems, H.2.8.o Spatial databases and GIS < H.2.8 Database Applications < H.2 Database Management < H Information Technology and Systems

Continuous *k*-Means Monitoring over Moving Objects

Zhenjie Zhang, Yin Yang, Anthony Tung, and Dimitris Papadias

Abstract— Given a dataset *P*, a *k*-means query returns *k* points in space (called *centers*), such that the average squared distance between each point in *P* and its nearest center is minimized. Since this problem is NP-hard, several approximate algorithms have been proposed and used in practice. In this paper, we study continuous *k*-means computation at a server that monitors a set of moving objects. Re-evaluating *k*-means every time there is an object update imposes a heavy burden on the server (for computing the centers from scratch) and the clients (for continuously sending location updates). We overcome these problems with a novel approach that significantly reduces the computation and communication costs, while guaranteeing that the quality of the solution, with respect to the re-evaluation approach, is bounded by a user-defined tolerance. The proposed method assigns each moving object a threshold (i.e., range) such that the object sends a location update only when it crosses the range boundary. First, we develop an efficient technique for maintaining the *k*-means. Then, we present mathematical formulae and algorithms for deriving the individual thresholds. Finally, we justify our performance claims with extensive experiments.

Regular Submission to TKDE, October 2007

Index Terms—*k*-means, Continuous monitoring, Query processing

1 Introduction

Efficient *k*-means computation is crucial in many practical applications, such as clustering, facility location planning and spatial decision making. Given a dataset $P = \{p_1, p_2, \ldots p_n\}$, a *k*-means query returns a *center set* M of k points $\{m_1, m_2, \ldots, m_k\}$, such that $cost(M) = \sum dist^2(p_i, NN(p_i,M))$ is minimized, where $NN(p_i,M)$ is the nearest neighbor of p_i in M, and *dist* is a distance (usually, Euclidean) metric. The data points whose NN is $m_i \in M$ form the *cluster* of m_i. Since the problem is NP-hard, the vast majority of existing work focuses on approximate solutions. In the data mining literature, several methods are based on *hill climbing* (HC). Specifically, HC starts with *k* random *seeds* as centers, and iteratively adjusts them until it reaches a *local optimum*, where no further improvement is possible. Then, it repeats this process, each time using a different set of seeds. The best local optimum among the several executions of HC constitutes the final result.

Recently, there is a paradigm shift from snapshot queries on static data to long-running queries over moving objects. In this scenario, mobile clients equipped with GPS devices send their locations to a central server. The server collects these locations and continuously updates the results of registered spatial queries. Our work focuses on *continuous k-means monitoring* over moving objects. Compared to conventional spatial queries (e.g. ranges or nearest neighbors), *k*-means monitoring is fundamentally more complex. The main reason is that in simple queries, the results are determined solely by the objects' *individual* locations with respect to a single reference (i.e., query) point. On the other hand, a *k*-means set depends upon the *combined* properties of all objects. Even a slight movement by an object causes its corresponding center to change. Moreover, when the object is close to the boundary of two clusters, it may be assigned to a different center, and the update may have far-reaching effects.

A simple method for continuous *k*-means monitoring, hereafter called REF (short for *reference* solution), works as follows. When the system starts at time τ=0, every object reports its location, and the server computes the *k*-means set $M(0)$ through the HC algorithm. Subsequently (τ>0), whenever an object moves, it sends a location update. The server obtains $M(\tau)$ by executing HC on the updated locations, using $M(\tau\text{-}1)$ as the seeds. The rationale is that $M(\tau)$ is more similar to $M(\tau\text{-}1)$ than a random seed set, reducing the number of HC iterations. REF produces high quality results because it continuously follows every object update. On the other hand, it incurs large communication and computation cost due to the frequent updates and re-computations.

To eliminate these problems, we propose a *threshold-based k-means monitoring* (TKM) method, based on the framework of Figure 1.1. In addition to *k*, a continuous *k*-

- *Z. Zhang and A. Tung are with the Department of Computer Science, National University of Singapore, 117590 Singapore. Email: {zhangzh2, atung}@comp.nus.edu.sg*
- *Y. Yang and D. Papadias are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. Email: {yini,dimitris}@cse.ust.hk*

means query specifies a *tolerance* Δ. The computation of $M(0)$ is the same as in REF. After the server computes a center set, it sends to every object p_i a *threshold* θ_i, such that p_i needs to issue an update, only if its current location deviates from the previous one by at least θ_i. When the server receives such an update, it obtains the new k-means using HC*, an optimized version of HC, over the last recorded locations of the other objects (which are still within their assigned thresholds). Then, it computes the new threshold values. Because most thresholds remain the same, the server only needs to send messages to a fraction of the objects. Let M^{REF} (M^{TKM}) be the k-means set maintained by REF (TKM). We prove that for any time instant and for all $1 \le i \le k$: $dist(m_i^{REF}, m_i^{TKM}) \le \Delta$, where $m_i^{REF} \in M^{REF}$ and $m_i^{TKM} \in M^{TKM}$; i.e., each center in M^{TKM} is within Δ distance of its counterpart in M^{REF}.

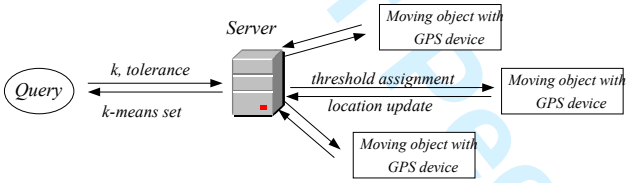


Figure 1.1 Threshold-based k-means monitoring

Our contributions are:

1. We present TKM, a general framework for continuous k-means monitoring over moving objects.
2. We propose HC*, an improved HC, which minimizes the cost of each iteration by only considering a small subset of the objects.
3. We model the threshold assignment task as a constrained optimization problem, and derive the corresponding mathematical formulae.
4. We develop an algorithm for the efficient computation of thresholds.
5. We design different mechanisms for the dissemination of thresholds, depending on the computational capabilities of the objects.

The rest of this paper is organized as follows: Section 2 overviews related work. Section 3 presents TKM and the optimized HC* algorithm. Section 4 focuses on the mathematical derivation of thresholds, proposes an algorithm for their efficient computation, and describes protocols for their dissemination. Section 5 contains a comprehensive set of experiments, and Section 6 concludes with directions for future work.

2 Background

Section 2.1 overviews k-means computation over static datasets. Section 2.2 surveys previous work on continuous monitoring and problems related to k-means.

2.1 *k*-Means Computation for Static Data

Numerous methods for computing k-means over static data have been proposed in the theoretical literature. Inaba et al. [IKI94] present an $O(n^{O(kd)})$ algorithm for optimal solutions and a $O(n(1/\varepsilon)^d)$ method for ε-approximate 2-means, where n is the data cardinality and d the dimensionality. Kumar et al. [KSS04] develop a $(1+\varepsilon)$-approximate algorithm for k-means in any Euclidean space, which is linear to n and d. Kanungo et al. [KMN+02] describe a swapping technique achieving $1+\varepsilon$-approximation. Several studies focus on the convergence speed of k-means algorithms [HS05, AV06].

Most data mining literature has applied hill climbing (HC) [L82] for solving k-means. Figure 2.1 shows a general version of the algorithm. HC starts with a set M that contains k random seeds, and iteratively improves it. Each iteration consists of two steps: the first (Lines 3-4) assigns every point to its nearest center in M, and the second (Lines 5-6) replaces each $m \in M$ with the *centroid* of its assigned points, which is the best location for minimizing the cost function (i.e., average squared distance) for the current point assignment [KSS04] [1]. HC *converges* at a *local optimum*, when there can be no further improvement on M. Since a local optimum is not necessarily the global optimum, usually several runs of the algorithm are executed with different sets of seeds. The best local optimum of these runs is returned as the final output. Although there is no guarantee on efficiency (a large number of iterations may be needed for convergence), or effectiveness (a local optimum may deviate significantly from the best solution), HC has been shown to perform well in practice.

HC (*dataset P, k*)
1. Choose k random seeds as the initial center set M
2. Repeat
3. For each point $p \in P$
4. Assign p to its nearest center in M
5. For each center m in M
6. Replace m with the centroid of all $p \in P$ assigned to m
7. Until no change happens in M

Figure 2.1 Hill climbing for computing k-means

Zhang et al. [ZDT06] propose a method, hereafter referred to as ZDT, for predicting the best possible cost that can be achieved by HC. We describe ZDT in detail because it is utilized by the proposed techniques. Let M be the current k-means set after one iteration of HC (i.e., after Line 6). At present, each center m in M is the centroid of its cluster. For a point p, we use m_p to denote the currently assigned center of p, which is not necessarily the nearest. A key observation is that there is a constant δ, such that in all subsequent HC iterations, no center can deviate from its position in M by more than δ. Equivalently, as shown in Figure 2.2, each $m \in M$ is restricted to a *confinement circle* centered at its current position with radius δ. Therefore, when HC eventually converges to a local optimum M^*, each center in M^* must be within its corresponding confinement circle. Zhang et al. [ZDT06] prove that $cost(M)$-$n\delta^2$ is a lower bound of $cost(M^*)$, where n is the data cardinality. If this bound exceeds the best cost achieved by a previous local optimum (with a different seed set), subsequent iterations of HC can be pruned.

[1] The centroid is computed by taking the average coordinates of all assigned points on each dimension.

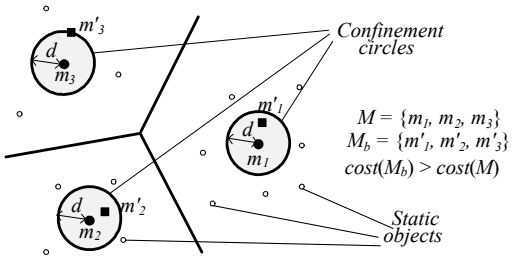


Figure 2.2 Centers restricted in confinement circles

It remains to clarify the computation of δ. Let M_b be a set constructed by moving one of the centers in M to the boundary of its confinement circle, and the rest within their respective confinement circles. In the example of Figure 2.2, $M=\{m_1, m_2, m_3\}$, $M_b=\{m'_1, m'_2, m'_3\}$, and m'_3 is on the boundary of the confinement circle of m_3. The defining characteristic of δ is that for *any* such M_b, $cost(M_b) > cost(M)$. We cannot compute δ directly from the inequality $cost(M_b) > cost(M)$ because there is an infinite number of possible M_b's. Instead, ZDT derives a lower bound $LB(M_b)$ for $cost(M_b)$ of *any* M_b, and solves the smallest δ satisfying $LB(M_b) > cost(M)$. Let P_m ($|P_m|$) be the set (cardinality) of data points assigned to center m. It can be shown [ZDT06] that:

$$LB(M_b) = \sum_{p\in P} dist^2(p,m_p) + min_m(|P_m|\delta^2) - \sum_{p\in P_{move}} Y(p)$$

$$\text{where } Y(p) = \left(dist(p,m_p)+\delta\right)^2 - \left(max\left(0, min_{m\in M\setminus\{m_p\}} dist(p,m) - \delta\right)\right)^2 \quad 2.1$$

$$\text{and } P_{move} = \{p \mid Y(p) > 0\}$$

The intuition behind Equation 2.1 is: (i) we first assume that every point p stays with $m'_p \in M_b$, which corresponds to $m_p \in M$, the currently assigned center of p, and calculate the minimum cost of M_b under such an assignment; then (ii) we estimate an upper bound on the cost reduction by shifting some objects to other clusters. In step (i), the current center set M is optimal with respect to the current point assignment (every $m \in M$ is the centroid of its assigned points P_m) and achieves cost $\sum_{p\in P} dist^2(p,m_p)$. Meanwhile, moving a center $m \in M$ by δ increases the cost by $|P_m|\delta^2$ [KSS04]. Therefore, the minimum cost of M_b when one center moves to the boundary of its confinement circle (while the rest remain at their original locations) is $\sum_{p\in P} dist^2(p,m_p) + min_m(|P_m|\delta^2)$.

In step (ii), let P_{move} be the set of points satisfying the following property: for any $p \in P_{move}$, if we re-assign p to another center in M_b (i.e. other than m'_p), the total cost must decrease. Function $Y(p)$ upper-bounds the cost reduction by re-assigning p. Clearly, p is in P_{move} iff $Y(p) > 0$. Regarding $Y(p)$, it is estimated using the (squared) maximum distance $dist(p,m_p)+\delta$ between p and its assigned center $m'_p \in M_b$, subtracted by the (squared) minimum distance $min_{m\in M\setminus\{m_p\}} dist(p,m)-\delta$ between p and any other center $m' \in M_b$, $m' \neq m'_p$. The former is reached when m'_p is δ away from p than its original location $m_p \in M$; the latter is reached when the nearest center $m \in M$ has a corresponding location m' in M_b which is δ closer to p than m. Recall that from the definition of M_b, the distance between a center $m \in M$ and its corresponding center $m' \in M_b$ is at most δ. Finally, Zhang et al. [ZDT06] provide a method that computes the minimum δ in $O(n \log n)$ time, where n is the data cardinality. We omit further details of this algorithm since we do not compute δ in this work.

2.2 Other Related Work

Clustering, one of the fundamental problems in data mining, is closely related to k-means computation. For a comprehensive survey on clustering techniques see [JMF99]. Previous work has addressed the efficient maintenance of clusters over data streams (e.g. [BDMO03, GMM+03]), as well as moving objects (e.g. [LHY04, KMB05]). However, in the moving object setting, the existing approaches require the objects to report every update, which is prohibitively expensive in terms of network overhead and battery power. In addition, some methods, such as [LHY04], are limited to the case of linearly moving objects, whereas we assume arbitrary and unknown motion patterns.

Recently, motivated by the need to find the best locations for placing k facilities, the database community has proposed algorithms for computing k-medoids [MPP07], and adding centers to an existing k-means set [ZDXT06]. These methods address snapshot queries on datasets indexed by disk-based, spatial access methods. Since they do not include modules for incremental computation of results, they are not suitable for continuous monitoring. Finally, there exist several systems aimed at minimizing the processing cost for continuous monitoring of spatial queries, such as ranges [MXA04] and nearest neighbors [XMA05, YPK05]. Other systems [MPBT05, HXL05] minimize the network overhead, but currently there is no method that optimizes both factors. Moreover, as discussed in Section 1, k-means computation is inherently more complex and challenging than continuous monitoring of conventional spatial queries. In the sequel, we propose a comprehensive framework, based on a solid mathematical background and including efficient processing algorithms.

3 Threshold-based *k*-Means Monitoring

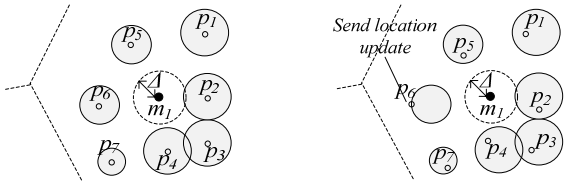
Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n moving points in the d-dimensional space. We represent the location of an object p_i ($1 \le i \le n$) at timestamp τ as $p_i(\tau) = \{p_i(\tau)[1], p_i(\tau)[2], \ldots, p_i(\tau)[d]\}$, where $p_i(\tau)[j]$ ($1 \le j \le d$) denotes the coordinate of p_i on the j-th axis. When τ is clear from the context, we simply use p_i to signify the object's position. The system does not rely on any assumption about the objects' moving patterns[2]. A central server collects the objects' locations and maintains a k-means set $M = \{m_1, m_2, \ldots, m_k\}$, where each m_i ($1 \le i \le k$) is a d-dimensional point, called the i-th *cluster center*, or simply *center*. We use $m_i(\tau)$ to denote the location of m_i at timestamp τ, and $M(\tau)$ for the entire k-means set at τ. The *quality* of $M(\tau)$ is measured by the function $cost(M(\tau)) = \sum_p dist(p(\tau), m_p(\tau))^2$, where $m_p(\tau)$ is the center assigned to $p(\tau)$ at τ, and $dist(p(\tau),m_p(\tau))$ is their Euclidean distance.

