Medoid Queries in Large Spatial Databases*

Kyriakos Mouratidis[1], Dimitris Papadias[1], and Spiros Papadimitriou[2]

[1] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
{kyriakos, dimitris}@cs.ust.hk
[2] Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
spapadim@cs.cmu.edu

Abstract. Assume that a franchise plans to open k branches in a city, so that the average distance from each residential block to the closest branch is minimized. This is an instance of the *k-medoids* problem, where residential blocks constitute the input dataset and the k branch locations correspond to the medoids. Since the problem is NP-hard, research has focused on approximate solutions. Despite an avalanche of methods for small and moderate size datasets, currently there exists no technique applicable to very large databases. In this paper, we provide efficient algorithms that utilize an existing data-partition index to achieve low CPU and I/O cost. In particular, we exploit the intrinsic grouping properties of the index in order to avoid reading the entire dataset. Furthermore, we apply our framework to solve *medoid-aggregate* queries, where k is not known in advance; instead, we are asked to compute a medoid set that leads to an average distance close to a user-specified parameter T. Compared to previous approaches, we achieve results of comparable or better quality at a small fraction of the CPU and I/O costs (seconds as opposed to hours, and tens of node accesses instead of thousands).

1 Introduction

Given a set P of points, we wish to find a set of medoids $R \subseteq P$ with cardinality k that minimizes the average Euclidean distance $||p - r(p)||$ between each point $p \in P$ and its closest medoid $r(p) \in R$. Formally, our aim is to minimize the function

$$C(R) = \frac{1}{|P|} \sum_{p \in P} ||p - r(p)||$$

under the constraint that $R \subseteq P$ and $|R| = k$. Figure 1 shows an example, where $|P| = 23$, $k = 3$, and $R = \{h, o, t\}$. Assuming that the points of P constitute residential blocks, the three medoids h, o, t constitute candidate locations for service facilities (e.g., franchise branches), so that the average distance $C(R)$ from each block to its closest facility is minimized. A related problem is the *medoid-aggregate* (MA) query, where k is not known in advance. The goal is to select a minimal set R of medoids, such that $C(R)$ best approximates an input value T. Considering again the franchise example, instead of specifying the number of facilities, we seek the minimum set of branches that leads to an average distance (between each residential block and the closest branch) of about $T = 500$ meters.

* Supported by grant HKUST 6180/03E from Hong Kong RGC.

C. Bauzer Medeiros et al. (Eds.): SSTD 2005, LNCS 3633, pp. 55–72, 2005.

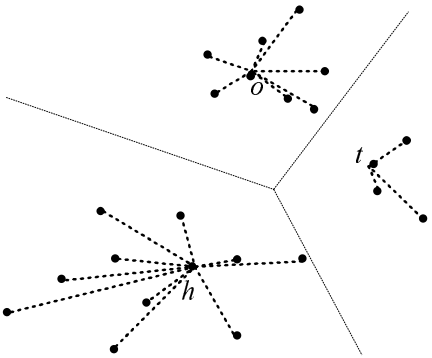


Fig. 1. Example of a k-medoid query

Efficient solutions to medoid queries are essential in several applications related to resource allocation and spatial decision making. In this paper, we propose TPAQ (*Tree-based PArtition Querying*), a strategy that avoids reading the entire dataset by exploiting the grouping properties of a data partition method on P. TPAQ initially traverses the index top-down, stopping at an appropriate level and placing the corresponding entries into groups according to proximity. Finally, it returns the most centrally located point within each group as the corresponding medoid. Compared to previous approaches, TPAQ achieves solutions of comparable or better quality, at a small fraction of the cost (seconds as opposed to hours).

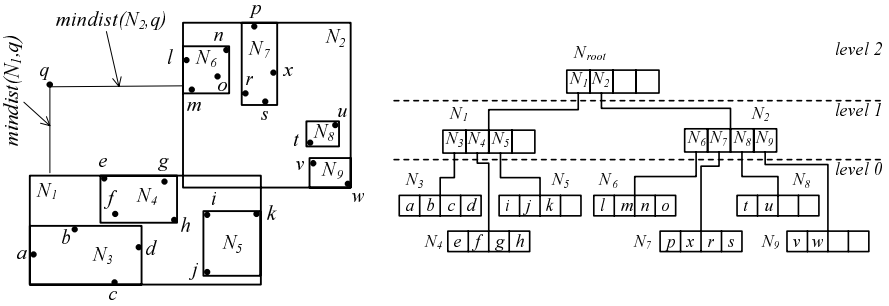
The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces key concepts and describes the intuition and the general framework for our techniques. Section 4 considers k-medoid queries and Section 5 focuses on MA queries. Section 6 presents experimental results on both real and synthetic datasets. Finally, Section 7 concludes the paper.

2 Background

Although our techniques can be used with any data-partition method, here we assume R*-trees [1] due to their popularity. Section 2.1 overviews R*-trees and their application to nearest neighbor queries. Section 2.2 presents existing algorithms for k-medoids and related problems.

2.1 R-Trees and Nearest Neighbor Search

We illustrate our examples with the R-tree of Figure 2 assuming a capacity of four entries per node. Points that are nearby in space (e.g., a, b, c, d) are inserted into the same leaf node (N_3). Leaf nodes are recursively grouped in a bottom-up manner according to their vicinity, up to the top-most level that consists of a single root. Each node is represented as a minimum bounding rectangle (MBR) enclosing all the points in its sub-tree. The nodes of an R*-tree are meant to be compact, have small margin and



(a) R-tree node extents and locations (b) R-tree data structure

Fig. 2. R-tree example

achieve minimal overlap (among nodes of the same level) [2]. Additionally, in practice, nodes at the same level contain a similar number of data points, due to a minimum utilization constraint (typically, 40%). These properties imply that the R-tree (or any other data-partition method based on similar concepts) provides a natural way to partition P according to object proximity and group cardinality criteria. However, with few exceptions (discussed in the next subsection) R-trees have been used exclusively for processing spatial queries such as range search, nearest neighbors and spatial joins.

