Kyriakos Mouratidis[1], Dimitris Papadias[2], Spiros Papadimitriou[3]

Tree-based Partition Querying: A Methodology for Computing Medoids in Large Spatial Datasets

Abstract

Besides traditional domains (e.g., resource allocation, data mining applications), algorithms for medoid computation and related problems will play an important role in numerous emerging fields, such as location based services and sensor networks. Since the *k*-medoid problem is NP hard, all existing work deals with approximate solutions on relatively small datasets. This paper aims at efficient methods for very large spatial databases, motivated by: (i) the high and ever increasing availability of spatial data, and (ii) the need for novel query types and improved services. The proposed solutions exploit the intrinsic grouping properties of a data partition index in order to read only a small part of the dataset. 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). In addition, we study *medoid-aggregate* queries, where *k* is not known in advance, but we are asked to compute a medoid set that leads to an average distance close to a user-specified value. Similarly, *medoid-optimization* queries aim at minimizing both the number of medoids *k* and the average distance. We also consider the *max* version for the aforementioned problems, where the goal is to minimize the maximum (instead of the average) distance between any object and its closest medoid. Finally, we investigate *bichromatic* and *weighted* medoid versions for all query types, as well as, *maximum capacity* and *dynamic* medoids.

Keywords: Spatial databases, Query processing, Medoid queries

1. Introduction

Consider 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. Efficient solutions to medoid queries are essential in several applications related to resource allocation and spatial decision making. Since the problem is NP-hard [GJ79], research has focused on approximate algorithms. Despite a bulk of methods for small and moderate size datasets, currently there exists no technique applicable to very large databases.

More formally, given a set P of points, we wish to find a set of medoids $R \subseteq P$ with cardinality k that minimizes the average (*avg*) Euclidean distance $||p-r(p)||$ between each point $p \in P$ and its closest medoid $r(p) \in R$. Equivalently, 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 1.1 shows an example, where the points of P are residential blocks, $k = 3$ and $R = \{h, o, t\}$. The three medoids h, o, t are 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.

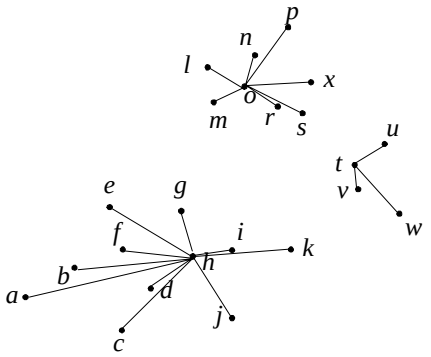


Figure 1.1: Example *k*-medoid query

In addition to conventional queries, we introduce and solve several alternative forms of the *k*-medoid problem with practical relevance. In *medoid-aggregate* (MA) queries, the value of *k* is not known in advance, but the goal is to select a minimal set R of medoids, such that $C(R)$ best approximates an

[1] *Singapore Management University*
kyriakos@smu.edu.sg
[2] *Hong Kong University of Science and Technology*
dimitris@cs.ust.hk
[3] *IBM T.J. Watson Research Center*
spapadim@us.ibm.com

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. A *medoid-optimization* (MO) query asks for the minimal medoid set that achieves the smallest value of a function f, which is monotonically increasing with both the number of medoids $|R|$ and the resulting $C(R)$. In the running example, assume that we also take into account the construction cost of each branch and we wish to minimize function $f(C(R), |R|) = C(R) + 5 \cdot |R|$ in order to achieve the best tradeoff between the number of branches and the average distance.

Interesting variants of the above three query types arise when the quality of a medoid set is determined by the maximum distance between the input points and their closest medoid; i.e., when $C(R) = max_{p \in P} ||p\text{-}r(p)||$. For instance, in the franchise example our goal may be to minimize the maximum distance between the residential blocks and their closest branches, potentially achieving a desired $C(R)$ with the minimal set of branches (MA), or minimizing a cost function (MO). Furthermore, all the above query types can be extended to their *bichromatic* versions, where the candidate medoids belong to a dataset different from that of the data points, e.g., there is a distinct set of potential branch sites. In the *weighted* version of the problem, each data point (e.g., residential block) is assigned a numeric weight indicating its importance (e.g., depending on the number of its residents). Another interesting instance is the *maximum capacity* medoids, where each medoid (e.g., branch) can serve up to a maximum number of data points (e.g., blocks). Finally, in the *dynamic* version of the problem, dataset P receives point insertions and deletions, and our task is to maintain the medoid set without re-computation from scratch.