[2] The performance can be improved if each object's average speed is known. We discuss this issue in Section 4.3.

The two objectives of TKM are *effectiveness* and *efficiency*. Effectiveness refers to the quality of the k-means set, while efficiency refers to the computation and network overhead. These two goals may contradict each other, as intuitively, it is more expensive to obtain better results. We assume that the user specifies a *quality tolerance* Δ, and the system optimizes efficiency while always satisfying Δ. Δ is defined based on the *reference solution* (REF) discussed in Section 1. Specifically, let $M^{REF}(\tau)$ ($M^{TKM}(\tau)$) be the k-means set maintained by REF (TKM) at τ. At any time instant τ and for all $1 \le i \le k$, $dist(m_i^{REF}(\tau), m_i^{TKM}(\tau)) \le \Delta$, where $m_i^{REF}(\tau) \in M^{REF}(\tau)$ and $m_i^{TKM}(\tau) \in M^{TKM}(\tau)$; i.e., each center in $M^{TKM}(\tau)$ is within Δ distance of its counterpart in $M^{REF}(\tau)$. Given this property, we can prove [ZDT06] that $cost(M^{TKM}(\tau)) \le cost(M^{REF}(\tau)) + n\Delta^2$, thus providing the guarantee on the quality of M^{TKM}.

TKM works as follows. At τ=0, each object sends its location to the server, which computes the initial k-means set. Then, the server transmits to every object p_i a threshold θ_i, such that p_i sends a location update if and only if it deviates from its current position (i.e. $p_i[0]$) by at least θ_i. Alternatively, the server can broadcast certain statistical information, and each object computes its own θ_i's locally. Figure 3.1a shows an example, where p_1-p_7 start at positions $p_1(0)$-$p_7(0)$, and receive thresholds θ_1-θ_7 from the server, respectively. At timestamp 1, the objects move to the new locations shown in Figure 3.1b. According to REF, all 7 objects have to issue location updates, whereas in TKM, only p_6 informs the server since the other objects have not crossed their thresholds. When a new object appears, it sends its location to the server. When an existing point leaves the system, it also informs the server. In both cases, the message is processed as a location update.

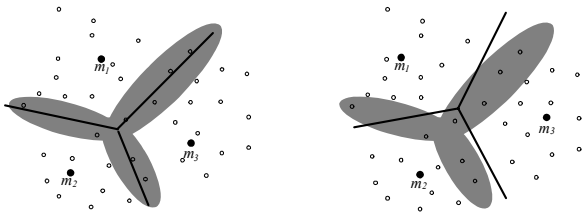
Upon receiving a location update from p_i at timestamp τ (in Figure 3.1b, from p_6), the server computes the new k-means using $p_i(\tau)$ and the last recorded positions of the other objects. Specifically, the server feeds the previous k-means set as seeds to an optimized version of HC, hereafter referred to as HC*. HC* computes exactly the same result as HC and performs the same iterations, but visits only a fraction of the dataset in each iteration.



(a) Timestamp 0 (b) Timestamp 1

Figure 3.1 Example update

HC* exploits the fact that the point assignment for the seed is usually similar to the converged result. As illustrated in Figure 3.2, only points close to the perpendicular bisectors (the shaded area), defined by the centers, may change clusters. Specifically, before an iteration starts, HC* estimates P_{active}, which is a super set of the points that may be re-assigned in the following iteration, and considers exclusively these points. Recall from Section 2.1 that an iteration consists of two steps: the first reassigns points to their respective nearest centers and the second computes the centroids of each cluster. HC* re-assigns only the points of P_{active} in the first step; whereas in the second step, for each cluster, HC* computes its new centroid based on the previous one and P_{active}. Specifically, for a cluster of points P_m, the centroid m_g is defined as $m_g[i] = \sum_{p \in P_m}(p[i] / |P_m|)$ for every dimension $1 \le i \le d$. HC* maintains an aggregate point SUM_m for each cluster such that $SUM_m[i] = \sum_{p \in P_m} p[i]$ $(1 \le i \le d)$. The centroid m_g can be thus computed using SUM_m and $|P_m|$. If in the previous step of the current iteration, one point p in P_{active} is reassigned from center m to m', HC* subtracts p from SUM_m and adds it to $SUM_{m'}$.



(a) Seeds (b) Converged result

Figure 3.2 Example of P_{active}

Figure 3.3 illustrates the pseudo-code of HC* and clarifies the computation of P_{active}. HC* initializes P_{active} with points that will change clusters in the first iteration (Lines 1-4), and after each iteration, it adds to P_{active} an estimated superset of the points that will change cluster in the next iteration (Line 17). Specifically, for each point p, it pre-computes: (i) d_p, which is the distance between p and its assigned center m_p in the seed set MS, and (ii) d'_p, the distance between p and its nearest center in $MS \setminus \{m_p\}$.

*HC** (*dataset* P, k, *Seed* MS)
1. For each point $p \in P$
2. Compute $d_p = dist(p, m_p)$, $d'_p = min_{m \in MS \setminus \{m_p\}} dist(p, m)$
3. Build a heap H on P in increasing order of $d'_p - d_p$
4. Compute set $P_{active} = \{p \mid d'_p - d_p < 0\}$ using H
5. For each center $m \in MS$
6. For each dimension i, $SUM_m[i] = \sum_{p \in P_m} p[i]$
7. Initialize $\sigma = 0$, $M = MS$
8. Repeat
9. For each point $p \in P_{active}$
10. Assign p to its nearest neighbor in M
11. If p is re-assigned from center m to m'
12. For each dimension i, adjust $SUM_m[i] = SUM_m[i] - p[i]$ and $SUM_{m'}[i] = SUM_{m'}[i] + p[i]$
13. For each center $m \in M$, corresponding to $m_s \in MS$
14. For each dimension i, $m_g[i] = SUM_m[i] / |P_m|$
15. Adjust $\sigma = max(\sigma, dist(m_s, m_g))$
16. Replace m with m_g
17. Adjust $P_{active} = \{p \mid d'_p - d_p < 2\sigma\}$ using the heap H
18. Until no change happens in M

Figure 3.3 Algorithm HC*

Figure 3.4 illustrates an example. Points satisfying $d'_p < d_p$ are re-assigned in the first iteration. HC* finds such points using a heap H sorted in increasing order of $d'_p - d_p$ (Lines 3-4). During the iterations, HC* maintains a value σ, which is the maximum distance that a center m deviates its original position $m_s \in MS$ in all previous iterations. After finishing an iteration, only points satisfying the

property $d'_p - d_p < 2\sigma$ may change cluster in the next one, because in the worst case, $m_p \in MS$ moves σ distance away from p and another center moves σ distance closer, as illustrated in Figure 3.4. HC* extracts such points from the heap H as the new P_{active}. Since the minimum element in the heap can be extracted in $O(\log n)$ time, the total construction cost of P_{active} is small.

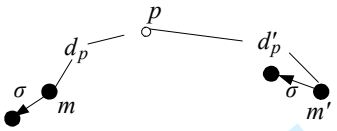


Figure 3.4 Illustration of d_p and d'_p

After HC* converges, TKM computes the new thresholds. Most thresholds remain the same, and the server only needs to send messages to a fraction of the objects. An object may discover that it has already incurred a violation, if its new threshold is smaller than the previous one. In this case, it sends a location update to the server. In the worst case, all objects have to issue updates, but this situation is rare. The main complication in TKM is how to compute the thresholds so that the guarantee on the result is maintained, and at the same time, the average update frequency of the objects is minimized. We discuss these issues in the following section.

4 Threshold Assignments

Section 4.1 derives a mathematical formulation of the threshold assignment problem. Section 4.2 proposes an effective algorithm for threshold computation. Section 4.3 integrates the objects' speed in the threshold computation, assuming that this knowledge is available. Section 4.4 discusses alternative methods for threshold dissemination.

4.1 Mathematical Formulation of Thresholds

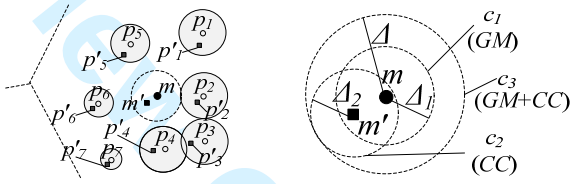
The threshold assignment routine takes as input the objects' locations $P = \{p_1, p_2, \ldots, p_n\}$ and the k-means set $M = \{m_1, m_2, \ldots, m_k\}$, and outputs a set of n real values $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, i.e. the thresholds. We formulate this task into a constrained optimization problem, where the objective is to minimize the average update frequency of the objects subject to the user's tolerance Δ. We first derive the objective function. Without any knowledge of the motion patterns, we assume that the average time interval between two consecutive location updates of each object p_i is proportional to θ_i. Intuitively, the larger the threshold, the longer the object can move in an arbitrary direction without violating it. Considering that all objects have equal weights, the (minimization) objective function is:

$$\sum_{i=1}^{n} 1/\theta_i \qquad 4.1$$

Next we formulate the requirement that no center in M^{TKM} deviates from its counterpart in M^{REF} by more than Δ. Note that this requirement can always be satisfied by setting $\theta_i = 0$ for all $1 \le i \le n$, in which case TKM reduces to REF. If $\theta_i > 0$, TKM re-computes M only when there is a violation of θ_i. Accordingly, when all objects are within their thresholds, the execution of HC*[3] should not cause any center to shift from its original position by more than Δ.

As shown in Section 2.1, the result of HC* can be predicted in the form of confinement circles. Compared to the static case [ZDT06], the problem of deriving confinement circles in our settings is further complicated by the fact that the exact locations of the objects are unknown. Consequently, for each cluster of points P_m assigned to center $m \in M$, m is not necessarily the centroid of P_m, since the points may have moved inside their confinement circles. In the example of Figure 4.1a, m is the assigned center for points p_1-p_7, computed by HC* on the points' original locations (denoted as dots). After some time, the points are located at p'_1-p'_7 shown as solid squares. Their new centroid is m', rather than m.

On the other hand, ZDT relies on the assumption that each center is the centroid of its assigned points. To bridge this gap, we split the tolerance Δ into $\Delta = \Delta_1 + \Delta_2$, and specify two constraints called the *geometric-mean* (GM), and the *confinement-circle* (CC) constraint. GM requires that for each cluster centered at $m \in M$, the centroid m' must be within Δ_1-distance from m, while CC demands that starting from each centroid m', the derived confinement circle must have a radius no larger than Δ_2. Figure 4.1b visualizes these constraints: according to GM, m' should be inside circle c_1, whereas according to CC, the confinement circle of m', and thus the center in the converged result, should be within c_2. Combining GM and CC, the center in the converged result must be within circle c_3, which expresses the user's tolerance Δ.



(a) Centroid m' (b) Constraints

Figure 4.1 Dividing Δ into Δ_1 and Δ_2

We now formulate GM and CC. For GM, the largest displacement occurs when all points move in the same direction, in which case the deviation of the centroid is the average of the points' displacement. Let $|P_m|$ be the cardinality of data points assigned to m, and θ_p be the threshold of p. GM requires that:

$$\forall m, \frac{1}{|P_m|} \sum_{p \in P_m} \theta_p \le \Delta_1 \qquad 4.2$$

Note that Inequality 4.2 expresses k constraints, one for each cluster.

The formulation of CC follows the general idea of ZDT, except that we do not know the objects' current positions, or their centroids. In short, we are trying to predict the output of HC* without having the exact in-

[3] Since HC* performs exactly the same iterations as HC, the results of [ZDT06] apply directly to HC*.

put. Let p'_i be the precise location of p_i, and $P'=\{p'_1, p'_2, \ldots, p'_n\}$. $M' = \{m'_1, m'_2, \ldots m'_k\}$, where m'_i is the centroid of all points assigned to $m_i \in M$. Similar to Section 2.1, $m'_{p'}$ denotes the assigned center for a point $p' \in P'$; $P_{m'}$ is the set of points assigned to center $m' \in M'$. By applying ZDT on the input P' and M', and using Δ_2 as the radius of the confinement circles, we obtain:

$$LB(M_b) > cost(M')$$

$$\text{where } LB(M_b) = \sum_{p' \in P'} dist^2(p', m_{p'}) + min_{m'}(|P'_{m'}| \Delta_2^2) - \sum_{p' \in P'_{move}} Y(p')$$

$$\text{and } Y(p') = \left(dist(p', m'_{p'}) + \Delta_2\right)^2 - \left(max\left(0, min_{m' \in M' \setminus \{m'_{p'}\}} dist(p', m') - \Delta_2\right)\right)^2 \quad 4.3$$

$$\text{and } P'_{move} = \{p' \mid Y(p') > 0\}$$

Inequality 4.3 is equivalent to 2.1, except for their inputs (P, M in 2.1; P', M' in 4.3). We apply known values (i.e. P, M) to re-formulate Inequality 4.3. First, it is impossible to compute $cost(M')$ with respect to P' since both P' and M' are unknown. Instead, we use $\sum_{p' \in P'} dist^2(p', m'_{p'})$ as an upper bound on $cost(M')$; i.e., every point $p' \in P'$ is assigned to $m'_{p'}$, which is not necessarily the nearest center for p'. Comparing $LB(M_b)$ and $cost(M')$, we reduce the inequality $LB(M_b) > cost(M')$ to:

$$\sum_{p' \in P'_{move}} Y(p') \leq min_{m'}(|P_{m'}| \Delta_2^2) \quad 4.4$$

The right side of Inequality 4.4, is exactly $min_m(|P_m| \Delta_2^2)$, since the point assignment in M' is the same as in M. It remains to re-formulate the function $Y(p)$ with known values. Recall from Section 2.1 that $Y(p)$ is an upper bound of the maximum cost reduction by shifting point p to another cluster. We thus derive an upper bound $UB\_Y(p)$ of $Y(p)$ using the upper bound and lower bound of $dist(p', m')$ for a point/center pair p' and m'. Because each $p' \in P'$ is always within θ_p distance to its corresponding point $p \in P$, and each center $m' \in M'$ is within Δ_1 distance to its corresponding center $m \in M$, we have:

$$\forall p' \in P', \forall m' \in M', |dist(p', m') - dist(p, m)| \leq \Delta_1 + \theta_p \quad 4.5$$

Using Inequality 4.5, we obtain $UB\_Y(p)$ in Equation 4.6, where $\Delta = \Delta_1 + \Delta_2$:

$$UB\_Y(p) = \left(dist(p, m_p) + \Delta + \theta_p\right)^2 - \left(max\left(0, min_{m \in M \setminus \{m_p\}} dist(p, m) - \Delta - \theta_p\right)\right)^2 \quad 4.6$$

Summarizing, we formulate constraint CC into:

$$\sum_{p \in P_{move}} UB\_Y(p) \leq min_m(|P_m| \Delta_2^2)$$

$$\text{where } UB\_Y(p) = \left(dist(p, m_p) + \Delta + \theta_p\right)^2 - \left(max\left(0, min_{m \in M \setminus \{m_p\}} dist(p, m) - \Delta - \theta_p\right)\right)^2 \quad 4.7$$

$$\text{and } P_{move} = \{p \mid UB\_Y(p) > 0\}$$

Thus, the optimization problem for threshold assignment has the objective function expressed by equation (4.1), subject to the GM (Inequality 4.2) and CC (Inequality 4.7) constraints. However, the quadratic and non-differentiable nature of Inequality 4.7 renders an efficient solution for the optimal values of θ_i infeasible. Instead, in the next section we propose a simple and effective algorithm.