A nearest neighbor (NN) query retrieves the data object that is closest to an input point q. R-tree algorithms for processing NN queries utilize some metrics to prune the search space. The most common such metric is $mindist(N, q)$, which is defined as the minimum possible distance between q and any point in the sub-tree rooted at node N. Figure 2 shows the $mindist$ between q and nodes N_1 and N_2. The algorithm of [3] traverses the tree in a depth-first (DF) manner: starting from the root, it first visits the node with the minimum $mindist$ (i.e., N_1 in our example). The process is repeated recursively until a leaf node (N_4) is reached, where the first potential nearest neighbor (point e) is found. Subsequently, the algorithm only visits entries whose minimum distance is less than $||e - q||$. In the example, N_3 and N_5 are pruned since their $mindist$ from q is greater than $||e - q||$. Similarly, when backtracking to the upper level, node N_2 is also excluded and the process terminates with e as the result. The extension to k (> 1) NNs is straightforward. Hjaltason and Samet [4] propose a best-first variation which is I/O optimal (i.e., it only visits nodes that may contain NNs) and incremental (the number of NNs does need to be known in advance).

2.2 *k*-Medoids and Related Problems

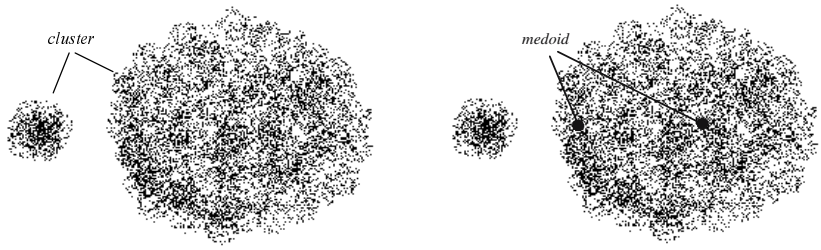
A number of approximation schemes for k-medoids[1] and related problems appear in the literature [5]. Most of these findings, however, are largely theoretical in nature. Kaufmann and Rousseeuw [6] propose *partitioning around medoids* (PAM), a practical

[1] If the selected points (R) do not necessarily belong to the dataset P (i.e., they are arbitrary points in the Euclidean space), the problem is known as *Euclidean k-medians* [5].

algorithm based on the hill climbing paradigm. In particular, PAM starts with a random set of k medoids $R_0 \subseteq P$. At each iteration i, it updates the current set R_i of medoids by exhaustively considering all *neighbor sets* R'_i that result from R_i by exchanging one of its elements with another object. For each of these $k \cdot (|P| - k)$ alternatives, it computes the function $C(R'_i)$ and chooses as R_{i+1} the one that achieves the lowest value. It stops when no further improvement is possible. Since computing $C(R'_i)$ requires $(O|P|)$ distance calculations, PAM is prohibitively expensive for large $|P|$. Thus, [6] also present *clustering large applications* (CLARA), which draws one or more random samples from P and runs PAM on those. Ng and Han [7] propose *clustering large applications based on randomized search* (CLARANS) as an extension to PAM. CLARANS draws a random sample of size *maxneighbors* from all the $k \cdot (|P| - k)$ possible neighbor sets R'_i of R_i. It performs *numlocal* restarts and selects the best local minimum as the final answer.

Although CLARANS is more scalable than PAM, it is inefficient for disk-resident datasets because each computation of $C(R'_i)$ requires a scan of the entire database. Assuming that P is indexed with an R-tree, Ester et al. [8,9] develop *focusing on representatives* (FOR) for large datasets indexed by R-trees. FOR takes the most centrally located point of each leaf node and forms a sample set, which is considered as representative of the entire P. Then, it applies CLARANS on this sample to find the k medoids. Although FOR is more efficient than CLARANS, it still has to read the entire dataset in order to extract the representatives. Furthermore, in very large databases, the leaf level population may still be too high for the efficient application of CLARANS (the experiments of [8] use R-trees with only 50,559 points and 1,027 leaf nodes).

The k-medoid problem is related to clustering. Clustering methods designed for large databases include DBSCAN [10], BIRCH [11], CURE [12] and OPTICS [13]. However, the objective of clustering is to partition data objects in groups (clusters) such that objects within the same group are more similar to each other than points in other groups. Figure 3(a) depicts a 2-way clustering for a dataset, while Figure 3(b) shows the two medoids. Clearly, assigning a facility per cluster would not achieve the purpose of minimizing the average distance between points and facilities. Furthermore, although the number of clusters depends on the data characteristics, the number of medoids is an input parameter determined by the application requirements.



(a) 2 clusters (b) 2 medoids

Fig. 3. Clustering versus medoids problem

Extensive work on medoids and clustering has been carried out in the areas of statistics [14,6,15], machine learning [16,17,18] and data mining [10,19]. However, the focus there is on assessing the statistical quality of a given clustering, usually based on assumptions about the data distribution [15,6,17,18]. Only few approaches aim at discovering the number of clusters dynamically [17,18]. Besides tackling a problem of different nature, existing algorithms are computationally intensive and unsuitable for disk-resident datasets. In summary, there is need for methods that fully exploit spatial access methods and can answer alternative types of medoid queries.

3 General Framework and Definitions

The TPAQ framework traverses the R-tree in a top-down manner, stopping at the topmost level that provides enough information for answering the given query. In the case of k-medoids, this decision depends on the number of entries at the level. On the other hand, for MA queries, the selection of the partitioning level is also based on the spatial extents and the expected cardinality of its entries. Next, TPAQ groups the entries of the partitioning level into *slots*. For given k, this procedure is performed by a fast algorithm that applies a single pass over the initial entries. For MA, multiple passes over the entries might be required. The last step returns the NN of each slot center as the medoid of the corresponding partition. We first provide some basic definitions, which are used throughout the paper.

Definition 1 (Extended entry). *An extended entry e consists of an R-tree entry N, augmented with information about the underlying data points, i.e.,*

$$e = \langle c, w, N \rangle$$

where the weight w is the expected number of points in the subtree rooted at N. The center c is a vector of co-ordinates that corresponds to the geometric centroid of N, assuming that the points in the sub-tree of N are uniformly distributed.