In addition to resource allocation and data mining, medoid queries arise in a wide variety of modern applications including mobile computing and sensor networks. For example, consider a number of users accessing a location based service through their mobile devices (cellular phones, PDAs). To save communication cost, the devices select super-nodes among them, which collect, aggregate and forward to the location server messages received from their vicinity. Due to the error prone nature of the wireless medium, the devices should be close to some super-node. Therefore, selecting super-nodes is actually a medoid computation task. If the number of super-nodes is fixed, then this is a k-medoid problem. On the other hand, if packet loss/signal attenuation is unacceptably high when the communication range exceeds T distance units, then the case corresponds to an MA query.

Medoid queries also arise in the field of sensor networks. Typically, in order to prolong the battery life, only a fraction of the sensors are kept awake, and used as representatives for a particular region of the monitored area [XWZ+05]. If the application requires that only k sensors should be awake, then the best representatives are the k medoids. On the other hand, if the sensing range of each unit is T, then a MA query returns a set of representatives that roughly cover the entire monitored area. Since the sensing coverage of the area essentially determines the accuracy of the acquired measurements, there exists a tradeoff between the number of sensors that stay awake and the achieved accuracy. In this case, a MO query with an appropriately selected cost function f, computes the optimal number of representatives and their locations.

In this paper, we propose TPAQ (*Tree-based PArtition Querying*), a methodology that can efficiently process all the above query types. TPAQ avoids reading the entire dataset by exploiting the grouping properties of a data partition method on P. It 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 general framework of TPAQ. Section 4 considers k-medoid queries. Sections 5 and 6 focus on MA and MO queries, respectively. Section 7 discusses the application of TPAQ to bichromatic, weighted and other related query types. Section 8 presents experimental results using both real and synthetic datasets. Finally, Section 9 concludes the paper.

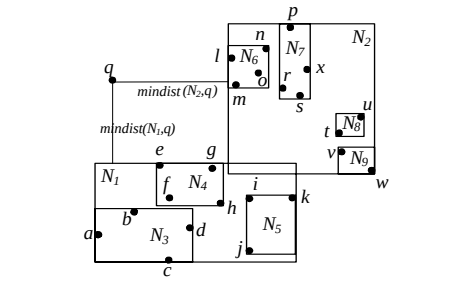
2. Background

Although our techniques can be used with any data partition method, here we assume R*-trees [BKSS90] 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.

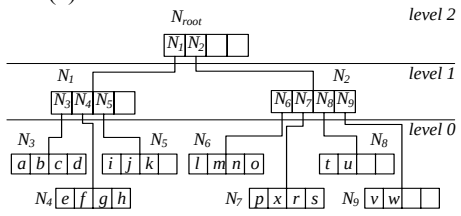
2.1. R-trees and Nearest Neighbor Search

We illustrate our examples with the R*-tree of Figure

2.1, containing the data points of Figure 1.1, 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 proximity, 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 achieve minimal overlap among nodes of the same level [TSS00]. 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. Furthermore, the R*-tree is a standard index for spatial query processing. Specialized structures may yield solutions of better quality for *k*-medoid problems, but would have limited applicability in existing systems, where R-trees are prevalent.



(a) R-tree node extents and locations



(b) R-tree data structure

Figure 2.1: R-tree example

With few exceptions (discussed in the next subsection), the R-tree family of indexes has been used exclusively for 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 2.1 shows the *mindist* between *q* and nodes N_1 and N_2. The algorithm of [RKV95] traverses the tree in a depth-first 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 [HS99] 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 [ARR98]. Most of this work, however, is largely theoretical in nature. Kaufmann and Rousseeuw [KR90] propose *partitioning around medoids* (PAM), a practical 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*|. *Clustering large applications* (CLARA) [KR90] alleviates the problem by generating random samples from *P* and executing PAM on those. Ng and Han [NH94] 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

[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* [ARR98].

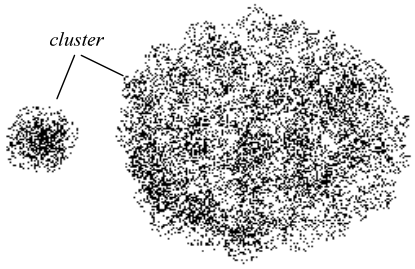
R-tree, Ester et al. [EKX95a, EKX95b] develop *focusing on representatives* (FOR). FOR takes the most centrally located point of each leaf node and forms a sample set, which is considered as representative of the entire set P. Then, it applies CLARANS on this sample to find the k medoids. Although FOR 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 [EKX95a] use R-trees with only 50,559 points and 1,027 leaf nodes).

Regarding the *max* case, to the best of our knowledge, there does not exist any method for disk-resident data. For in-memory processing, the method of [G85] answers *max* k-medoid queries in $O(k \cdot |P|)$ time with an approximation factor of 2. In other words, the returned medoid set is guaranteed to achieve a maximum distance $C(R)$ that is no more than two times larger than the optimal one. The algorithm proceeds as follows. The first medoid is randomly selected from P and forms set R_1. The second medoid is the point in P that lies furthest from the point in R_1. These two medoids form R_2. In general, the ith medoid is the one that has the maximum distance from any point in R_{i-1}. Finally, the set R_k is returned as the result. The algorithm is simple and works well in practice. However, its adaptation for large datasets would be very expensive in terms of both CPU and I/O cost, since in order to find the ith medoid it has to scan the entire dataset and compute the distance between every data point and all elements of R_{i-1}.