4.2 Computation of Thresholds

In order to derive a solution, we simplify the problem by fixing Δ_1 and Δ_2 as constants. The ratio of Δ_1 and Δ_2 is chosen by the server and adjusted dynamically, but we defer the discussion of this issue until the end of the subsection. We adopt a two-step approach: in the first step, we ignore constraint CC, and compute a set of thresholds using only Δ_1; in the second step, we shrink thresholds that are restricted by CC. The problem of the first step is solved by the *Lagrange multiplier method* [AW95]. The optimal solution is:

$$\theta_1 = \theta_2 = \ldots = \theta_n = \Delta_1 \quad 4.8$$

Note that all thresholds have the same value because both the objective function (4.1) and constraint GM (4.2) are symmetric. We set each $\theta_i = \Delta_1$, and use these values to compute set P_{move}. In the next step, when we decrease the threshold θ_p of a point p, P_{move} may change accordingly. We first assume that P_{move} is fixed, and later propose solutions when this assumption does not hold.

Regarding the second step, according to Inequality 4.7, constraint CC only restricts the thresholds of points $p \in P_{move}$. Therefore, to satisfy CC, it suffices to decrease the thresholds of those points. Next we study UB_Y in Inequality 4.7. Note that UB_Y is a function of threshold θ_p, associated with point p. We observe that UB_Y is in the form of $(A+\theta_p)^2 - (max(0,(B-\theta_p))^2)$, where A, B are constants with respect to θ_p. Hence, as long as $B - \theta_p \geq 0$, UB_Y is linear to θ_p because the two quadratic terms cancel out; on the other hand, when $B - \theta_p < 0$, UB_Y becomes quadratic to θ_p. The turning point where UB_Y changes from linear to quadratic is the only non-differentiable point of UB_Y. Therefore, we simplify the problem by making $B - \theta_p \geq 0$ a hard constraint, which ensures that UB_Y is always differentiable and linear to θ_p. Specifically, we have the following constraint:

$$\theta_p \leq min_{m \in M \setminus \{m_p\}} dist(p, m) - \Delta \quad 4.9$$

We then decrease the affected threshold values accordingly[4]. Inequality 4.9 enables us to transform constraint CC (Inequality 4.7) to:

$$\sum_{p \in P_{move}} UB\_Y(p) \leq min_m(|P_m| \Delta_2^2)$$

$$\text{where } UB\_Y(p) = 2\left(dist(p, m_p) + min_{m \in M \setminus \{m_p\}} dist(p, m)\right)\theta_p + \left(dist(p, m_p) + \Delta\right)^2 - \left(min_{m \in M \setminus \{m_p\}} dist(p, m) - \Delta\right)^2 \quad 4.10$$

Note that P_{move} is already determined in the previous step. Solving the optimization problem (again using *Lagrange multiplier*) with (4.1) as the objective function and (4.10) as the only constraint, the optimal solution is:

[4] Inequality 4.9 may require $\theta_p < 0$. If this pathological case occurs, we set θ_p to a pre-defined value, treat it as a constant, and solve the optimization problem with one less variable.

$$\forall p \in P_{move}, \theta_p^* = \frac{\sqrt{d(p)+d'(p)}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{d(q)+d'(q)}} \times\left(min_m\left(|P_m|\Delta_2^2\right)-\sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2-\left(d'(q)-\Delta\right)^2\right) \quad 4.11$$

where $d(p)=dist(p,m_p), d'(p)=min_{m\in M\setminus\{m_p\}}dist(p,m)$

After that, for each threshold θ_p of point $p\in P_{move}$ whose current value is above θ_p^*, we decrease it to θ_p^*. In rare cases, this decrease causes $UB\_Y(p)\le 0$, meaning that p should not be a member of P_{move}. When this happens, we remove p from P_{move}, and repeat step 2 until P_{move} is stable, which signifies that CC is satisfied. In practice, this procedure usually converges very fast. Because we never increase any threshold value, GM is still satisfied, thus we now have a feasible solution to the original problem. The algorithm for threshold computation is summarized in Figure 4.2.

Compute_Thresholds(*Dataset P*, *Center set M*)
1. Set all θ_i ($1\le i\le n$) to Δ_1 // step 1
2. Compute P_{move} using the current θ_i's
3. For each point $p \in P_{move}$ // step 2
4. Decrease θ_p so that it satisfies Inequality 4.9
5. Let θ_p^* be results of evaluating Equation 4.11
6. If $\theta_p^* < \theta_p$, $\theta_p = \theta_p^*$
7. For each point $p \in P_{move}$
8. If $UB\_Y(p) \le 0$, Remove p from P_{move}
9. If P_{move} has changed, Goto Line 3 // repeat step 2

Figure 4.2 Algorithm for computing thresholds

Finally we discuss the choice of Δ_1 and Δ_2. Let $a = \Delta_1/\Delta$. Initially, TKM chooses a according to past experience, e.g. 0.7 in our experiments. After the system starts, TKM examines periodically whether it can improve performance by adjusting a. Specifically, using the routine for computing thresholds described above, we express the objective function (4.1) as a function of a and analyze the first order derivative of it. Let P_{GM} be the set of points assigned threshold Δ_1 in the routine described above, and $P_{CC} = P\setminus P_{GM}$. Then, the objective function is expressed as:

$$obj(\alpha)=\sum_{p\in P_{GM}}\frac{1}{\alpha\Delta}+\sum_{p\in P_{CC}}\frac{1}{\theta_p^*}$$
$$\text{where } \theta_p^*=\frac{\sqrt{d(p)+d'(p)}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{d(q)+d'(q)}} \quad 4.12$$
$$\times\left(min_m\left(|P_m|\right)(1-\alpha)^2\Delta^2-\sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2-\left(d'(q)-\Delta\right)^2\right)$$
$$\text{and } d(p)=dist(p,m_p), d'(p)=min_{m\in M\setminus\{m_p\}}dist(p,m)$$

The derivative $obj'(a)$ is computed during the assignment process. If the absolute value of $obj'(a)$ is above a predefined threshold, TKM adjusts a by adding (subtracting) a step value ε to (from) it, depending on whether $obj'(a) < 0$. In our experiments, we simply set $\varepsilon = 0.05a$. Alternatively, ε can be a function of $obj'(a)$.

4.3 Utilizing the Object Speed

In practice, the speed of moving users is restricted by their means of transportation (e.g., pedestrians vs. cars etc). In this section, we assume that the system knows roughly the average speed s_i of each object p_i ($1\le i\le n$). Intuitively, fast objects are assigned larger thresholds, so that they do not need to issue updates in the near future. In addition to location updates, p_i sends a *speed update* to the server when its speed changes significantly, e.g. a pedestrian mounts a motor vehicle. We first re-formulate the constrained optimization problem to incorporate the speed information. The two constraints GM and CC remain the same, but the objective function (4.1) becomes:

$$\sum_{i=1}^{n} s_i/\theta_i \quad 4.13$$

The rationale is that when p_i moves constantly in one direction, it is expected to violate the threshold after θ_i/s_i timestamps. Next we compute the thresholds using the approach of Section 4.2. In the first step, we solve the optimization problem with (4.13) as the objective and (4.2) as the only constraint. Using the Lagrange multiplier method, we obtain the optimal solution:

$$\forall i, \theta_i = n\Delta_1\sqrt{s_i}\Big/\sum_{j=1}^{n}\sqrt{s_j} \quad 4.14$$

Comparing Equation 4.8 with 4.14, the former gives equal thresholds to all objects, while the latter assigns thresholds proportional to the square root of the objects' speeds. In the second step, we solve the problem with (4.13) as the objective function and (4.10) as the only constraint. Let s_p be the speed of object p, the optimal solution is:

$$\forall p \in P_{move}, \theta_p^* = \frac{\sqrt{\left(d(p)+d'(p)\right)s_p}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{\left(d(q)+d'(q)\right)s_q}} \times\left(min_m\left(|P_m|\Delta_2^2\right)-\sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2-\left(d'(q)-\Delta\right)^2\right) \quad 4.15$$
$$\text{where } d(p)=dist(p,m_p), d'(p)=min_{m\in M\setminus\{m_p\}}dist(p,m)$$

The threshold routine of Figure 4.2 is modified accordingly, by substituting Equation 4.8 with Equation 4.14, and Equation 4.11 with Equation 4.15.

4.4 Dissemination of Thresholds

After computing the thresholds, the server needs to disseminate them. We propose two approaches, depending on the computational capabilities of the objects. The first is based on broadcasting, motivated by the fact that the network overhead of broadcasting is significantly smaller than sending individual messages to all moving objects, and does not increase with the dataset cardinality. Initially, the server broadcasts Δ. After the first k-means set is computed, the broadcast information includes the center set M, Δ_1 (if it has changed with respect to the previous broadcast), the two sum values in Equation 4.11, and $min_m\ |P_m|$. Each object computes its own threshold based on the broadcast information.

The second approach assumes that objects have limited computational capabilities. Initially, the server sends Δ_1 to all objects through single-cast messages. In subsequent updates, it sends the threshold to an object only when it has changed. Note that most objects, besides those in P_{move}, have identical threshold Δ_1. As we show experimentally in Section 5, usually P_{move} is only a small

(a) *spatial* (b) *road*

Figure 5.1 Illustration of the datasets

fraction of P. Therefore, the overhead of sending these the thresholds is not large. Alternatively, we propose a variant of TKM that sends messages only to objects that issue location updates. Let P_u be the set of objects that have issued updates, and Θ_{old} be the set of thresholds before processing these updates. After re-evaluating the k-means, the server computes the set of thresholds Θ_{new} for all objects. Then, it constructs the threshold set Θ as follows. For each point in P_u, its threshold in Θ is the same as in Θ_{new}, whereas all other thresholds remain the same as in Θ_{old}. After that, the sever checks whether Θ satisfies the two constraints GM and CC. If so, it simply disseminates Θ to P_u. Otherwise, it modifies the constrained optimization problem, by treating only the thresholds of points in P_u as variables, and all other thresholds as constants whose values are from Θ_{old}. Using the two-step approach of Section 4.2, this version of TKM obtains the optimal solution Θ'_{new}. If Θ'_{new} is *valid*, i.e. each threshold is non-negative, the server disseminates Θ'_{new} to objects in P_u; otherwise, it disseminates Θ_{new} to all affected objects.

5 Experimental Evaluation

This section compares TKM against REF using two datasets. In the first one, denoted as *spatial*, we randomly select the initial position and the destination of each object from a real dataset, California Roads (available at *www.rtreeportal.org*) illustrated in Figure 5.1a. Each object follows a linear trajectory between the two points. Upon reaching the endpoint, a new random destination is selected and the same process is repeated. The second dataset, denoted as *road* [MYPM06], is based on the generator of [B02] using sub-networks of the San Francisco road map (illustrated in Figure 5.1b) with about 10K edges. Specifically, an object appears on a network node, completes a shortest path to a random destination and then disappears. To ensure that the data cardinality n (a parameter) remains constant, whenever an object disappears, a new one enters the system. In both datasets, the coordinates are normalized to the range [0..1] on each dimension, and every object covers distance 1/1000 per axis at each timestamp. In *spatial*, movement is unrestricted, whereas in *road*, objects are restricted to move on the network edges.

REF employs HC to re-compute the k-means set, whereas TKM utilizes HC*. Since every object moves at each timestamp, there are threshold violations in most timestamps, especially for small tolerance values. This implies that TKM usually resorts to re-computation from scratch and the CPU gains with respect to REF are mostly due to HC*. For the dissemination of thresholds we use the single-cast protocol, where the server informs each object individually about its threshold. Table 5.1 summarizes the parameters under investigation. The default (median) values are typeset in boldface. In each experiment we vary a single parameter, while setting the remaining ones to their median values. The reported results represent the average value over 10 simulations. For each simulation, we monitor the k-means set for 100 timestamps[5]. We measure: (i) the overall CPU time at the server for all timestamps, (ii) the total number of messages exchanged between the server and the objects, and (iii) the quality of the solutions.

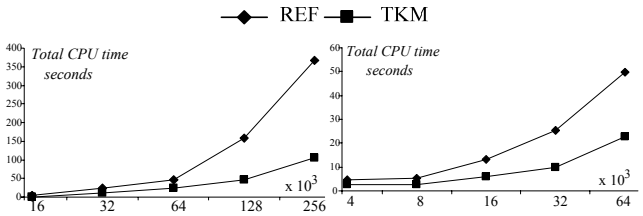
Table 5.1 Experimental parameters

Parameter	*Spatial*	*Road*
Cardinality n ($\times 10^3$)	16, 32, **64**, 128, 256	4, 8, **16**, 32, 64
Number of means k	2, 4, 8, **16**, 32, 64	2, 4, 8, **16**, 32, 64
Tolerance Δ	0.0125, 0.025, **0.05**, 0.1, 0.2	0.05, 0.1, **0.2**, 0.4, 0.8

Figure 5.2 evaluates the effect of the object cardinality n on the computation cost for the *spatial* and *road* datasets. The CPU overhead of REF is dominated by re-computing the k-means set using HC at every timestamp, whereas the cost of TKM consists of both k-means computations with HC*, and threshold evaluations. The results show that TKM consistently outperforms REF, and the performance gap increases with n. This confirms both the superiority of HC* over HC and the efficiency of the threshold computation algorithm, which increases the CPU overhead only marginally. An important observa-

[5] Due to the huge space and time requirements of simulations in large network topologies, we use smaller values for n in *road*.

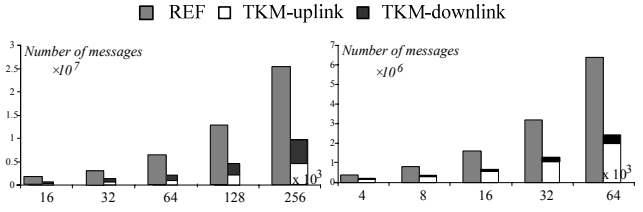
tion is that TKM scales well with the object cardinality (note that the x-axis is in logarithmic scale).



(a) *spatial* (b) *road*

Figure 5.2 CPU time versus data cardinality n

Figure 5.3 illustrates the total number of messages as a function of n. In REF all messages are *uplink*, i.e., location updates from objects to the server. TKM includes also *downlink* messages, by which the server informs the objects about their new thresholds. We assume that the costs of uplink and downlink messages are equal[6]. In both datasets, TKM achieves more than 60% reduction on the overall communication overhead. Specifically, TKM incurs significantly fewer uplink messages since an object does not update its location while it remains inside its threshold. On the other hand, the number of downlink messages in TKM never exceeds the number of uplink messages, which is ensured by the threshold dissemination algorithm. Interestingly, the number of downlink messages in the *road* dataset is smaller than in *spatial*. This is because TKM only updates the thresholds of the objects lying close to the boundary of clusters (i.e. those in P_{move}). In *road*, objects' movements are restricted by the road network, leading to highly skewed distributions. Consequently, the majority of objects lie close their respective centers. In *spatial*, however, the distribution of the objects is more uniform; therefore, a large number of objects lie on the boundary of the clusters.