Definition 2 (Slot). *A slot s consists of a set E of extended entries, along with aggregate information about them. Formally, a slot s is defined as $s = \langle c, w, E \rangle$, where w is the expected number of points represented by s,*

$$w = \sum_{e \in E} e.w$$

and c is the weighted center of s,

$$c = \frac{1}{w} \sum_{e \in E} e.w \cdot e.c$$

A fundamental operation is the insertion of an extended entry e into a slot s. The pseudo-code for this function is shown in Figure 4. The insertion computes the new center taking into account the relative positions and weights of the slot s and the entry e, e.g., if s and e have the same weights, the new center is at the midpoint of the line segment connecting $s.c$ and $e.c$. Table 1 summarizes the frequently used symbols, along with short descriptions. In the subsequent sections, we describe the algorithmic details for each query type.

Function **InsertEntry**(extended entry e, slot s)
1. $s.c = (e.w \cdot e.c + s.w \cdot s.c)/(e.w + s.w)$
2. $s.w = e.w + s.w$
3. $s.E = s.E \cup \{e\}$

Fig. 4. The *InsertEntry* function

Table 1. Frequently used symbols

Symbol	Description
P	Set of data points
$\|\|p_1 - p_2\|\|$	Euclidean distance between points p_1 and p_2
R	Set of medoids
k	Number of medoids $k = \|R\|$
$r(p)$	Closest medoid of $p \in P$
$C(R)$	Average distance achieved by R
T	Target distance (for MA queries)
N	R-tree node
E	Set of entries $e_i = \langle c_i, w_i, N_i \rangle$
S	Set of slots $s_i = \langle c_i, w_i, E_i \rangle$

4 *k*-Medoid Queries

Given a k-medoid query, TPAQ finds the top-most level with $k' \geq k$ entries. For example, if $k = 3$ in the tree of Figure 2, TPAQ descends to level 1, which contains $k' = 7$ entries, N_3 through N_9. The weights of these entries are computed as follows. Since $|P| = 23$, the weight of the root node N_{root} is $w_{root} = 23$. Assuming that the entries of N_{root} are equally distributed between the two children N_1 and N_2, $w_1 = w_2 = N/2 = 11.5$ (whereas, the true cardinalities are 11 and 12, respectively). The process is repeated for the children of N_1 ($w_3 = w_4 = w_5 = w_1/3 = 3.83$) and N_2($w_6 = w_7 = w_8 = w_9 = w_2/4 = 2.87$). Figure 5 illustrates the algorithm for computing the initial set of entries. Note that *InitEntries* considers that k does not exceed the number of leaf nodes. This assumption is not restrictive because the lowest level typically contains several thousand nodes (e.g., in our datasets, between 3,000–60,000), which is sufficient for all ranges of k that are of practical interest. Nevertheless, if needed, larger values of k can be accommodated by conceptually splitting leaf level nodes.

The next step merges the k' initial entries in order to obtain exactly k groups. Initially, k out of the k' entries are selected as slot *seeds*, i.e., each of the chosen entries forms an initial slot. Clearly, the seed locations play an important role in the quality of the final answer. The seeds should capture the distribution of points in P, i.e., dense areas should contain many seeds. Our approach for seed selection is based on *space-filling curves*, which map a multi-dimensional space into a linear order. Among several alternatives, Hilbert curves best preserve the locality of points [20,21]. Therefore, we first Hilbert-sort the k' entries and select every other m-th entry as a seed, where $m = k'/k$.

Function **InitEntries**(P, k)
1. Load the root of the R-tree of P
2. Initialize $list = \{e\}$, where $e = \langle N_{root}.c, |P|, N_{root}\rangle$
3. While $list$ contains fewer than k extended entries
4. Initialize an empty list $next\_level\_entries$
5. For each $e = \langle c, w, N\rangle$ in $list$ do
6. Let num be the number of child entries in node N
7. For each entry N_i in node N do
8. $w_i = w/num$ // the expected cardinality of N_i
9. Insert extended entry $\langle N_i.c, w_i, N_i\rangle$ to $next\_level\_entries$
10. Set $list = next\_level\_entries$
11. Return $list$

Fig. 5. The *InitEntries* function

This procedure is fast and produces well-spaced seeds that follow the data distribution. Returning to our example, Figure 6 shows the level 1 MBRs (for the R-tree of Figure 2) and the output seeds $s_1 = N_4, s_2 = N_9$ and $s_3 = N_7$ according to their Hilbert order. Recall that each slot is represented by its weight (e.g., $s_1.w = w_4 = 3.83$), its center (e.g., $s_1.c$ is the centroid of N_4) and its MBR.

Then, each of the remaining $(k' - k)$ entries is inserted into the k seed slots, based on proximity criteria. More specifically, for each entry e, we choose the slot s whose weighted center $s.c$ is closest to the entry's center $e.c$. In the running example, assuming that N_3 is considered first, it is inserted into the slot s_1 using the *InsertEntry* function of Figure 4. The center of $s_1$is updated to the midpoint of N_3 and N_4's centers, as shown in Figure 7(a). TPAQ proceeds in this manner, until the final slots and weighted centers are computed as shown in Figure 7(b).

After grouping all entries into exactly k slots, we find one medoid per slot by performing a nearest-neighbor query. The query point is the slot's weighted center $s.c$, and the search space is the set of entries $s.E$. Since all the levels of the R-tree down to the partition level have already been loaded in memory, the NN queries incur very few

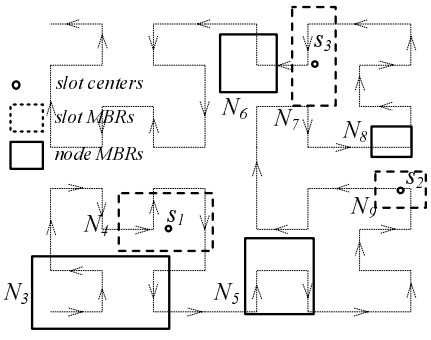
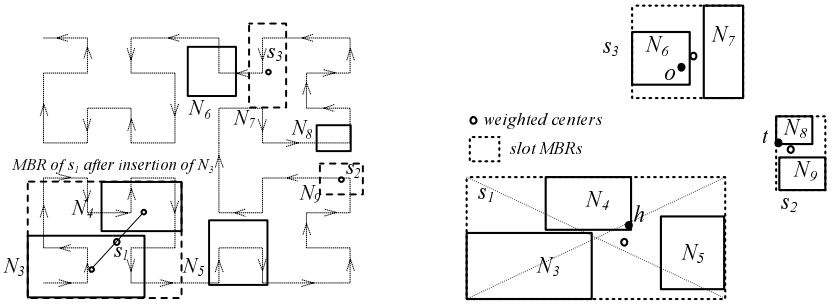