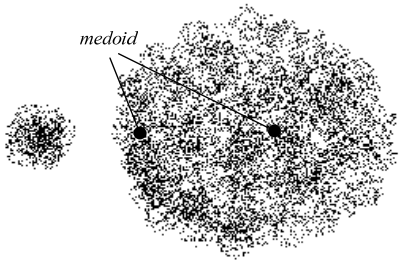
A problem related to k-medoids is *min-dist optimal-location* (MDOL) computation. Given a set of data points P, a set of existing facilities, and a user-specified spatial region Q (i.e., range for a new facility), a MDOL query computes the location in Q which, if a new facility is built there, minimizes the average distance between each data point and its closest facility. The main difference with respect to k-medoids is that the output of a MDOL query is a single point (as opposed to k) that does not necessarily belong to P, but it can be anywhere in Q. Zhang et al. [ZDXT06] propose an exact method for this problem. This method is complementary to the proposed algorithms since it can be used to increase the cardinality of an existing medoid set, when there is a need for incremental processing (e.g., a franchise chain decides to add a new branch in a given area).

The k-medoid problem is related to clustering. Clustering methods designed for large databases include DBSCAN [EKSX96], BIRCH [ZRL96], CURE [GRS98] and OPTICS [ABKS99]. 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 to points in other groups. Figure 2.2a depicts a 2-way clustering for a dataset, while Figure 2.2b shows the two medoids in the *avg* case. Clearly, assigning a facility per cluster would not achieve the purpose of minimizing the average distance between points and facilities. Furthermore, the number of clusters depends on the data characteristics, whereas the number of medoids is an input parameter determined by the application requirements.

Extensive work on medoids and clustering has been carried out in the areas of statistics [H75, KR90, HTF01], machine learning [PM99, PM00, HE03] and data mining [EKSX96, FPSU96]. However, the focus there is on assessing the statistical quality of a given clustering, usually based on assumptions about the data distribution [HTF01, KR90, PM00, HE03]. Only few approaches aim at dynamically discovering the number of clusters [PM00, HE03]. 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 several types of medoid queries.



(a) 2 clusters



(b) 2 medoids

Figure 2.2: Clustering versus medoids problem

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 and MO queries, the selection of the partitioning level is also based on the spatial extents and (in the *avg* case) on 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 slotting algorithm. For MA and MO, multiple calls of the slotting algorithm 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. We focus on un-weighted, monochromatic queries, i.e., all data points have the same importance (i.e., unit weight) and each medoid is a data point. The extension to bichromatic, weighted and other queries is discussed in Section 7.

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 sub-tree 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,

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

In the *avg* case, vector c is the weighted center of s,

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

In the *max* case, vector c is the center of the *minimum enclosing circle* of all the entry centers $e.c$ in s; i.e., c is the center of the circle enclosing $e.c$ $\forall e \in E$ that has the minimum possible radius.

A fundamental operation is the insertion of an extended entry e into a slot s. The pseudo-code for this function in the *avg* case is shown in Figure 3.1. 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 $e.c$. In the *max* case, the new slot center is computed as the center of the minimum circle enclosing $e.c$ and all the entry centers currently in s. We use the incremental algorithm of [W91] that finds the new slot center in expected constant time.

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. $s.E = s.E \cup \{e\}$

Figure 3.1: The *InsertEntry* function for *avg*

In the subsequent sections, we describe the algorithmic details for each query type. For every considered medoid problem, we first present the *avg* case, followed by *max*. Note that, similar to PAM, CLARA, CLARANS and FOR, TPAQ aims at efficient processing without theoretical guarantees on the quality of the medoid set. Meaningful quality bounds are impossible because TPAQ is based on the underlying R-trees, which are heuristic-based structures. Nevertheless, as we show in the experimental evaluation, TPAQ computes medoid sets that are better than those of the existing methods at a small fraction of the cost (usually several orders of magnitude faster). Furthermore, it is more general in terms of the problem variants it can process. Table 3.1 summarizes the frequently used symbols.

Symbol	Description
P	Set of data points
$\|p_1\text{-}p_2\|$	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/maximum 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 $s_j = \langle c_j, w_j, E_j \rangle$

Table 3.1: Frequently used symbols

4. *k*-Medoid Queries

Given an *avg* *k*-medoid query, TPAQ finds the top-most level with $k' \geq k$ entries. For example, if $k = 3$ in the tree of Figure 2.1, 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, $w_1 = w_2 = N/2 = 11.5$ (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 4.1 illustrates the algorithm for computing the initial set of entries. Note that *InitEntries* assumes that k does not exceed the number of leaf nodes. This is not restrictive because the lowest level typically contains several thousand nodes (e.g., in our datasets, between 3,000 and 60,000), which is sufficient for all ranges of k that are of practical interest. If needed, larger

values of k can be accommodated by splitting leaf level nodes.