(a) *spatial* (b) *road*

Figure 5.3 Number of messages versus data cardinality n

Having established the performance gain of TKM with respect to REF, we evaluate its effectiveness. Figure 5.4 depicts the average squared distance achieved by the two methods versus n. The column *MaxDiff* corresponds to the maximum cost difference between TKM and REF in any timestamp. Clearly, the average cost achieved by TKM is almost identical to that of REF. Moreover, *MaxDiff* is negligible compared to the average costs, and grows linearly with n, since it is bounded by $n\Delta^2$ as described in Section 3.1.

[6] In practice uplink messages are more expensive in terms of battery consumption (i.e., the battery consumption is 2-3 times higher in the sending than the receiving mode [DVCK99]). This distinction favors TKM.

n ($\times 10^3$)	TKM	REF	MaxDiff
16	142.23	142.15	0.18
32	289.58	289.40	0.34
64	566.25	566.14	0.70
128	1139.87	1139.64	1.40
256	2302.60	2302.22	2.82

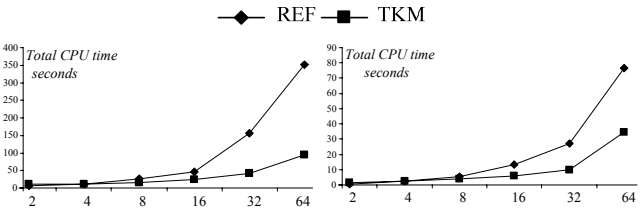
(a) *spatial*

n ($\times 10^3$)	TKM	REF	MaxDiff
4	19.67	19.46	1.012
8	37.78	37.69	1.050
16	76.07	76.05	1.849
32	151.09	150.73	2.913
64	303.21	302.69	3.884

(b) *road*

Figure 5.4 Quality versus data cardinality n

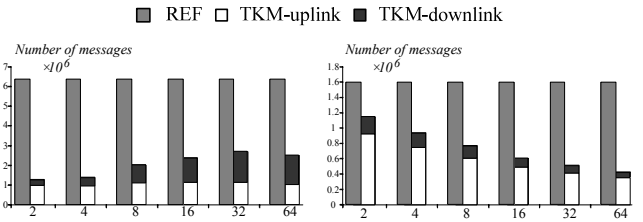
Next we investigate the impact of the number of means k. Figure 5.5 plots the CPU overhead as a function of k. TKM outperforms REF on all values of k, although the difference for small k is not clearly visible due to the high cost of large sets. The advantage of TKM increases with k because the first step of HC has to find the nearest center of every point, a process that involves k distance computations per point. On the other hand, HC* only considers points near the cluster boundaries, therefore saving unnecessary distance computations.



(a) *spatial* (b) *road*

Figure 5.5 CPU time versus number of means k

Figure 5.6 studies the effect of k on the communication cost. While consistently better than REF, TKM behaves differently in the two datasets. In *spatial*, the number of uplink messages is stable, and the downlink messages increase with k. This is because the objects are more uniformly distributed, and a larger k causes an increased boundary area, meaning that more points need to update their thresholds. On the other hand, in the *road* dataset, the numbers of both uplink and downlink messages decline with the increase of k. The dominating factor in *road* is the high degree of skewness of the objects. A larger k fits this distribution better because more dense areas are populated with centers. Consequently, the boundary area decreases, leading to fewer downlink messages. The number of uplink messages also decreases because the assigned thresholds are larger, since the objects are closer to their respective centers.



(a) *spatial* (b) *road*

Figure 5.6 Number of messages vs. number of means k

Figure 5.7 measures the result quality as a function of k. The average k-means cost of TKM and REF are very close for all values of k. In terms of the maximum cost differ-

ence, the two dataset exhibit different characteristics. The result of *spatial* is insensitive to k, whereas in *road*, the maximum difference increases with k. The reason is that since the objects are skewed in *road*, a slight deviation of a center causes a large increase in cost. This effect is more pronounced with large k, since the centers fit the distribution better. Nevertheless, the quality guarantee $n\Delta^2$ is always kept.

k	TKM	REF	MaxDiff
2	5552.35	5552.20	1.080
4	2231.06	2230.87	1.440
8	1156.37	1156.68	1.010
16	566.25	566.14	1.565
32	286.58	286.57	1.373
64	143.78	143.64	1.362

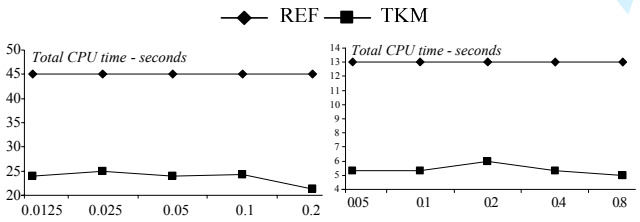
(a) *spatial*

k	TKM	REF	MaxDiff
2	678.97	678.94	0.36
4	326.25	326.23	0.15
8	166.11	166.06	0.70
16	76.07	76.05	1.85
32	36.98	36.95	2.47
64	18.09	17.67	2.85

(b) *road*

Figure 5.7 Quality versus number of means k

The last set of experiments studies the effect of the user's quality tolerance Δ. Because REF does not involve Δ, its results are constant in all experiments, and we focus mainly on TKM. Figure 5.8 demonstrates the effect of Δ on the CPU cost, where TKM is the clear winner for all settings. Specifically, its CPU time is relatively stable with respect to Δ, except for very large values. In these cases, the CPU time of TKM drops because at some timestamps, no objects issue location updates (and the server does not need to perform any computation).

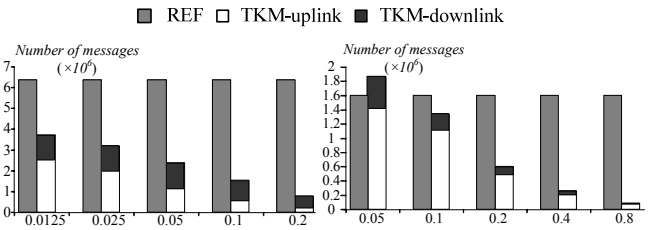


(a) *spatial* (b) *road*

Figure 5.8 CPU time versus tolerance Δ

Figure 5.9 shows the impact of Δ on the communication cost. The numbers of both uplink and downlink messages drop as Δ increases, with the former dropping faster than the latter. Naturally, the number of uplink messages decreases because the update frequency is inversely proportional to the threshold value. The downlink messages drop for two reasons. First, in the threshold computation procedure, the value of each threshold θ^* given by Equation 4.11 increases as Δ grows. According to the algorithm, when θ^* exceeds Δ_1, the object's threshold remains Δ_1. Consequently, when Δ is large, many objects in P_{move} retain Δ_1, and do not need to be updated by a downlink message. The second reason is that for large Δ, at some timestamps there are no location updates, and therefore, no threshold dissemination.

Finally, Figure 5.10 investigates the effect of Δ on the result quality. As expected, in both datasets the result quality drops with the larger Δ. This is more pronounced in *road* due to the more skewed distribution. Observe that even when Δ is very large (reaching 80% of the axis length in *road*), TKM is still competitive in terms of quality.



(a) *spatial* (b) *road*

Figure 5.9 Number of messages versus tolerance Δ

Δ	TKM	REF	MaxDiff
0.0125	566.21	566.14	1.212
0.025	566.21	566.14	1.272
0.05	566.25	566.14	1.565
0.1	566.54	566.14	1.722
0.2	569.39	566.14	8.535

(a) *spatial*

Δ	TKM	REF	MaxDiff
0.05	75.41	76.05	0.072
0.1	75.52	76.05	0.651
0.2	76.07	76.05	1.849
0.4	76.87	76.05	7.535
0.8	81.56	76.05	13.829

(b) *road*

Figure 5.10 Quality versus tolerance Δ

Summarizing the experimental evaluation, TKM outperforms REF by a wide margin on both CPU cost and communication overhead, while it incurs negligible deterioration of the result quality. Therefore, it allows monitoring of k-means in very large datasets of continuously moving objects. Moreover, TKM scales better than REF on the number of means, implying that it can be used in applications that require high values of k.

6 Conclusions

This paper proposes TKM, the first approach for continuous k-means computation over moving objects. Compared to the simple solution of re-evaluating k-means for every object update, TKM achieves considerable savings by assigning each object a threshold, such that the object needs to inform the server only when there is a threshold violation. We present mathematical formulae and an efficient algorithm for threshold computation. In addition, we develop an optimized hill climbing technique for reducing the CPU cost, and discuss optimizations of TKM for the case that object speeds are known. Finally, we design different threshold dissemination protocols depending on the computational capabilities of the objects.

In the future, we plan to extend the proposed techniques to related problems. For instance, k-medoids are similar to k-means, but the centers are restricted to points in the dataset. TKM could be used to find the k-means set, and then replace each center with the closest data point. It would be interesting to study performance guarantees (if any) in this case, as well as devise adaptations of TKM for the problem. Finally, another direction concerns distributed monitoring of k-means. In this scenario, there exist multiple servers maintaining the locations of distinct sets of objects. The goal is to continuously compute the k-means using the minimum amount of communication between servers.

REFERENCES

[AV06] Arthur, D., Vassilvitskii, S. How Slow is the *k*-means Method. *SoCG*, 2006.

[AW95] Arfken, G., Weber, H. *Mathematical Methods for Physicists*. Academic Press, 1995.

[B02] Brinkhoff, T. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2): 153-180, 2002.

[BDMO03] Babcock, B., Datar, M., Motwani, R., O'Callaghan, L. Maintaining Variance and *k*-Means over Data Stream Windows. *PODS*, 2003.

[BF98] Bradley, P., Fayyad, U. Refining Initial Points for *k*-Means Clustering. *ICML*, 1998.

[DVCK99] Datta, A., Vandermeer, D., Celik, A., Kumar, V. Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users. *ACM TODS*, 24(1): 1-79, 1999.

[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.

[HS05] Har-Peled, S., Sadri, B. How Fast is the *k*-means Method. *SODA*, 2005.

[HXL05] Hu, H., Xu, J., Lee, D. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. *SIGMOD*, 2005.

[IKI94] Inaba, M., Katoh, N., Imai, H. Applications of Weighted Voronoi Diagrams and Randomization to Variance-based Clustering. *SoCG*, 1994.

[JMF99] Jain, A., Murty, M., Flynn, P. Data Clustering: a Review. *ACM Computing Surveys*, 31(3): 264-323, 1999.

[KMB05] Kalnis, P., Mamoulis, N. Bakiras, S. On Discovering Moving Clusters in Spatio-temporal Data. *SSTD*, 2005.

[KMN+02] Kanungo, T., Mount, M., Netanyahu, N., Piatko, C., Silverman, R., Wu, A. An Efficient *k*-means Clustering Algorithm: Analysis and Implementation. *IEEE PAMI*, 24(7): 881-892, 2002.

[KR90] Kaufman, L., Rousseeuw, P. J. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[KSS04] Kumar, A., Sabharwal, Y., Sen, S. A Simple Linear Time (1+ε)-Approximation Algorithm for *k*-means Clustering in Any Dimensions. *FOCS*, 2004.

[L82] Lloyd, S. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2): 129-136, 1982.

[LHY04] Li, Y., Han, J., Yang, J. Clustering Moving Objects. *KDD*, 2004.

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

[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 Partitioning Querying: A Methodology for Computing Medoids in Large Spatial Datasets. *VLDB J.*, to appear.

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

[MYPM06] Mouratidis, K., Yiu, M., Papadias, D., Mamoulis, N. Continuous Nearest Neighbor Monitoring in Road Networks. *VLDB*, 2006.

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

[PM99] Pelleg, D., Moore, A. Accelerating Exact *k*-Means Algorithms with Geometric Reasoning. *KDD*, 1999.

[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.

[ZDT06] Zhang, Z., Dai, B., Tung, A. On the Lower Bound of Local Optimum in *k*-means Algorithm. *ICDM*, 2006.

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

Continuous *k*-Means Monitoring over Moving Objects

Zhenjie Zhang, Yin Yang, Anthony K.H. Tung, and Dimitris Papadias

Abstract— Given a dataset *P*, a *k*-means query returns *k* points in space (called *centers*), such that the average squared distance between each point in *P* and its nearest center is minimized. Since this problem is NP-hard, several approximate algorithms have been proposed and used in practice. In this paper, we study continuous *k*-means computation at a server that monitors a set of moving objects. Re-evaluating *k*-means every time there is an object update imposes a heavy burden on the server (for computing the centers from scratch) and the clients (for continuously sending location updates). We overcome these problems with a novel approach that significantly reduces the computation and communication costs, while guaranteeing that the quality of the solution, with respect to the re-evaluation approach, is bounded by a user-defined tolerance. The proposed method assigns each moving object a threshold (i.e., range) such that the object sends a location update only when it crosses the range boundary. First, we develop an efficient technique for maintaining the *k*-means. Then, we present mathematical formulae and algorithms for deriving the individual thresholds. Finally, we justify our performance claims with extensive experiments.

Revised submission to TKDE, February 2008

Index Terms—*k*-means, Continuous monitoring, Query processing

1 Introduction

Efficient *k*-means computation is crucial in many practical applications, such as clustering, facility location planning and spatial decision making. Given a dataset $P = \{p_1, p_2, \ldots p_n\}$ of 2D points, a *k*-means query returns a *center set* M of k points $\{m_1, m_2, \ldots, m_k\}$, such that $cost(M) = \sum_{i=1}^{n} dist^2(p_i, NN(p_i,M))$ is minimized, where for all i satisfying $1 \le i \le n$, $NN(p_i,M)$ is the nearest neighbor of p_i in M, and *dist* is a distance (usually, Euclidean) metric. The data points whose NN is $m_j \in M$ ($1 \le j \le k$) form the *cluster* of m_j. Since the problem is NP-hard [M06], the vast majority of existing work focuses on approximate solutions. In the data mining literature, several methods (e.g., [L82, HS05, AV06]) are based on *hill climbing* (HC). Specifically, HC starts with k random *seeds* as centers, and iteratively adjusts them until it reaches a *local optimum*, where no further improvement is possible. Then, it repeats this process, each time using a different set of seeds. The best local optimum among the several executions of HC constitutes the final result.

Recently, there is a paradigm shift from snapshot queries on static data to long-running queries over moving objects. In this scenario, mobile clients equipped with GPS devices send their locations to a central server. The server collects these locations and continuously updates the results of registered spatial queries. Our work focuses on *continuous k-means monitoring* over moving objects, which has numerous practical applications. For instance, in real-time traffic control systems, monitoring the *k*-means helps detect congestions, which tend to happen near the cluster centers of the vehicles [JLO07]. In a military campaign, the *k*-means of the positions of ground troops determine the best locations to airdrop critical supplies such as ammunitions and medical kits. Compared to conventional spatial queries (e.g. ranges or nearest neighbors), *k*-means monitoring is fundamentally more complex. The main reason is that in simple queries, the results are based solely on the objects' *individual* locations with respect to a single reference (i.e., query) point. On the other hand, a *k*-means set depends upon the *combined* properties of all objects. Even a slight movement by an object causes its corresponding center to change. Moreover, when the object is close to the boundary of two clusters, it may be assigned to a different center, and the update may have far-reaching effects.