Fig. 6. Hilbert seeds on example dataset



(a) Insertion of N_3 (b) Final slot content

Fig. 7. Insertion of entries into slots

node accesses and negligible CPU cost. Observe that an actual medoid (i.e., a point in P that minimizes the sum of distances) is more likely to be closer to $s.c$ than simply to the center of the MBR of s. The intuition is that $s.c$ captures information about the point distribution within s. The NN queries on these points return the final medoids $R = \{h, o, t\}$. Figure 8 shows the complete TPAQ k-medoid computation algorithm.

Note that the problem of seeding the slot table is similar to that encountered in spatial hash joins, where the number of buckets is bounded by the available main memory [22,23,24]. However, our ultimate goals are different. First, in the case of hash joins, the table capacity is an upper bound. Reaching it is desirable in order to exploit available memory as much as possible, but falling slightly short is not a problem. In contrast, we want *exactly* k slots. Second, in our case slots should minimize the average distance $C(R)$ on one dataset, whereas slot selection in spatial joins attempts to minimize the number of intersection tests that must be performed between objects that belong to different datasets.

Algorithm **TPAQ**(P, k)
1. Initialize a set $S = \emptyset$, and empty $list$
2. Set $E =$ the set of entries returned by *InitEntries*(P, k)
3. Hilbert-sort the centers of the entries in E and store them in a sorted list $sorted\_list$
4. For $i = 1$ to k do // compute the slot seeds
5. Form a slot containing the $(i \cdot |E|/k)$-th entry of $sorted\_list$ and insert it into S
6. For each entry e in E (apart from the ones selected as seeds) do
7. Find the slot s in S with the minimum distance $||e.c - s.c||$
8. *InsertEntry*(e, s)
9. For each $s \in S$ do
10. Perform a NN search at $s.c$ on the points under $s.E$
11. Append the retrieved point to $list$
12. Return $list$

Fig. 8. The *TPAQ* algorithm

5 Medoid-Aggregate Queries

A medoid-aggregate (MA) query specifies the desired average distance T (between points and medoids), and asks for the minimal medoid set R that achieves $C(R) = T$. Consider the example of Figure 9, and assume that we know a priori all the optimal i-medoid sets R^i and the corresponding $C(R^i)$, for $i = 1, \ldots, 23$. If $C(R^4)$ is the average distance that best approximates T (compared to $C(R^i)$ $\forall i \neq 4$), set R^4 is returned as the result of the query. The proposed algorithm, TPAQ-MA, is based on the fact that as the number of medoids $|R|$ increases, the corresponding $C(R)$ decreases. In a nutshell, it first descends the R-tree of P down to an appropriate (partitioning) level. Next, it estimates the value of $|R|$ that achieves the closest average distance $C(R)$ to T and returns the corresponding medoid set R.

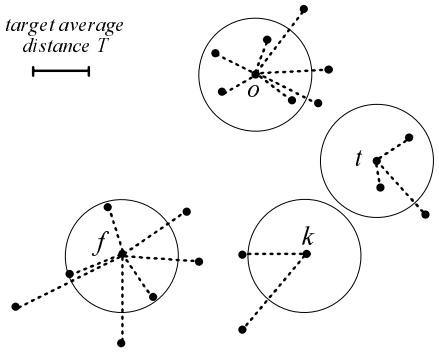


Fig. 9. A medoid-aggregate query example

The first step of TPAQ-MA is to determine the partitioning level. The algorithm selects for partitioning the top-most level whose *minimum possible average distance* (MPD) is less than or equal to T. The MPD of a level is the smallest $C(R)$ that can be achieved if partitioning takes place in this level. According to the methodology presented in Section 4, MPD equals to the $C(R)$ resulting if we extract one medoid from each entry in the level. Since computing the exact $C(R)$ requires scanning the entire dataset P, we use an estimate of $C(R)$ as the MPD. In particular, for each entry e of the level, we assume that the underlying points are distributed uniformly in its MBR[2], and that the corresponding medoid is at $e.c$. The average distance $\bar{C}(e)$ between the points in e and the $e.c$ is given by the following lemma.

Lemma 1. *If the points in e are uniformly distributed in its MBR, then their average distance from $e.c$ is*

$$\bar{C}(e) = \frac{1}{3}\left(\frac{D}{2} + \frac{B^2}{8A}\ln\left(\frac{D+A}{D-A}\right) + \frac{A^2}{8B}\ln\left(\frac{D+B}{D-B}\right)\right),$$

where A and B are the side lengths of the MBR of e and D is its diagonal length.

[2] This is a reasonable assumption for low-dimensional R-trees [2].

Proof. If we translate the MBR of e so that its center $e.c$ falls at the origin (0,0), $\bar{C}(e)$ is the average distance of points $(x, y) \in [-A/2, A/2] \times [-B/2, B/2]$ from (0,0). Hence,

$$\bar{C}(e) = \frac{1}{AB} \int_{-A/2}^{A/2} \int_{-B/2}^{B/2} \sqrt{x^2 + y^2}\, dxdy,$$

which evaluates to the quantity of Lemma 1.

The MPD of the considered level is estimated by averaging $\bar{C}(e)$ over all $e \in E$, where E is the set of entries at the level, i.e.,

$$\text{MPD} = \frac{1}{|P|} \sum_{e \in E} e.w \cdot \bar{C}(e)$$

TPAQ-MA applies the *InitEntries* function to select the top-most level that has MPD $\leq T$. The pseudo-code of *InitEntries* is the same as shown in Figure 5, after replacing the while-condition of line 3 with the expression: "the estimated MPD is more than T". Returning to our running example, the root node N_{root} of the R-tree of P has MPD $= \bar{C}(N_{root})$ higher than T. Therefore, *InitEntries* proceeds with level 2 (containing entries N_1 and N_2), whose MPD is also higher than T. Next, it loads the level 1 nodes and computes the MPD over the entries from N_3 to N_9. The MPD is less than T, and level 1 is selected for partitioning. The *InitEntries* procedure returns a list containing 7 extended entries corresponding to N_3 up to N_9.

The next step of TPAQ-MA is to determine the number of medoids that better approximate the value T. If E is the set of entries in the partitioning level, then the candidate values for $|R|$ range between 1 and $|E|$. Assuming that $C(R)$ is decreasing with respect to $|R|$, TPAQ-MA performs binary search in order to select the value of $|R|$ that yields the closest average distance to T. This procedure considers $O(\log |E|)$ different values for $|R|$, and creates slots for each of them as discussed in Section 4. Since the exact evaluation of $C(R)$ for every examined $|R|$ would be very expensive, we produce an estimate $\bar{C}(S)$ of $C(R)$ for the corresponding set of slots S. Particularly, we assume that the medoid of each slot s is located at $s.c$, and that the average distance from the points in every entry $e \in s$ equals the distance $||e.c - s.c||$. Hence, the estimated value for $C(R)$ is given by the formula

$$\bar{C}(S) = \frac{1}{|P|} \sum_{s \in S} \sum_{e \in s} e.w \cdot ||e.c - s.c||$$

where S is the set of slots produced by partitioning the entries in E into $|R|$ groups. Note that we could use a more accurate estimator assuming uniformity within each entry $e \in s$, similar to Lemma 1. However, the derived expression would be more complex and more expensive to evaluate, because now we need the average distance from $s.c$ (as opposed to the center $e.c$ of the entry's MBR). The overall TPAQ-MA algorithm is shown in Figure 10.

In the example of Figure 9, the partitioning level contains entries $E = \{N_3, N_4, N_5, N_6, N_7, N_8, N_9\}$. The binary search considers values of $|R|$ between 1 and 7. Starting with $|R| = (1 + 7)/2 = 4$, the algorithm creates S with 4 slots, as shown in

Algorithm **TPAQ-MA**(P, T)
1. Initialize an empty $list$
2. Set E = set of the entries at the topmost level with MPD $\leq T$
3. $low = 1$; $high = |E|$
4. While $low \leq high$ do
5. $mid = (low + high)/2$
6. Group the entries in E into mid slots
7. S = the set of created slots
8. If $\bar{C}(S) < T$, set $high = mid$
9. Else, set $low = mid$
10. For each $s \in S$ do
11. Perform a NN search at $s.c$ on the points under $s.E$
12. Append the retrieved point to $list$
13. Return $list$

Fig. 10. The *TPAQ-MA* algorithm

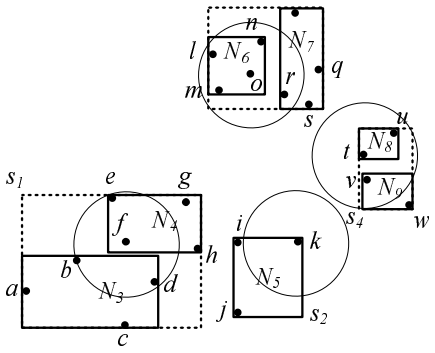


Fig. 11. Entries and final slots

Figure 11. It computes $\bar{C}(S)$, which is lower than T. It recursively continues the search for $|R| \in [1, 4]$ in the same way, and finally decides that $|R| = 4$ yields a value of $\bar{C}(S)$ that best approximates T. Next, TPAQ-MA performs a NN search at the center $s.c$ of the slots corresponding to $|R| = 4$, and returns the retrieved points (f, i, o, and k) as the result.

6 Experimental Evaluation

In this section we evaluate the performance of the proposed methods for k-medoid and medoid-aggregate queries. We use both synthetic and real datasets. The synthetic ones (SKW) follow a zipf distribution with parameter $\alpha = 0.8$, and have cardinality 256K, 512K, 1M, 2M and 4M points. The real datasets are (i) NA, with 569,120 points (available at *www.maproom.psu.edu/dcw*), and (ii) LA, with 1,314,620 points (available

at *www.rtreeportal.org*). All datasets are normalized to cover the same space with extent $10^4 \times 10^4$ and indexed by an R*-tree [1] (the block size ranges between 1 and 4Kbytes). For the experiments we use a Pentium 3GHz CPU.

6.1 *k*-Medoid Queries

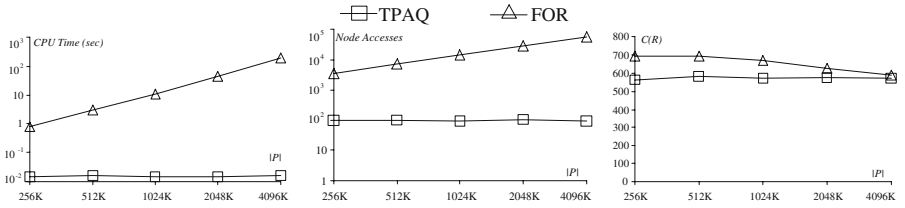
First, we focus on k-medoid queries and compare TPAQ against FOR (which, as discussed in Section 2.2, is the only other method that utilizes R-trees for computing k-medoids). For TPAQ, we use the depth-first algorithm of [3] to retrieve the nearest neighbor of each computed centroid. In the case of FOR we have to set the parameters *numlocal* (number of restarts) and *maxneighbors* (sample size of the possible neighbor sets) of the CLARANS component. Ester et al. [8] suggest setting *numlocal* = 2 and *maxneighbors* = $k \cdot (M - k)/800$, where M is the number of leaf nodes in the R-tree of P. With these parameters, FOR does not terminate within reasonable time for our datasets. Therefore, we set *maxneighbors* = $k \cdot (M - k)/(8000 \cdot \log M)$ and keep *numlocal* = 2. These values speed up FOR considerably, while the deterioration of the resulting solutions (with respect to the suggested values of *numlocal* and *maxneighbors*) is negligible. One final remark concerning FOR, is that all results presented in this section are average values over 10 runs of the algorithm. This is necessary because the performance of FOR depends on the random choices of its CLARANS component. The algorithms are compared for different data cardinality $|P|$, number of medoids k and block size. Table 2 summarizes the parameters under investigation along with their ranges and default values. In each experiment we vary a single parameter, while setting the remaining ones to their default (median) values.