A simple method for continuous *k*-means monitoring, hereafter called REF (short for *reference* solution), works as follows. When the system starts at time $\tau=0$, every object reports its location, and the server computes the *k*-means set $M(0)$ through the HC algorithm. Subsequently ($\tau>0$), whenever an object moves, it sends a location update. The server obtains $M(\tau)$ by executing HC on the updated locations, using $M(\tau\text{-}1)$ as the seeds. The rationale is that $M(\tau)$ is expected to be more similar to $M(\tau\text{-}1)$ than a random seed set, reducing the number of HC it-

- *Z. Zhang and A. Tung are with the Department of Computer Science, National University of Singapore, 117590 Singapore. Email: {zhangzh2, atung}@comp.nus.edu.sg*
- *Y. Yang and D. Papadias are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. Email: {yini,dimitris}@cse.ust.hk*

erations. REF produces high quality results because it continuously follows every object update. On the other hand, it incurs large communication and computation cost due to the frequent updates and re-computations.

To eliminate these problems, we propose a *threshold-based k-means monitoring* (TKM) method, based on the framework of Figure 1.1. In addition to k, a continuous k-means query specifies a *tolerance* Δ. The computation of $M(0)$ is the same as in REF. After the server computes a center set, it sends to every object p_i a *threshold* θ_i, such that p_i needs to issue an update, only if its current location deviates from the previous one by at least θ_i. When the server receives such an update, it obtains the new k-means using HC*, an optimized version of HC, over the last recorded locations of the other objects (which are still within their assigned thresholds). Then, it computes the new threshold values. Because most thresholds remain the same, the server only needs to send messages to a fraction of the objects. Let M^{REF} (M^{TKM}) be the k-means set maintained by REF (TKM). We prove that for any time instant and for all $1 \le i \le k$: $dist(m_i^{REF}, m_i^{TKM}) \le \Delta$, where $m_i^{REF} \in M^{REF}$ and $m_i^{TKM} \in M^{TKM}$; i.e., each center in M^{TKM} is within Δ distance of its counterpart in M^{REF}.

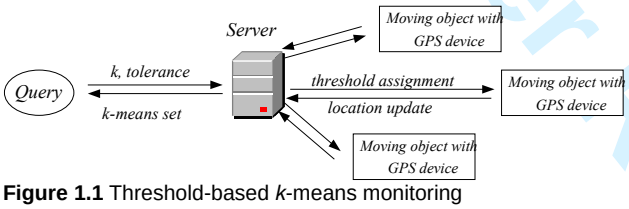


Figure 1.1 Threshold-based k-means monitoring

Our contributions are:

1. We present TKM, a general framework for continuous k-means monitoring over moving objects.
2. We propose HC*, an improved HC, which minimizes the cost of each iteration by only considering a small subset of the objects.
3. We model the threshold assignment task as a constrained optimization problem, and derive the corresponding mathematical formulae.
4. We develop an algorithm for the efficient computation of thresholds.
5. We design different mechanisms for the dissemination of thresholds, depending on the computational capabilities of the objects.

The rest of this paper is organized as follows: Section 2 overviews related work. Section 3 presents TKM and the optimized HC* algorithm. Section 4 focuses on the mathematical derivation of thresholds, proposes an algorithm for their efficient computation, and describes protocols for their dissemination. Section 5 contains a comprehensive set of experiments, and Section 6 concludes with directions for future work.

2 Background

Section 2.1 overviews k-means computation over static datasets. Section 2.2 surveys previous work on continuous monitoring and problems related to k-means.

2.1 *k*-Means Computation for Static Data

Numerous methods for computing k-means over static data have been proposed in the theoretical literature. Inaba et al. [IKI94] present an $O(n^{O(kd)})$ algorithm for optimal solutions and a $O(n(1/\varepsilon)^d)$ method for ε-approximate 2-means, where n is the data cardinality and d the dimensionality. Kumar et al. [KSS04] develop a $(1+\varepsilon)$-approximate algorithm for k-means in any Euclidean space, which is linear to n and d. Kanungo et al. [KMN+02] describe a swapping technique achieving $1+\varepsilon$-approximation. Several studies focus on the convergence speed of k-means algorithms [HS05, AV06].

Most data mining literature has applied hill climbing (HC) [L82] for solving k-means. Figure 2.1 shows a general version of the algorithm. HC starts with a set M that contains k random seeds, and iteratively improves it. Each iteration consists of two steps: the first (Lines 3-4) assigns every point to its nearest center in M, and the second (Lines 5-6) replaces each $m \in M$ with the *centroid* of its assigned points, which is the best location for minimizing the cost function (i.e., average squared distance) for the current point assignment [KSS04] [1]. HC *converges* at a *local optimum*, when there can be no further improvement on M. Since a local optimum is not necessarily the global optimum, usually several runs of the algorithm are executed with different sets of seeds. The best local optimum of these runs is returned as the final output. Although there is no guarantee on efficiency (a large number of iterations may be needed for convergence), or effectiveness (a local optimum may deviate significantly from the best solution), HC has been shown to perform well in practice.

HC (*dataset P*, *k*)
1. Choose k random seeds as the initial center set M
2. Repeat
3. For each point $p \in P$
4. Assign p to its nearest center in M
5. For each center m in M
6. Replace m with the centroid of all $p \in P$ assigned to m
7. Until no change happens in M

Figure 2.1 Hill climbing for computing k-means

Zhang et al. [ZDT06] propose a method, hereafter referred to as ZDT, for predicting the best possible cost that can be achieved by HC. We describe ZDT in detail because it is utilized by the proposed techniques. Let M be the current k-means set after one iteration of HC (i.e., after Line 6). At present, each center m in M is the centroid of its cluster. For a point p, we use m_p to denote the currently assigned center of p, which is not necessarily the nearest. A key observation is that there is a constant δ, such that in all subsequent HC iterations, no center can deviate from its position in M by more than δ. Equivalently, as shown in Figure 2.2, each $m \in M$ is restricted to a *confinement circle* centered at its current position with radius δ. Therefore, when HC eventually converges to a

[1] The centroid is computed by taking the average coordinates of all assigned points on each dimension.

local optimum M^*, each center in M^* must be within its corresponding confinement circle. Zhang et al. [ZDT06] prove that $cost(M)$-$n\delta^2$ is a lower bound of $cost(M^*)$, where n is the data cardinality. If this bound exceeds the best cost achieved by a previous local optimum (with a different seed set), subsequent iterations of HC can be pruned.

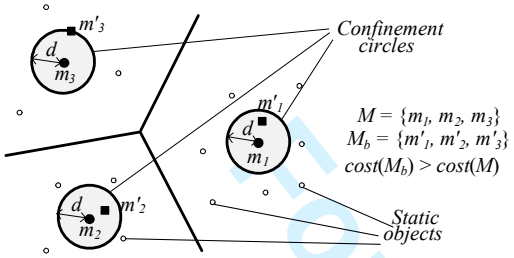


Figure 2.2 Centers restricted in confinement circles

It remains to clarify the computation of δ. Let M_b be a set constructed by moving one of the centers in M to the boundary of its confinement circle, and the rest within their respective confinement circles. In the example of Figure 2.2, M={m_1, m_2, m_3}, M_b={m'_1, m'_2, m'_3}, and m'_3 is on the boundary of the confinement circle of m_3. The defining characteristic of δ is that for *any* such M_b, $cost(M_b) > cost(M)$. We cannot compute δ directly from the inequality $cost(M_b) > cost(M)$ because there is an infinite number of possible M_b's. Instead, ZDT derives a lower bound $LB(M_b)$ for $cost(M_b)$ of *any* M_b, and solves the smallest δ satisfying $LB(M_b) > cost(M)$. Let P_m ($|P_m|$) be the set (cardinality) of data points assigned to center m. It can be shown [ZDT06] that:

$$LB(M_b) = \sum_{p\in P} dist^2(p,m_p) + min_m(|P_m|\delta^2) - \sum_{p\in P_{move}} Y(p)$$

$$\text{where } Y(p) = \left(dist(p,m_p)+\delta\right)^2 - \left(max\left(0, min_{m\in M\setminus\{m_p\}} dist(p,m) - \delta\right)\right)^2 \quad 2.1$$

$$\text{and } P_{move} = \{p \mid Y(p) > 0\}$$

The intuition behind Equation 2.1 is: (i) we first assume that every point p stays with $m'_p \in M_b$, which corresponds to $m_p \in M$, the currently assigned center of p, and calculate the minimum cost of M_b under such an assignment; then (ii) we estimate an upper bound on the cost reduction by shifting some objects to other clusters. In step (i), the current center set M is optimal with respect to the current point assignment (every $m \in M$ is the centroid of its assigned points P_m) and achieves cost $\sum_{p\in P} dist^2(p,m_p)$. Meanwhile, moving a center $m \in M$ by δ increases the cost by $|P_m|\delta^2$ [KSS04]. Therefore, the minimum cost of M_b when one center moves to the boundary of its confinement circle (while the rest remain at their original locations) is $\sum_{p\in P} dist^2(p,m_p) + min_m(|P_m|\delta^2)$.

In step (ii), let P_{move} be the set of points satisfying the following property: for any $p \in P_{move}$, if we re-assign p to another center in M_b (i.e. other than m'_p), the total cost must decrease. Function $Y(p)$ upper-bounds the cost reduction by re-assigning p. Clearly, p is in P_{move} iff $Y(p) > 0$. Regarding $Y(p)$, it is estimated using the (squared) maximum distance $dist(p,m_p)+\delta$ between p and its assigned center $m'_p \in M_b$, subtracted by the (squared) minimum distance $min_{m\in M\setminus\{m_p\}} dist(p,m)$-$\delta$ between p and any other center $m' \in M_b$, $m' \neq m'_p$. The former is reached when m'_p is δ away from p than its original location $m_p \in M$; the latter is reached when the nearest center $m \in M$ has a corresponding location m' in M_b which is δ closer to p than m. Recall that from the definition of M_b, the distance between a center $m \in M$ and its corresponding center $m' \in M_b$ is at most δ. Finally, Zhang et al. [ZDT06] provide a method that computes the minimum δ in $O(n\log n)$ time, where n is the data cardinality. We omit further details of this algorithm since we do not compute δ in this work.

2.2 Other Related Work

Clustering, one of the fundamental problems in data mining, is closely related to k-means computation. For a comprehensive survey on clustering techniques see [JMF99]. Previous work has addressed the efficient maintenance of clusters over data streams (e.g. [BDMO03, GMM+03]), as well as moving objects (e.g. [LHY04, KMB05, JLO07]). However, in the moving object setting, the existing approaches require the objects to report every update, which is prohibitively expensive in terms of network overhead and battery power. In addition, some methods, such as [LHY04] and [JLO07], are limited to the case of linearly moving objects, whereas we assume arbitrary and unknown motion patterns.

Recently, motivated by the need to find the best locations for placing k facilities, the database community has proposed algorithms for computing k-medoids [MPP07], and adding centers to an existing k-means set [ZDXT06]. These methods address snapshot queries on datasets indexed by disk-based, spatial access methods. Papadopoulos et al. [PSM07] extend [MPP07] for monitoring the k-medoid set. Finally, there exist several systems aimed at minimizing the processing cost for continuous monitoring of spatial queries, such as ranges (e.g., [GL04, MXA04]) nearest neighbors (e.g., [XMA05, YPK05]), reverse nearest neighbors (e.g., [XZ06, KMS+07]) and their variations (e.g., [JLOZ06, ZDH08]). Other systems [MPBT05, HXL05] minimize the network overhead, but currently there is no method that optimizes both factors. Moreover, as discussed in Section 1, k-means computation is inherently more complex and challenging than continuous monitoring of conventional spatial queries. In the sequel, we propose a comprehensive framework, based on a solid mathematical background and including efficient processing algorithms.

3 Threshold-based *k*-Means Monitoring

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n moving points in the d-dimensional space. We represent the location of an object p_i ($1 \le i \le n$) at timestamp τ as $p_i(\tau) = \{p_i(\tau)[1], p_i(\tau)[2], \ldots, p_i(\tau)[d]\}$, where $p_i(\tau)[j]$ ($1 \le j \le d$) denotes the coordinate of p_i on the j-th axis. When τ is clear from the context, we simply use p_i to signify the object's position. The system does not rely on any assumption about the objects' moving patterns[2]. A central server collects the objects' locations

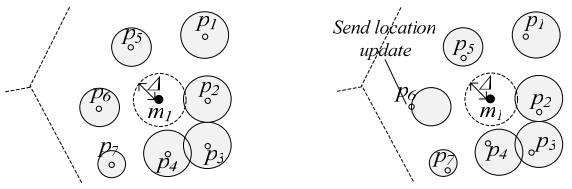
[2] The performance can be improved if each object's average speed is known. We discuss this issue in Section 4.3.

and maintains a k-means set $M = \{m_1, m_2, \ldots, m_k\}$, where each m_i ($1 \le i \le k$) is a d-dimensional point, called the i-th *cluster center*, or simply *center*. We use $m_i(\tau)$ to denote the location of m_i at timestamp τ, and $M(\tau)$ for the entire k-means set at τ. The *quality* of $M(\tau)$ is measured by the function $cost(M(\tau)) = \sum_p dist^2(p(\tau), m_p(\tau))$, where $m_p(\tau)$ is the center assigned to $p(\tau)$ at τ, and $dist(p(\tau), m_p(\tau))$ is their Euclidean distance.

The two objectives of TKM are *effectiveness* and *efficiency*. Effectiveness refers to the quality of the k-means set, while efficiency refers to the computation and network overhead. These two goals may contradict each other, as intuitively, it is more expensive to obtain better results. We assume that the user specifies a *quality tolerance* Δ, and the system optimizes efficiency while always satisfying Δ. Δ is defined based on the *reference solution* (REF) discussed in Section 1. Specifically, let $M^{REF}(\tau)$ ($M^{TKM}(\tau)$) be the k-means set maintained by REF (TKM) at τ. At any time instant τ and for all $1 \le i \le k$, $dist(m_i^{REF}(\tau), m_i^{TKM}(\tau)) \le \Delta$, where $m_i^{REF}(\tau) \in M^{REF}(\tau)$ and $m_i^{TKM}(\tau) \in M^{TKM}(\tau)$; i.e., each center in $M^{TKM}(\tau)$ is within Δ distance of its counterpart in $M^{REF}(\tau)$. Given this property, we can prove [ZDT06] that $cost(M^{TKM}(\tau)) \le cost(M^{REF}(\tau)) + n\Delta^2$, thus providing the guarantee on the quality of M^{TKM}.

TKM works as follows. At τ=0, each object sends its location to the server, which computes the initial k-means set. Then, the server transmits to every object p_i a threshold θ_i, such that p_i sends a location update if and only if it deviates from its current position (i.e. $p_i[0]$) by at least θ_i. Alternatively, the server can broadcast certain statistical information, and each object computes its own θ_i's locally. Figure 3.1a shows an example, where p_1-p_7 start at positions $p_1(0)$-$p_7(0)$, and receive thresholds θ_1-θ_7 from the server, respectively. At timestamp 1, the objects move to the new locations shown in Figure 3.1b. According to REF, all 7 objects have to issue location updates, whereas in TKM, only p_6 informs the server since the other objects have not crossed their thresholds. When a new object appears, it sends its location to the server. When an existing point leaves the system, it also informs the server. In both cases, the message is processed as a location update.

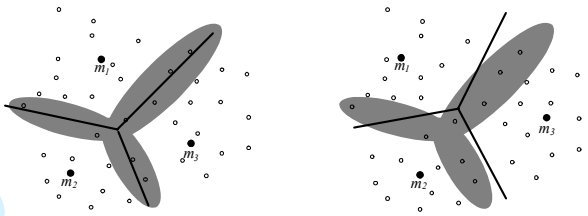
Upon receiving a location update from p_i at timestamp τ (in Figure 3.1b, from p_6), the server computes the new k-means using $p_i(\tau)$ and the last recorded positions of the other objects. Specifically, the server feeds the previous k-means set as seeds to an optimized version of HC, hereafter referred to as HC*. HC* computes exactly the same result as HC and performs the same iterations, but visits only a fraction of the dataset in each iteration.