Table 2. Default parameter values

Parameter	Range	Default
$\|P\|$	256K – 4M	1M
k	2 – 512	32
Block size	1KB – 4KB	2KB

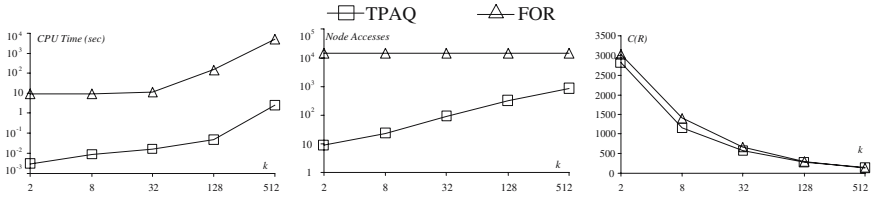
The first set of experiments measures the effect of $|P|$. In Figure 12(a), we show the running time of TPAQ and FOR, for SKW when $k = 32$ and $|P|$ varies between 256K and 4M. TPAQ is 2 to 4 orders of magnitude faster than FOR. Even for $|P| = 4M$ objects, our method terminates in less than 0.04 seconds (while FOR needs more than 3 minutes). Figure 12(b) shows the I/O cost (number of node accesses) for the same experiment. FOR is around 2 to 3 orders of magnitude more expensive than TPAQ since it reads the entire dataset once. Both the CPU and the I/O costs of TPAQ are relatively stable and small, because partitioning takes place at a high level of the R-tree.

The cost improvements of TPAQ come with no compromise of the answer quality. Figure 12(c) shows the average distance $C(R)$ achieved by the two algorithms. TPAQ outperforms FOR in all cases. An interesting observation is that the average distance for FOR drops when the cardinality of the dataset $|P|$ increases. This happens because higher $|P|$ implies more possible "paths" to a local minimum. To summarize, the results



(a) SKW–CPU time (b) SKW–Node accesses (c) SKW–Average distance

Fig. 12. Performance versus $|P|$



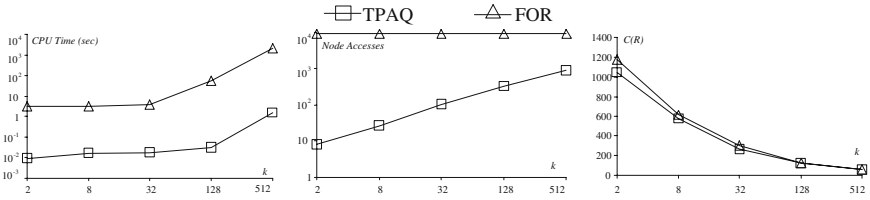
(a) SKW–CPU time (b) SKW–Node accesses (c) SKW–Average distance

Fig. 13. Performance versus k (synthetic data)

of Figure 12 verifies that TPAQ scales gracefully with dataset cardinality and incurs much lower cost than FOR, without sacrificing the medoid quality.

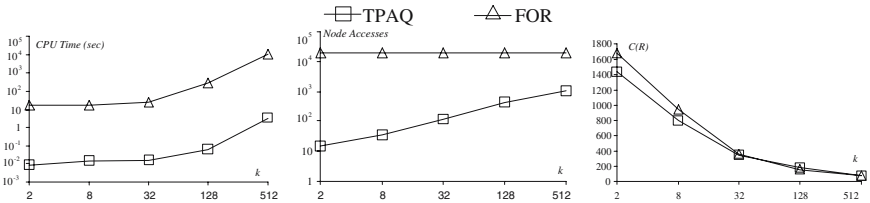
The next set of experiments studies the performance of TPAQ and FOR when k varies between 2 and 512, using a SKW dataset of cardinality $|P| = 1M$. Figure 13(a) compares the running time of the methods. In both cases, TPAQ is 3 orders of magnitude faster than FOR. It is worth mentioning that for $k = 512$ our method terminates in 2.5 seconds, while FOR requires around 1 hour and 20 minutes. For $k = 512$, both the partitioning into slots of TPAQ and the CLARANS component of FOR are applied on an input of size 14,184; the input of the TPAQ partitioning algorithm consists of the extended entries at the leaf level, while the input of CLARANS is the set of actual representatives retrieved in each leaf node. The large difference in CPU time verifies the efficiency of our partitioning algorithm.

Figure 13(b) shows the effect of k on the I/O cost. The node accesses of FOR are constant and equal to the total number of nodes in the R-tree of P (i.e., 14,391). On the other hand, TPAQ accesses more nodes as k increases. This happens because (i) it needs to descend more R-tree levels in order to find one with a sufficient number (i.e., k) of entries, and (ii) it performs more NN queries (i.e., k) at the final step. However, TPAQ is always more efficient than FOR; in the worst case TPAQ reads all R-tree nodes up to level 1 (this is the situation for $k = 512$), while FOR reads the entire dataset P for any value of k. Figure 13(c) compares the accuracy of the methods. TPAQ achieves lower $C(R)$ for all values of k.



(a) NA–CPU time (b) NA–Node accesses (c) NA–Average distance

Fig. 14. Performance versus k (NA)

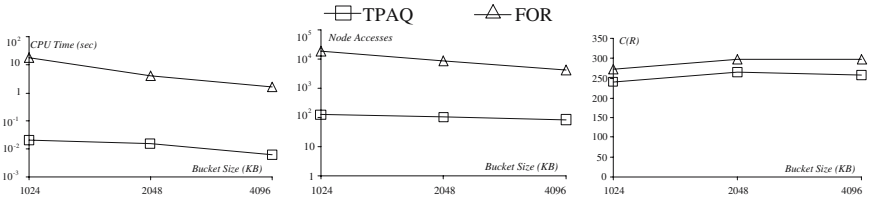


(a) LA–CPU time (b) LA–Node accesses (c) LA–Average distance

Fig. 15. Performance versus k (LA)

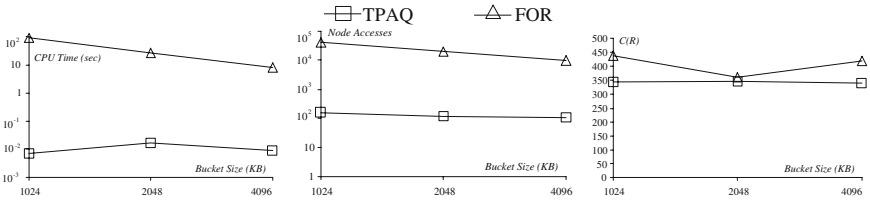
In order to confirm the generality of our observations, Figures 14 and 15 repeat the above experiment for real datasets NA and LA. TPAQ outperforms FOR by orders of magnitude in terms of both CPU time (Figures 14(a) and 15(a) for NA and LA, respectively) and number of node accesses (Figures 14(b) and 15(b)). Regarding the average distance $C(R)$, the methods achieve similar results, with TPAQ being the winner. Note that the CPU and I/O costs of the methods are higher for LA (than NA), since it is larger and its R-tree has more entries per level. The achieved $C(R)$ values are lower for NA, because it is more skewed than LA (i.e., the objects are concentrated in a smaller area of the workspace).

Figures 16(a) and 17(a) show the running time of TPAQ and FOR on 32-medoid queries as a function of the block size for datasets NA and LA. When the block size increases, the number of leaf nodes drops. Thus the CPU cost of FOR decreases because its expensive CLARANS step processes fewer representatives. TPAQ does not necessarily follow the same trend. For NA, the running time drops, since the number of entries at the partitioning level is 618, 143 and 33 for block size 1KB, 2KB and 4KB, respectively. For LA the populations of the partitioning levels are 43, 313 and 77, respectively, yielding higher running time in the 2KB case. Concerning the I/O cost, larger block size implies smaller R-tree height, and fewer nodes per level. Therefore, both methods are less costly (as illustrated in Figures 16(b) and 17(b)). Independently of the block size, TPAQ incurs much fewer node accesses than FOR. Finally, Figures 16(c) and 17(c) illustrate the effect of the block size in the quality of the retrieved medoid sets. In all cases, the average distance achieved by TPAQ is lower than that of FOR.



(a) NA–CPU time (b) NA–Node accesses (c) NA–Average distance

Fig. 16. Performance versus block size (NA)



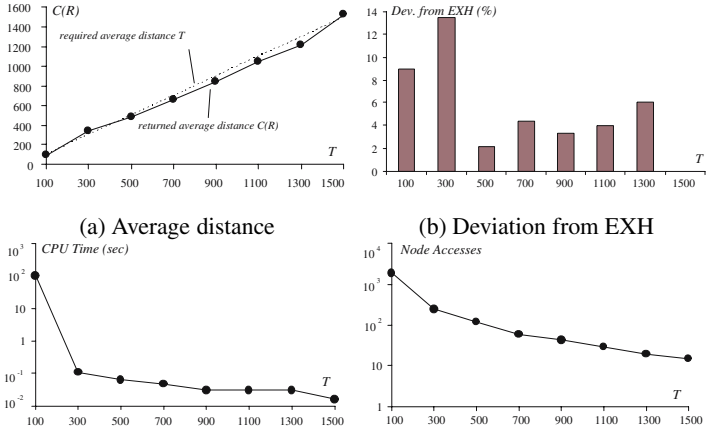
(a) LA–CPU time (b) LA–Node accesses (c) LA–Average distance

Fig. 17. Performance versus block size (LA)

6.2 Medoid-Aggregate Queries

In this section we study the performance of TPAQ-MA. We use datasets SKW (with 1M objects) and LA, and vary T in the range from 100 to 1500 (recall that our datasets cover a space with extent $10^4 \times 10^4$). Since there is no existing algorithm for processing such queries on large indexed datasets, we compare TPAQ-MA against an exhaustive algorithm (EXH) that works as follows. Let E be the set of entries at the partitioning level of TPAQ-MA; then, EXH computes and evaluates all the medoid sets for $|R| = 1$ up to $|R| = |E|$, by performing partitioning of E into slots with the technique presented in Section 4. EXH returns the medoid set that yields the closest average distance to T. Note that EXH is prohibitively expensive in practice because, for each examined value of $|R|$, it scans the entire dataset P in order to exactly evaluate $C(R)$. Therefore, we exclude EXH from the CPU and I/O cost charts.

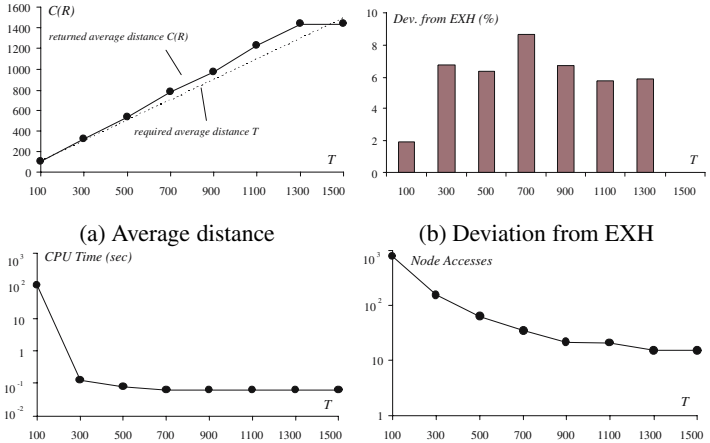
Our evaluation starts with SKW. Figure 18(a) shows the $C(R)$ for TPAQ-MA versus T. Clearly, the average distance returned by TPAQ-MA approximates the desired distance (dotted line) very well. Figure 18(b) plots the deviation percentage between the average distances achieved by TPAQ-MA and EXH. The deviation is below 9% in all cases, except for $T = 300$ where it equals 13.4%. Interestingly, for $T = 1500$, TPAQ-MA returns exactly the same result as EXH with $|R| = 5$. Figures 18(c) and 18(d) illustrate the running time and the node accesses of our method, respectively. For $T = 100$, both costs are relatively high (100.8 seconds and 1839 node accesses) compared to larger values of T. The reason is that when $T = 100$, partitioning takes place