(a) Timestamp 0 (b) Timestamp 1

Figure 3.1 Example update

HC* exploits the fact that the point assignment for the seed is usually similar to the converged result. As illustrated in Figure 3.2, only points close to the perpendicular bisectors (the shaded area), defined by the centers, may change clusters. Specifically, before an iteration starts, HC* estimates P_{active}, which is a super set of the points that may be re-assigned in the following iteration, and considers exclusively these points. Recall from Section 2.1 that an iteration consists of two steps: the first reassigns points to their respective nearest centers and the second computes the centroids of each cluster. HC* re-assigns only the points of P_{active} in the first step; whereas in the second step, for each cluster, HC* computes its new centroid based on the previous one and P_{active}. Specifically, for a cluster of points P_m, the centroid m_g is defined as $m_g[i] = \sum_{p \in P_m} (p[i] / |P_m|)$ for every dimension $1 \le i \le d$. HC* maintains an aggregate point SUM_m for each cluster such that $SUM_m[i] = \sum_{p \in P_m} p[i]$ ($1 \le i \le d$). The centroid m_g can be thus computed using SUM_m and $|P_m|$. If in the previous step of the current iteration, one point p in P_{active} is reassigned from center m to m', HC* subtracts p from SUM_m and adds it to $SUM_{m'}$.



(a) Seeds (b) Converged result

Figure 3.2 Example of P_{active}

Figure 3.3 shows the pseudo-code of HC* and clarifies the computation of P_{active}. HC* initializes P_{active} with points that will change clusters in the first iteration (Lines 1-4), and after each iteration, it adds to P_{active} an estimated superset of the points that will change cluster in the next iteration (Line 17). Specifically, for each point p, it precomputes: (i) d_p, which is the distance between p and its assigned center m_p in the seed set MS, and (ii) d'_p, the distance between p and its nearest center in $MS \setminus \{m_p\}$.

*HC** (*dataset* P, k, *Seed* MS)
1. For each point $p \in P$
2. Compute $d_p = dist(p, m_p)$, $d'_p = min_{m \in MS \setminus \{m_p\}} dist(p, m)$
3. Build a heap H on P in increasing order of $d'_p - d_p$
4. Compute set $P_{active} = \{p \mid d'_p - d_p < 0\}$ using H
5. For each center $m \in MS$
6. For each dimension i, $SUM_m[i] = \sum_{p \in P_m} p[i]$
7. Initialize $\sigma = 0$, $M = MS$
8. Repeat
9. For each point $p \in P_{active}$
10. Assign p to its nearest neighbor in M
11. If p is re-assigned from center m to m'
12. For each dimension i, adjust $SUM_m[i] = SUM_m[i] - p[i]$ and $SUM_{m'}[i] = SUM_{m'}[i] + p[i]$
13. For each center $m \in M$, corresponding to $m_s \in MS$
14. For each dimension i, $m_g[i] = SUM_m[i] / |P_m|$
15. Adjust $\sigma = max(\sigma, dist(m_s, m_g))$
16. Replace m with m_g
17. Adjust $P_{active} = \{p \mid d'_p - d_p < 2\sigma\}$ using the heap H
18. Until no change happens in M

Figure 3.3 Algorithm HC*

Figure 3.4 illustrates an example. Points satisfying $d'_p < d_p$ are re-assigned in the first iteration. HC* finds such points using a heap H sorted in increasing order of $d'_p - d_p$ (Lines 3-4). During the iterations, HC* maintains a value σ, which is the maximum distance that a center m deviates its original position $m_s \in MS$ in all previous iterations. After finishing an iteration, only points satisfying the property $d'_p - d_p < 2\sigma$ may change cluster in the next one, because in the worst case, $m_p \in MS$ moves σ distance away from p and another center moves σ distance closer, as illustrated in Figure 3.4. HC* extracts such points from the heap H as the new P_{active}. Note that over all the iterations of HC*, σ is monotonically non-decreasing (Line 15), meaning that P_{active} never shrinks when adjusted (Line 17), reaching its maximum size after HC* terminates. Let this maximum size be n_a ($n_a \le n$), and L be the number of iterations HC* takes to converge. The overall time complexity of HC* is $O(n_a \log n + Ln_ak)$, where the first term ($n_a \log n$) is the time consumed by computing and adjusting P_{active} using the heap. In contrast, the unoptimized HC takes $O(Lnk)$ time. Note that the two algorithms take exactly the same number of iterations (L) to converge. Following the assumption that only a fraction of points change clusters, n_a is expected to be far smaller than n, thus HC* is significantly faster. These theoretical results are confirmed by our experimental evaluation in Section 6.

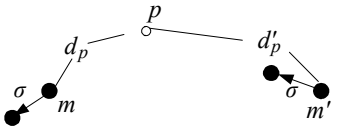


Figure 3.4 Illustration of d_p and d'_p

After HC* converges, TKM computes the new thresholds. Most thresholds remain the same, and the server only needs to send messages to a fraction of the objects. An object may discover that it has already incurred a violation, if its new threshold is smaller than the previous one. In this case, it sends a location update to the server. In the worst case, all objects have to issue updates, but this situation is rare. The main complication in TKM is how to compute the thresholds so that the guarantee on the result is maintained, and at the same time, the average update frequency of the objects is minimized. We discuss these issues in the following section.

4 Threshold Assignments

Section 4.1 derives a mathematical formulation of the threshold assignment problem. Section 4.2 proposes an algorithm for threshold computation. Section 4.3 integrates the objects' speed in the threshold computation, assuming that this knowledge is available. Section 4.4 discusses methods for threshold dissemination.

4.1 Mathematical Formulation of Thresholds

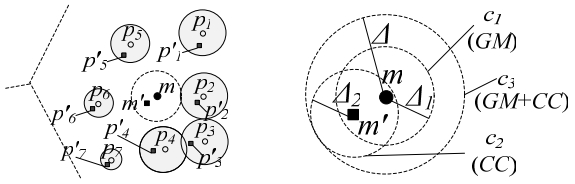
The threshold assignment routine takes as input the objects' locations $P = \{p_1, p_2, \ldots, p_n\}$ and the k-means set $M = \{m_1, m_2, \ldots, m_k\}$, and outputs a set of n real values $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, i.e. the thresholds. We formulate this task into a constrained optimization problem, where the objective is to minimize the average update frequency of the objects subject to the user's tolerance Δ. We first derive the objective function. Without any knowledge of the motion patterns, we assume that the average time interval between two consecutive location updates of each object p_i is proportional to θ_i. Intuitively, the larger the threshold, the longer the object can move in an arbitrary direction without violating it. Considering that all objects have equal weights, the (minimization) objective function is:

$$\sum_{i=1}^{n} 1/\theta_i \qquad 4.1$$

Next we formulate the requirement that no center in M^{TKM} deviates from its counterpart in M^{REF} by more than Δ. Note that this requirement can always be satisfied by setting $\theta_i = 0$ for all $1 \le i \le n$, in which case TKM reduces to REF. If $\theta_i > 0$, TKM re-computes M only when there is a violation of θ_i. Accordingly, when all objects are within their thresholds, the execution of HC*[3] should not cause any center to shift from its original position by more than Δ.

As shown in Section 2.1, the result of HC* can be predicted in the form of confinement circles. Compared to the static case [ZDT06], the problem of deriving confinement circles in our settings is further complicated by the fact that the exact locations of the objects are unknown. Consequently, for each cluster of points P_m assigned to center $m \in M$, m is not necessarily the centroid of P_m, since the points may have moved inside their confinement circles. In the example of Figure 4.1a, m is the assigned center for points p_1-p_7, computed by HC* on the points' original locations (denoted as dots). After some time, the points are located at p'_1-p'_7 shown as solid squares. Their new centroid is m', rather than m.

On the other hand, ZDT relies on the assumption that each center is the centroid of its assigned points. To bridge this gap, we split the tolerance Δ into $\Delta = \Delta_1 + \Delta_2$, and specify two constraints called the *geometric-mean* (GM), and the *confinement-circle* (CC) constraint. GM requires that for each cluster centered at $m \in M$, the centroid m' must be within Δ_1-distance from m, while CC demands that starting from each centroid m', the derived confinement circle must have a radius no larger than Δ_2. Figure 4.1b visualizes these constraints: according to GM, m' should be inside circle c_1, whereas according to CC, the confinement circle of m', and thus the center in the converged result, should be within c_2. Combining GM and CC, the center in the converged result must be within circle c_3, which expresses the user's tolerance Δ.



(a) Centroid m' (b) Constraints

Figure 4.1 Dividing Δ into Δ_1 and Δ_2

[3] Since HC* performs exactly the same iterations as HC, the results of [ZDT06] apply directly to HC*.

We now formulate GM and CC. For GM, the largest displacement occurs when all points move in the same direction, in which case the deviation of the centroid is the average of the points' displacement. Let $|P_m|$ be the cardinality of data points assigned to m, and θ_p be the threshold of p. GM requires that:

$$\forall m, \frac{1}{|P_m|}\sum_{p\in P_m}\theta_p \le \Delta_1 \quad (4.2)$$

Note that Inequality 4.2 expresses k constraints, one for each cluster. The formulation of CC follows the general idea of ZDT, except that we do not know the objects' current positions, or their centroids. In short, we are trying to predict the output of HC* without having the exact input. Let p'_i be the precise location of p_i, and $P'=\{p'_1, p'_2, \ldots, p'_n\}$. $M' = \{m'_1, m'_2, \ldots m'_k\}$, where m'_i is the centroid of all points assigned to $m_i \in M$. Similar to Section 2.1, $m'_{p'}$ denotes the assigned center for a point $p' \in P'$; $P_{m'}$ is the set of points assigned to center $m' \in M'$. By applying ZDT on the input P' and M', and using Δ_2 as the radius of the confinement circles, we obtain:

$$LB(M_b) > cost(M')$$
$$\text{where } LB(M_b) = \sum_{p'\in P'} dist^2(p', m_{p'}) + min_{m'}(|P'_{m'}|\Delta_2^2) - \sum_{p'\in P'_{move}} Y(p')$$
$$\text{and } Y(p') = \left(dist(p', m'_{p'}) + \Delta_2\right)^2 - \left(max\left(0, min_{m'\in M'\setminus\{m'_{p'}\}} dist(p', m') - \Delta_2\right)\right)^2 \quad (4.3)$$
$$\text{and } P'_{move} = \{p' \mid Y(p') > 0\}$$

Inequality 4.3 is equivalent to 2.1, except for their inputs (P, M in 2.1; P', M' in 4.3). We apply known values (i.e. P, M) to re-formulate Inequality 4.3. First, it is impossible to compute $cost(M')$ with respect to P' since both P' and M' are unknown. Instead, we use $\sum_{p'\in P'} dist^2(p', m'_{p'})$ as an upper bound on $cost(M')$; i.e., every point $p' \in P'$ is assigned to $m'_{p'}$, which is not necessarily the nearest center for p'. Comparing $LB(M_b)$ and $cost(M')$, we reduce the inequality $LB(M_b) > cost(M')$ to:

$$\sum_{p'\in P'_{move}} Y(p') \le min_{m'}(|P_{m'}|\Delta_2^2) \quad (4.4)$$

The right side of Inequality 4.4, is exactly $min_m(|P_m|\Delta_2^2)$, since the point assignment in M' is the same as in M. It remains to re-formulate the function $Y(p)$ with known values. Recall from Section 2.1 that $Y(p)$ is an upper bound of the maximum cost reduction by shifting point p to another cluster. We thus derive an upper bound $UB\_Y(p)$ of $Y(p)$ using the upper bound and lower bound of $dist(p', m')$ for a point/center pair p' and m'. Because each $p' \in P'$ is always within θ_p distance to its corresponding point $p \in P$, and each center $m' \in M'$ is within Δ_1 distance to its corresponding center $m \in M$, we have:

$$\forall p' \in P', \forall m' \in M', \left|dist(p', m') - dist(p, m)\right| \le \Delta_1 + \theta_p \quad (4.5)$$

Using Inequality 4.5, we obtain $UB\_Y(p)$ in Equation 4.6, where $\Delta = \Delta_1 + \Delta_2$:

$$UB\_Y(p) = \left(dist(p, m_p) + \Delta + \theta_p\right)^2 - \left(max\left(0, min_{m\in M\setminus\{m_p\}} dist(p, m) - \Delta - \theta_p\right)\right)^2 \quad (4.6)$$

Summarizing, we formulate constraint CC into:

$$\sum_{p\in P_{move}} UB\_Y(p) \le min_m(|P_m|\Delta_2^2)$$
$$\text{where } UB\_Y(p) = \left(dist(p, m_p) + \Delta + \theta_p\right)^2 - \left(max\left(0, min_{m\in M\setminus\{m_p\}} dist(p, m) - \Delta - \theta_p\right)\right)^2 \quad (4.7)$$
$$\text{and } P_{move} = \{p \mid UB\_Y(p) > 0\}$$

Thus, the optimization problem for threshold assignment has the objective function expressed by equation (4.1), subject to the GM (Inequality 4.2) and CC (Inequality 4.7) constraints. However, the quadratic and non-differentiable nature of Inequality 4.7 renders an efficient solution for the optimal values of θ_i infeasible. Instead, in the next section we propose a simple and effective algorithm.

4.2 Computation of Thresholds

In order to derive a solution, we simplify the problem by fixing Δ_1 and Δ_2 as constants. The ratio of Δ_1 and Δ_2 is chosen by the server and adjusted dynamically, but we defer the discussion of this issue until the end of the subsection. We adopt a two-step approach: in the first step, we ignore constraint CC, and compute a set of thresholds using only Δ_1; in the second step, we shrink thresholds that are restricted by CC. The problem of the first step is solved by the *Lagrange multiplier method* [AW95]. The optimal solution is:

$$\theta_1 = \theta_2 = \ldots = \theta_n = \Delta_1 \quad (4.8)$$

Note that all thresholds have the same value because both the objective function (4.1) and constraint GM (4.2) are symmetric. We set each $\theta_i = \Delta_1$, and use these values to compute set P_{move}. In the next step, when we decrease the threshold θ_p of a point p, P_{move} may change accordingly. We first assume that P_{move} is fixed, and later propose solutions when this assumption does not hold.