(a) Average distance (b) Deviation from EXH

(c) CPU time (d) Node accesses

Fig. 18. Performance versus T (SKW)



(a) Average distance (b) Deviation from EXH

(c) CPU time (d) Node accesses

Fig. 19. Performance versus T (LA)

at level 1 (leaf level, which contains 14,184 entries) and returns $|R| = 1272$ medoids, incurring many computations and I/O operations. In all other cases, partitioning takes place at level 2 (containing 203 entries), and TPAQ-MA runs in less than 0.11 seconds and reads fewer than 251 pages.

Figure 19 repeats the above experiment for the LA dataset. Figures 19(a) and 19(b) compare the average distance achieved by TPAQ-MA with the input value T and the result of EXH, respectively. The deviation from EXH is always smaller than 8.6%,

while for $T = 1500$ the answer of TPAQ-MA is the same as EXH. Concerning the efficiency of TPAQ-MA, we observe that the algorithm has, in general, very low CPU and I/O cost. The highest cost is again in the case of $T = 100$ for the reasons explained in the context of Figure 18; TPAQ-MA partitions 19,186 entries into slots and extracts $|R| = 296$ medoids, taking in total 105.6 seconds and performing 781 node accesses.

7 Conclusion

This paper studies k-medoids and related problems in large databases. In particular, we consider k-medoid and medoid-aggregate (MA) queries, and propose TPAQ (*Tree-based PArtition Querying*), a framework for their efficient processing. TPAQ provides high-quality answers almost instantaneously, thus facilitating data analysis, especially in time-critical resource allocation applications. Our techniques are the first ones to fully exploit the data partitioning properties of an already existing spatial access method on the dataset. TPAQ processes a query in three steps. Initially, it descends the index, and stops at the topmost level that provides sufficient information about the underlying data distribution. Next, it partitions the entries of the selected level into a number of slots. In the case of k-medoid queries, the number of slots is equal to k. For MA, this number is decided using binary search in conjunction with some average distance estimators. Finally, TPAQ retrieves one medoid for each slot with a NN query therein. An extensive experimental evaluation shows that TPAQ outperforms the state-of-the-art method for k-medoid queries by orders of magnitude, and achieves results of better or comparable quality. Our empirical study also illustrates the effectiveness and efficiency of TPAQ for processing MA queries.

The quality of the medoid sets returned by TPAQ is determined by the achieved average distance. An interesting direction for future work is to extend our index-based strategies to other aggregate distance functions, such as *max*. In the *max* case, we wish to minimize the maximum distance between the points in the input dataset and their closest medoid; i.e., $C(R) = \max_{p \in P} ||p - r(p)||$. Further, it is also interesting to solve constrained partitioning queries. For example, consider that each facility can serve up to a maximum number of clients. In this case the algorithms must be extended to take into account existing capacity (or processing capability) constraints.

References

1. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD. (1990) 322–331
2. Theodoridis, Y., Stefanakis, E., Sellis, T.K.: Efficient cost models for spatial queries using r-trees. IEEE TKDE **12** (2000) 19–32
3. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD. (1995) 71–79
4. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM TODS **24** (1999) 265–318
5. Arora, S., Raghavan, P., Rao, S.: Approximation schemes for euclidean k-medians and related problems. In: STOC. (1998) 106–113

6. Kaufman, L., Rousseeuw, P.: Finding groups in data. Wiley-Interscience (1990)
7. Ng, R.T., Han, J.: Efficient and effective clustering methods for spatial data mining. In: VLDB. (1994) 144–155
8. Ester, M., Kriegel, H.P., Xu, X.: A database interface for clustering in large spatial databases. In: KDD. (1995) 94–99
9. Ester, M., Kriegel, H.P., Xu, X.: Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In: SSD. (1995) 67–82
10. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD. (1996) 226–231
11. Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: An efficient data clustering method for very large databases. In: SIGMOD. (1996) 103–114
12. Guha, S., Rastogi, R., Shim, K.: CURE: An efficient clustering algorithm for large databases. In: SIGMOD. (1998) 73–84
13. Ankerst, M., Breunig, M.M., Kriegel, H.P., Sander, J.: OPTICS: Ordering points to identify the clustering structure. In: SIGMOD. (1999) 49–60
14. Hartigan, J.A.: Clustering algorithms. Wiley (1975)
15. Hastie, T., Tibshirani, R., Friedman, J.: The elements of statistical learning. Springer-Verlag (2001)
16. Pelleg, D., Moore, A.W.: Accelerating exact k-means algorithms with geometric reasoning. In: KDD. (1999) 277–281
17. Pelleg, D., Moore, A.W.: X-means: Extending K-means with efficient estimation of the number of clusters. In: ICML. (2000) 727–734
18. Hamerly, G., Elkan, C.: Learning the k in k-means. In: NIPS. (2003)
19. Fayyad, U., Piatetsjy-Shapiro, G., Smyth, P., Uthurusamy, R.: Advances in knowledge discovery and data mining. AAAI/MIT (1996)
20. Kamel, I., Faloutsos, C.: On packing r-trees. In: CIKM. (1993) 490–499
21. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the clustering properties of the hilbert space-filling curve. IEEE TKDE **13** (2001) 124–141
22. Lo, M.L., Ravishankar, C.V.: Generating seeded trees from data sets. In: SSD. (1995) 328–347
23. Lo, M.L., Ravishankar, C.V.: The design and implementation of seeded trees: An efficient method for spatial joins. IEEE TKDE **10** (1998) 136–152
24. Mamoulis, N., Papadias, D.: Slot index spatial join. IEEE TKDE **15** (2003) 211–231