Regarding the second step, according to Inequality 4.7, constraint CC only restricts the thresholds of points $p \in P_{move}$. Therefore, to satisfy CC, it suffices to decrease the thresholds of those points. Next we study UB_Y in Inequality 4.7. Note that UB_Y is a function of threshold θ_p, associated with point p. We observe that UB_Y is in the form of $(A+\theta_p)^2-(max(0,(B-\theta_p))^2)$, where A, B are constants with respect to θ_p. Hence, as long as $B-\theta_p \ge 0$, UB_Y is linear to θ_p because the two quadratic terms cancel out; on the other hand, when $B-\theta_p < 0$, UB_Y becomes quadratic to θ_p. The turning point where UB_Y changes from linear to quadratic is the only non-differentiable point of UB_Y. Therefore, we simplify the problem by making $B-\theta_p \ge 0$ a hard constraint, which ensures that UB_Y is always differentiable and linear to θ_p. Specifically, we have the following constraint:

$$\theta_p \le min_{m\in M\setminus\{m_p\}} dist(p, m) - \Delta \quad (4.9)$$

We then decrease the affected threshold values accordingly[4]. Inequality 4.9 enables us to transform constraint CC (Inequality 4.7) to:

$$\sum_{p\in P_{move}} UB\_Y(p) \le min_m\left(|P_m|\Delta_2^2\right)$$

$$\text{where } UB\_Y(p) = 2\left(dist(p,m_p) + min_{m\in M\setminus\{m_p\}} dist(p,m)\right)\theta_p + \left(dist(p,m_p)+\Delta\right)^2 - \left(min_{m\in M\setminus\{m_p\}} dist(p,m) - \Delta\right)^2 \quad 4.10$$

Note that P_{move} is already determined in the previous step. Solving the optimization problem (again using *Lagrange multiplier*) with (4.1) as the objective function and (4.10) as the only constraint, the optimal solution is:

$$\forall p \in P_{move}, \theta_p^* = \frac{\sqrt{d(p)+d'(p)}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{d(q)+d'(q)}} \times\left(min_m\left(|P_m|\Delta_2^2\right) - \sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2 - \left(d'(q)-\Delta\right)^2\right) \quad 4.11$$

$$\text{where } d(p) = dist(p,m_p), d'(p) = min_{m\in M\setminus\{m_p\}} dist(p,m)$$

After that, for each threshold θ_p of point $p\in P_{move}$ whose current value is above θ_p^*, we decrease it to θ_p^*. In rare cases, this decrease causes $UB\_Y(p)\le 0$, meaning that p should not be a member of P_{move}. When this happens, we remove p from P_{move}, and repeat step 2 until P_{move} is stable, which signifies that CC is satisfied. In practice, this procedure usually converges very fast. Because we never increase any threshold value, GM is still satisfied, thus we now have a feasible solution to the original problem. The algorithm for threshold computation is summarized in Figure 4.2.

Compute_Thresholds(*Dataset P*, *Center set M*)
1. Set all θ_i ($1\le i\le n$) to Δ_1 // step 1
2. Compute P_{move} using the current θ_i's
3. For each point $p \in P_{move}$ // step 2
4. Decrease θ_p so that it satisfies Inequality 4.9
5. Let θ_p^* be results of evaluating Equation 4.11
6. If $\theta_p^* < \theta_p$, $\theta_p = \theta_p^*$
7. For each point $p \in P_{move}$
8. If $UB\_Y(p) \le 0$, Remove p from P_{move}
9. If P_{move} has changed, Goto Line 3 // repeat step 2

Figure 4.2 Algorithm for computing thresholds

Finally we discuss the choice of Δ_1 and Δ_2. Let $a = \Delta_1/\Delta$. Initially, TKM chooses a according to past experience, e.g. 0.7 in our experiments. After the system starts, TKM examines periodically whether it can improve performance by adjusting a. Specifically, using the routine for computing thresholds described above, we express the objective function (4.1) as a function of a and analyze the first order derivative of it. Let P_{GM} be the set of points assigned threshold Δ_1 in the routine described above, and $P_{CC} = P\setminus P_{GM}$. Then, the objective function is expressed as:

$$obj(\alpha) = \sum_{p\in P_{GM}}\frac{1}{\alpha\Delta} + \sum_{p\in P_{CC}}\frac{1}{\theta_p^*}$$

$$\text{where } \theta_p^* = \frac{\sqrt{d(p)+d'(p)}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{d(q)+d'(q)}} \times\left(min_m\left(|P_m|\right)(1-\alpha)^2\Delta^2 - \sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2 - \left(d'(q)-\Delta\right)^2\right) \quad 4.12$$

$$\text{and } d(p) = dist(p,m_p), d'(p) = min_{m\in M\setminus\{m_p\}} dist(p,m)$$

The derivative $obj'(a)$ is computed during the assignment process. If the absolute value of $obj'(a)$ is above a predefined threshold, TKM adjusts a by adding (subtracting) a step value ε to (from) it, depending on whether $obj'(a) < 0$. In our experiments, we simply set $\varepsilon = 0.05a$. Alternatively, ε can be a function of $obj'(a)$.

4.3 Utilizing the Object Speed

In practice, the speed of moving users is restricted by their means of transportation (e.g., pedestrians vs. cars etc). In this section, we assume that the system knows roughly the average speed s_i of each object p_i ($1\le i\le n$). Intuitively, fast objects are assigned larger thresholds, so that they do not need to issue updates in the near future. In addition to location updates, p_i sends a *speed update* to the server when its speed changes significantly, e.g. a pedestrian mounts a motor vehicle. We first re-formulate the constrained optimization problem to incorporate the speed information. The two constraints GM and CC remain the same, but the objective function (4.1) becomes:

$$\sum_{i=1}^{n} s_i/\theta_i \quad 4.13$$

The rationale is that when p_i moves constantly in one direction, it is expected to violate the threshold after θ_i/s_i timestamps. Next we compute the thresholds using the approach of Section 4.2. In the first step, we solve the optimization problem with (4.13) as the objective and (4.2) as the only constraint. Using the Lagrange multiplier method, we obtain the optimal solution:

$$\forall i, \theta_i = n\Delta_1\sqrt{s_i}\Big/\sum_{j=1}^{n}\sqrt{s_j} \quad 4.14$$

Comparing Equation 4.8 with 4.14, the former gives equal thresholds to all objects, while the latter assigns thresholds proportional to the square root of the objects' speeds. In the second step, we solve the problem with (4.13) as the objective function and (4.10) as the only constraint. Let s_p be the speed of object p, the optimal solution is:

$$\forall p \in P_{move}, \theta_p^* = \frac{\sqrt{\left(d(p)+d'(p)\right)s_p}}{2\left(d(p)+d'(p)\right)\sum_{q\in P_{move}}\sqrt{\left(d(q)+d'(q)\right)s_q}} \times\left(min_m\left(|P_m|\Delta_2^2\right) - \sum_{q\in P_{move}}\left(d(q)+\Delta\right)^2 - \left(d'(q)-\Delta\right)^2\right) \quad 4.15$$

$$\text{where } d(p) = dist(p,m_p), d'(p) = min_{m\in M\setminus\{m_p\}} dist(p,m)$$

The threshold routine of Figure 4.2 is modified accordingly, by substituting Equation 4.8 with Equation 4.14, and Equation 4.11 with Equation 4.15.

4.4 Dissemination of Thresholds

After computing the thresholds, the server needs to disseminate them. We propose two approaches, depending on the computational capabilities of the objects. The first

[4] Inequality 4.9 may require $\theta_p<0$. If this pathological case occurs, we set θ_p to a pre-defined value, treat it as a constant, and solve the optimization problem with one less variable.

(a) *spatial* (b) *road*

Figure 5.1 Illustration of the datasets

is based on broadcasting, motivated by the fact that the network overhead of broadcasting is significantly smaller than sending individual messages to all moving objects, and does not increase with the dataset cardinality. Initially, the server broadcasts Δ. After the first k-means set is computed, the broadcast information includes the center set M, Δ_1 (if it has changed with respect to the previous broadcast), the two sum values in Equation 4.11, and $min_m\ |P_m|$. Each object computes its own threshold based on the broadcast information.

The second approach assumes that objects have limited computational capabilities. Initially, the server sends Δ_1 to all objects through single-cast messages. In subsequent updates, it sends the threshold to an object only when it has changed. Note that most objects, besides those in P_{move}, have identical threshold Δ_1. As we show experimentally in Section 5, usually P_{move} is only a small fraction of P. Therefore, the overhead of sending these the thresholds is not large. Alternatively, we propose a variant of TKM that sends messages only to objects that issue location updates. Let P_u be the set of objects that have issued updates, and Θ_{old} be the set of thresholds before processing these updates. After re-evaluating the k-means, the server computes the set of thresholds Θ_{new} for all objects. Then, it constructs the threshold set Θ as follows. For each point in P_u, its threshold in Θ is the same as in Θ_{new}, whereas all other thresholds remain the same as in Θ_{old}. After that, the sever checks whether Θ satisfies the two constraints GM and CC. If so, it simply disseminates Θ to P_u. Otherwise, it modifies the constrained optimization problem, by treating only the thresholds of points in P_u as variables, and all other thresholds as constants whose values are from Θ_{old}. Using the two-step approach of Section 4.2, this version of TKM obtains the optimal solution Θ'_{new}. If Θ'_{new} is *valid*, i.e. each threshold is non-negative, the server disseminates Θ'_{new} to objects in P_u; otherwise, it disseminates Θ_{new} to all affected objects.

5 Experimental Evaluation

This section compares TKM against REF using two datasets. In the first one, denoted as *spatial*, we randomly select the initial position and the destination of each object from a real dataset, California Roads (available at *www.rtreeportal.org*) illustrated in Figure 5.1a. Each object follows a linear trajectory between the two points. Upon reaching the endpoint, a new random destination is selected and the same process is repeated. The second dataset, denoted as *road* [MYPM06], is based on the generator of [B02] using sub-networks of the San Francisco road map (illustrated in Figure 5.1b) with about 10K edges. Specifically, an object appears on a network node, completes a shortest path to a random destination and then disappears. To ensure that the data cardinality n (a parameter) remains constant, whenever an object disappears, a new one enters the system. In both datasets, the coordinates are normalized to the range [0..1] on each dimension, and every object covers distance 1/1000 at each timestamp. In *spatial*, movement is unrestricted, whereas in *road*, objects are restricted to move on the network edges.

REF employs HC to re-compute the k-means set, whereas TKM utilizes HC*. Since every object moves at each timestamp, there are threshold violations in most timestamps, especially for small tolerance values. This implies that TKM usually resorts to re-computation from scratch and the CPU gains with respect to REF are mostly due to HC*. For the dissemination of thresholds we use the single-cast protocol, where the server informs each object individually about its threshold. Table 5.1 summarizes the parameters under investigation. The default (median) values are typeset in boldface[5].

Table 5.1 Experimental parameters

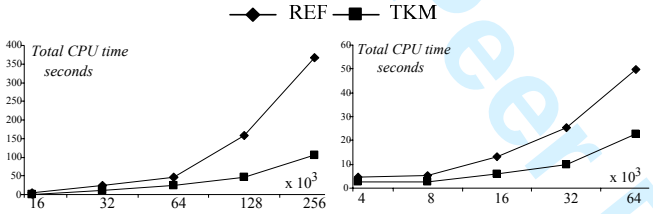
Parameter	*Spatial*	*Road*
Cardinality n ($\times 10^3$)	16, 32, **64**, 128, 256	4, 8, **16**, 32, 64
Number of means k	2, 4, 8, **16**, 32, 64	2, 4, 8, **16**, 32, 64
Tolerance Δ	0.0125, 0.025, **0.05**, 0.1, 0.2	0.05, 0.1, **0.2**, 0.4, 0.8

In each experiment we vary a single parameter, while setting the remaining ones to their median values. The

[5] Due to the huge space and time requirements of simulations in large network topologies, we use smaller values for n in *road*.

reported results represent the average value over 10 simulations. For each simulation, we monitor the k-means set for 100 timestamps. We measure: (i) the overall CPU time at the server for all timestamps, (ii) the total number of messages exchanged between the server and the objects, and (iii) the quality of the solutions.

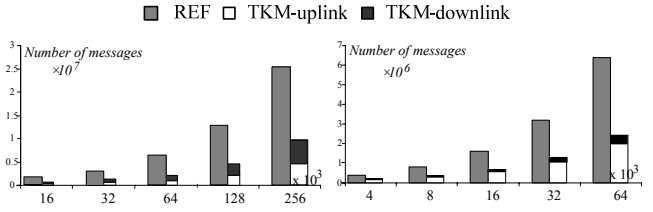
Figure 5.2 evaluates the effect of the object cardinality n on the computation cost for the *spatial* and *road* datasets. The CPU overhead of REF is dominated by recomputing the k-means set using HC at every timestamp, whereas the cost of TKM consists of both k-means computations with HC*, and threshold evaluations. The results show that TKM consistently outperforms REF, and the performance gap increases with n. This confirms both the superiority of HC* over HC and the efficiency of the threshold computation algorithm, which increases the CPU overhead only marginally. An important observation is that TKM scales well with the object cardinality (note that the x-axis is in logarithmic scale).



(a) *spatial* (b) *road*

Figure 5.2 CPU time versus data cardinality n

Figure 5.3 illustrates the total number of messages as a function of n. In REF all messages are *uplink*, i.e., location updates from objects to the server. TKM includes also *downlink* messages, by which the server informs the objects about their new thresholds. We assume that the costs of uplink and downlink messages are equal[6]. In both datasets, TKM achieves more than 60% reduction on the overall communication overhead. Specifically, TKM incurs significantly fewer uplink messages since an object does not update its location while it remains inside its threshold. On the other hand, the number of downlink messages in TKM never exceeds the number of uplink messages, which is ensured by the threshold dissemination algorithm.



(a) *spatial* (b) *road*

Figure 5.3 Number of messages versus data cardinality n

Interestingly, the number of downlink messages in the *road* dataset is smaller than in *spatial*. This is because TKM only updates the thresholds of the objects lying close to the boundary of clusters (i.e. those in P_{move}). In *road*, objects' movements are restricted by the road network, leading to highly skewed distributions. Consequently, the majority of objects lie close their respective centers. In *spatial*, however, the distribution of the objects is more uniform; therefore, a large number of objects lie on the boundary of the clusters.

Having established the performance gain of TKM with respect to REF, we evaluate its effectiveness. Figure 5.4 depicts the average squared distance achieved by the two methods versus n. The column *MaxDiff* corresponds to the maximum cost difference between TKM and REF in any timestamp. Clearly, the average cost achieved by TKM is almost identical to that of REF. Moreover, *MaxDiff* is negligible compared to the average costs, and grows linearly with n, since it is bounded by $n\Delta^2$ as described in Section 3.1.

n ($\times 10^3$)	TKM	REF	MaxDiff
16	142.23	142.15	0.18
32	289.58	289.40	0.34
64	566.25	566.14	0.70
128	1139.87	1139.64	1.40
256	2302.60	2302.22	2.82

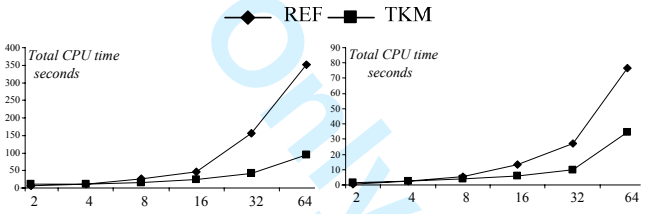
(a) *spatial*

n ($\times 10^3$)	TKM	REF	MaxDiff
4	19.67	19.46	1.012
8	37.78	37.69	1.050
16	76.07	76.05	1.849
32	151.09	150.73	2.913
64	303.21	302.69	3.884

(b) *road*

Figure 5.4 Quality versus data cardinality n

Next we investigate the impact of the number of means k. Figure 5.5 plots the CPU overhead as a function of k. TKM outperforms REF on all values of k, although the difference for small k is not clearly visible due to the high cost of large sets. The advantage of TKM increases with k because the first step of HC has to find the nearest center of every point, a process that involves k distance computations per point. On the other hand, HC* only considers points near the cluster boundaries, therefore saving unnecessary distance computations.



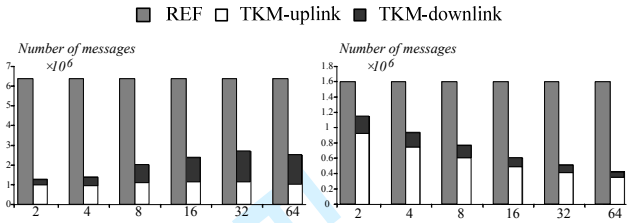
(a) *spatial* (b) *road*

Figure 5.5 CPU time versus number of means k

Figure 5.6 studies the effect of k on the communication cost. While consistently better than REF, TKM behaves differently in the two datasets. In *spatial*, the number of uplink messages is stable, and the downlink messages increase with k. This is because the objects are more uniformly distributed, and a larger k causes an increased boundary area, meaning that more points need to update their thresholds. On the other hand, in the *road* dataset, the numbers of both uplink and downlink messages decline with the increase of k. The dominating factor in *road* is the high degree of skewness of the objects. A larger k fits this distribution better because more dense areas are

[6] In practice uplink messages are more expensive in terms of battery consumption (i.e., the battery consumption is 2-3 times higher in the sending than the receiving mode [DVCK99]). This distinction favors TKM.

populated with centers. Consequently, the boundary area decreases, leading to fewer downlink messages. The number of uplink messages also decreases because the assigned thresholds are larger, since the objects are closer to their respective centers.



(a) *spatial* (b) *road*

Figure 5.6 Number of messages vs. number of means k

Figure 5.7 measures the result quality as a function of k. The average k-means cost of TKM and REF are very close for all values of k. In terms of the maximum cost difference, the two dataset exhibit different characteristics. The result of *spatial* is insensitive to k, whereas in *road*, the maximum difference increases with k. The reason is that since the objects are skewed in *road*, a slight deviation of a center causes a large increase in cost. This effect is more pronounced with large k, since the centers fit the distribution better. Nevertheless, the quality guarantee $n\Delta^2$ is always kept.

k	TKM	REF	MaxDiff
2	5552.35	5552.20	1.080
4	2231.06	2230.87	1.440
8	1156.37	1156.68	1.010
16	566.25	566.14	1.565
32	286.58	286.57	1.373
64	143.78	143.64	1.362

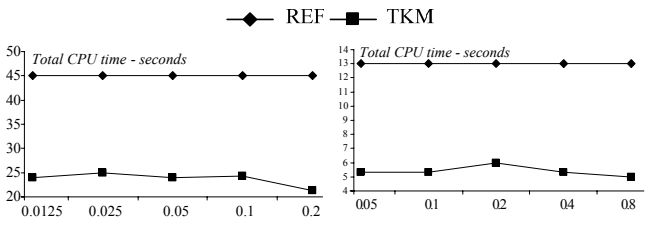
(a) *spatial*

k	TKM	REF	MaxDiff
2	678.97	678.94	0.36
4	326.25	326.23	0.15
8	166.11	166.06	0.70
16	76.07	76.05	1.85
32	36.98	36.95	2.47
64	18.09	17.67	2.85

(b) *road*

Figure 5.7 Quality versus number of means k

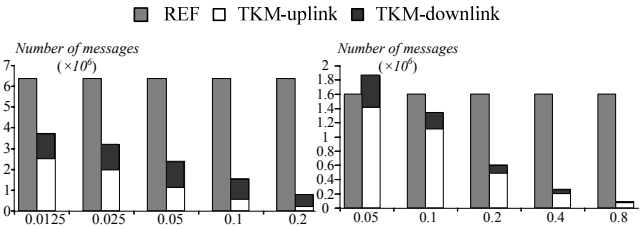
The last set of experiments studies the effect of the user's quality tolerance Δ. Because REF does not involve Δ, its results are constant in all experiments, and we focus mainly on TKM. Figure 5.8 demonstrates the effect of Δ on the CPU cost, where TKM is the clear winner for all settings. Specifically, its CPU time is relatively stable with respect to Δ, except for very large values. In these cases, the CPU time of TKM drops because at some timestamps, no objects issue location updates (and the server does not need to perform any computation).



(a) *spatial* (b) *road*

Figure 5.8 CPU time versus tolerance Δ

Figure 5.9 shows the impact of Δ on the communication cost. The numbers of both uplink and downlink messages drop as Δ increases, with the former dropping faster than the latter. Naturally, the number of uplink messages decreases because the update frequency is inversely proportional to the threshold value. The downlink messages drop for two reasons. First, in the threshold computation procedure, the value of each threshold θ^* given by Equation 4.11 increases as Δ grows. According to the algorithm, when θ^* exceeds Δ_1, the object's threshold remains Δ_1. Consequently, when Δ is large, many objects in P_{move} retain Δ_1, and do not need to be updated by a downlink message. The second reason is that for large Δ, at some timestamps there are no location updates, and therefore, no threshold dissemination.



(a) *spatial* (b) *road*

Figure 5.9 Number of messages versus tolerance Δ

Figure 5.10 investigates the effect of Δ on the result quality. As expected, in both datasets the result quality drops with the larger Δ. This is more pronounced in *road* due to the more skewed distribution. Observe that even when Δ is very large (reaching 80% of the axis length in *road*), TKM is still competitive in terms of quality.

Δ	TKM	REF	MaxDiff
0.0125	566.21	566.14	1.212
0.025	566.21	566.14	1.272
0.05	566.25	566.14	1.565
0.1	566.54	566.14	1.722
0.2	569.39	566.14	8.535

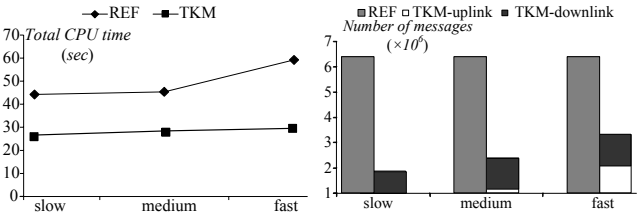
(a) *spatial*

Δ	TKM	REF	MaxDiff
0.05	75.41	76.05	0.072
0.1	75.52	76.05	0.651
0.2	76.07	76.05	1.849
0.4	76.87	76.05	7.535
0.8	81.56	76.05	13.829

(b) *road*

Figure 5.10 Quality versus tolerance Δ

Finally, Figure 5.11 demonstrates the impact of varying object speed on the *spatial* dataset. Specifically, every object at *medium* speed covers distance 1/1000 at each timestamp (i.e., the default speed value). *Slow* and *fast* objects cover distance 0.5/1000 and 1.5/1000, respectively. In each setting, all objects move with the same speed, and the other parameters are set to their default values. As shown in Figure 5.11a, the CPU cost of REF increases with the object speed, because the faster the objects move, the farther away the centers diverge from their original locations. Thus, HC needs more iterations to converge. In contrast, TKM is less sensitive to object speed due to the robustness of the underlying HC* algorithm. Figure 5.11b shows that TKM outperforms REF also in terms of communication overhead. The performance gap, however, shrinks as the speed increases because a higher speed causes more objects to cross the boundary of their assigned thresholds, necessitating location updates. Finally, according to Figure 5.11c, the result quality of TKM is always close to that of REF, and their difference is well below the user-specified threshold Δ (5%).



(a) *total CPU time vs. speed* (b) *number of messages vs. speed*

Speed	TKM	REF	MaxDiff
slow	604.80	604.60	0.125
medium	566.25	566.14	1.565
fast	512.73	512.55	2.804

(c) *Quality vs. speed*

Figure 5.11 Effect of object speed (*spatial* dataset)

Summarizing the experimental evaluation, TKM outperforms REF by a wide margin on both CPU cost and communication overhead, while it incurs negligible deterioration of the result quality. Therefore, it allows monitoring of *k*-means in very large datasets of continuously moving objects. Moreover, TKM scales better than REF on the number of means, implying that it can be used in applications that require high values of *k*.

6 Conclusions

This paper proposes TKM, the first approach for continuous *k*-means computation over moving objects. Compared to the simple solution of re-evaluating *k*-means for every object update, TKM achieves considerable savings by assigning each object a threshold, such that the object needs to inform the server only when there is a threshold violation. We present mathematical formulae and an efficient algorithm for threshold computation. In addition, we develop an optimized hill climbing technique for reducing the CPU cost, and discuss optimizations of TKM for the case that object speeds are known. Finally, we design different threshold dissemination protocols depending on the computational capabilities of the objects.

In the future, we plan to extend the proposed techniques to related problems. For instance, *k*-medoids are similar to *k*-means, but the centers are restricted to points in the dataset. TKM could be used to find the *k*-means set, and then replace each center with the closest data point. It would be interesting to study performance guarantees (if any) in this case, as well as devise adaptations of TKM for the problem. Finally, another direction concerns distributed monitoring of *k*-means. In this scenario, there exist multiple servers maintaining the locations of distinct sets of objects. The goal is to continuously compute the *k*-means using the minimum amount of communication between servers.

References

[AV06] Arthur, D., Vassilvitskii, S. How Slow is the *k*-Means Method. *SoCG*, 2006.

[AW95] Arfken, G., Weber, H. *Mathematical Methods for Physicists*. Academic Press, 1995.

[B02] Brinkhoff, T. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2): 153-180, 2002.

[BDMO03] Babcock, B., Datar, M., Motwani, R., O'Callaghan, L. Maintaining Variance and *k*-Means over Data Stream Windows. *PODS*, 2003.

[BF98] Bradley, P., Fayyad, U. Refining Initial Points for *k*-Means Clustering. *ICML*, 1998.

[DVCK99] Datta, A., Vandermeer, D., Celik, A., Kumar, V. Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users. *ACM TODS*, 24(1): 1-79, 1999.

[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.

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

[HS05] Har-Peled, S., Sadri, B. How Fast is the *k*-means Method. *SODA*, 2005.

[HXL05] Hu, H., Xu, J., Lee, D. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. *SIGMOD*, 2005.

[IKI94] Inaba, M., Katoh, N., Imai, H. Applications of Weighted Voronoi Diagrams and Randomization to Variance-based Clustering. *SoCG*, 1994.

[JMF99] Jain, A., Murty, M., Flynn, P. Data Clustering: a Review. *ACM Computing Surveys*, 31(3): 264-323, 1999.

[JLO07] Jensen, C., Lin, D., Ooi, B. C. Continuous Clustering of Moving Objects. *IEEE TKDE*, 19(9): 1161-1173, 2007.

[JLOZ06] Jensen, C., Lin, D., Ooi, B. C., Zhang, R. Effective Density Queries on Continuously Moving Objects. *ICDE*, 2006.

[KMB05] Kalnis, P., Mamoulis, N. Bakiras, S. On Discovering Moving Clusters in Spatio-temporal Data. *SSTD*, 2005.

[KMN+02] Kanungo, T., Mount, M., Netanyahu, N., Piatko, C., Silverman, R., Wu, A. An Efficient *k*-means Clustering Algorithm: Analysis and Implementation. *IEEE PAMI*, 24(7): 881-892, 2002.

[KMS+07] Kang, J. M., Mokbel, M., Shekhar, S., Xia, T., Zhang, D. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. *ICDE*, 2007.

[KR90] Kaufman, L., Rousseeuw, P. J. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[KSS04] Kumar, A., Sabharwal, Y., Sen, S. A Simple Linear Time (1+ε)-Approximation Algorithm for *k*-means Clustering in Any Dimensions. *FOCS*, 2004.

[L82] Lloyd, S. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2): 129-136, 1982.

[LHY04] Li, Y., Han, J., Yang, J. Clustering Moving Objects. *KDD*, 2004.

[M06] Meila, M. The Uniqueness of a Good Optimum for *k*-Means. *ICML*, 2006.

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

[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 Partitioning Querying: A Methodology for Computing Medoids in Large Spatial Datasets. *VLDB J.*, to appear.

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

[MYPM06] Mouratidis, K., Yiu, M., Papadias, D., Mamoulis, N. Continuous Nearest Neighbor Monitoring in Road Networks. *VLDB*, 2006.

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

[PM99] Pelleg, D., Moore, A. Accelerating Exact *k*-Means Algorithms with Geometric Reasoning. *KDD*, 1999.

[PSM07] Papadopoulos, S., Sacharidis, D., Mouratidis, K. Continuous Medoid Queries over Moving Objects. *SSTD*, 2007.

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

[XZ06] Xia, T., Zhang, D. Continuous Reverse Nearest Neighbor Monitoring. *ICDE*, 2006.

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

[ZDH08] Zhang, D., Du, Y., Hu, L. On Monitoring the top-*k* Unsafe Places. *ICDE*, 2008.

[ZDT06] Zhang, Z., Dai, B., Tung, A. On the Lower Bound of Local Optimum in *k*-means Algorithm. *ICDM*, 2006.

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

Reviewer 1: Major Revision

C1. *In the introduction, please give some specific examples to strengthen the motivation. In the first paragraph of the introduction, the range of i is not given for p_i. Please provide references when mentioning "the problem is NP-hard". Similarly, give references when saying that "several methods are based on HC".*
Response: Revised as suggested. Specifically, we have added (i) two motivating examples (in the second paragraph of the introduction), (ii) the range of i for p_i, (iii) a reference to the NP-hard nature of the problem, and (iv) three references to HC methods (in the first paragraph).

C2. *In the related work, a latest paper on moving object clustering is not cited. C. S. Jensen, D. Lin, and B. C. Ooi, "Continuous Clustering of Moving Objects", TKDE, 2007.*
Response: The new version contains several new references, including the above. As discussed in Section 2.2, [JLO07] requires the objects to move linearly, and report to the server whenever their velocity changes, whereas the proposed method does not impose any restrictions on the objects' moving patterns, and minimizes network transmissions.

C3. *It would be good to give a complete complexity analysis for the entire algorithm HC*.*
Response: In the revised version, we added the complexity analysis of HC* in the paragraph between Figure 3.3 and 3.4.

C4. *A few more experiments would be useful to provide more insights of the proposed approach. The first one is about the scalability test. In current setting, the maximum dataset is 256K, while in most other moving object works, the dataset is up to 1M. I suggest to enlarge the dataset to 1M. It is not clear whether the speed of moving objects will affect the performance. Intuitively, the faster the objects move, the less effective the threshold would be. Therefore, it is good to show some experimental result regarding the effect of object speed. The authors have used datasets based on real maps, which is very good. It could be better if the authors can also show one figure of experiments on datasets of different distributions like uniform and Gaussian, since moving objects are not limited to vehicles.*
Response: We added Figure 5.11 (and the related discussion) to evaluate the effect of the object speed. Regarding the data cardinality, although we agree that other papers on spatial queries (e.g., ranges, NN) use millions of objects, continuous *k*-means monitoring is inherently more expensive because the search space cannot be restricted (in the vicinity of the query). Actually the default and maximum cardinalities in our evaluation are already much larger than those in other *k*-means papers. Furthermore, the trends (with respect to the cardinality) are clear in the present diagrams. Finally, we do not to include uniform and Gaussian distributions because (i) they are not common in spatial applications, and (ii) the relative performance of the algorithms is similar for *road* and *spatial* despite their completely different nature, suggesting that analogous results should be obtained for any data distribution.

Reviewer 2: Accept with No Changes

C5. *Minor editing comments:*
** section 1, 1st paragraph: "Given a dataset P = {...} of points in 2D space ..."*
** section 1, 3rd paragraph: "The rationale is that M(tau) is expected to be more similar ..."*
** section 3, 1st paragraph: "cost(M(tau)) = SUM_p dist^2(...)"*
Response: Revised as suggested.

Reviewer 3: Minor Revision

C6. *My only concern is that the paper should review related work more thoroughly. In particular, as reviewed in*
D. Zhang, Y. Du, and L. Hu, "On Monitoring the top-k Unsafe Places", Proc. of 24th International Conference on Data Engineering (ICDE), Cancun, Mexico, 2008.
there is a large body of continuous monitoring queries that are related.
Response: We added six new references ([GL04, JLOZ06, JLO07, KMS+07, PSM07, XZ06]) in addition to the above, covering several types of continuous spatial queries.