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*

Figure 4.1: The *InitEntries* function

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 a singleton 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 [KF93, BJFS01]. Therefore, we first Hilbert-sort the k' entries and select every m^{th} entry as a seed, where $m = k'/k$. This procedure is fast and produces well-spaced seeds that follow the data distribution. Returning to our example, Figure 4.2 shows the level 1 MBRs (for the R-tree of Figure 2.1) 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.

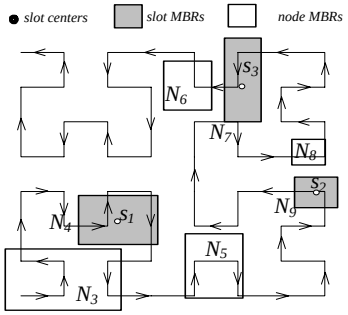
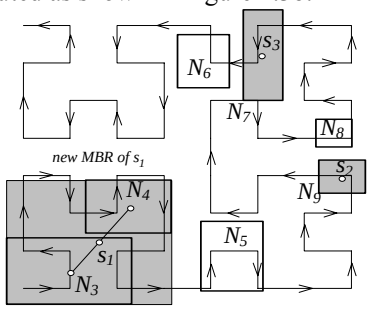
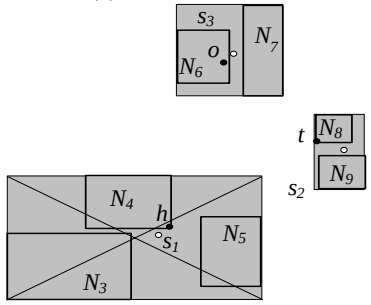


Figure 4.2: Hilbert seeds on example dataset

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 3.1. The center of s_1 is updated to the midpoint of N_3 and N_4's centers, as shown in Figure 4.3a. TPAQ proceeds in this manner, until the final slots and weighted centers are computed as shown in Figure 4.3b.



(a) Insertion of N_3



(b) Final slot contents

Figure 4.3: Insertion of entries into slots

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 node accesses and negligible CPU cost. Observe that an actual medoid (i.e., a point in P that minimizes the average distance) 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 4.4 shows the complete TPAQ k-medoid computation algorithm. 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 [LR95, LR98, MP03]. 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=∅, 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·|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*

Figure 4.4: The *TPAQ* algorithm

TPAQ follows similar steps for the *max* case. The function *InitEntries* proceeds as before, but without computing the expected cardinality for entries and slots; in the *max* version of the problem, we use only the geometric centroids of the R-tree entries. We apply the algorithm of [G85], discussed in Section 2.2, to select seeds. In particular, if E is the set of entries in the partitioning level, we compute the k-medoids over their centers $e.c$. Then, we insert the remaining entries in E one by one into the slot with the closest center. Finally, we perform a NN search at the center of each slot to retrieve the actual corresponding medoid. Recall that the center of each slot is the center of the minimum circle enclosing its entries' centers. Returning to our running example, if a 3-medoid query is given in the tree of Figure 2.1, level 1 is chosen as the partitioning level. Among the entries of level 1, assume that the algorithm of [G85] returns the centers of N_4, N_6 and N_9 as the seeds. The insertion of the remaining entries into the created slots (s_1, s_2, and s_3) results in the partitioning shown in Figure 4.5.

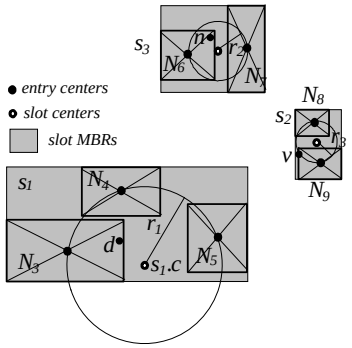


Figure 4.5: A 3-medoid query in the *max* case

The three circles correspond to the minimum circles enclosing the centers of nodes in each slot. The final step of the TPAQ algorithm retrieves the NNs of $s_1.c$, $s_2.c$, and $s_3.c$, which are points d, v and n, respectively. The returned medoid set is R={d, v, n}.

5. Medoid-Aggregate Queries

A medoid-aggregate (MA) query specifies the desired distance T (between points and medoids), and asks for the minimal medoid set R that achieves $C(R) = T$. Consider the example of Figure 5.1 for the *avg* case, and assume that we know a priori all the optimal i-medoid sets R^i and the corresponding $C(R^i)$, for i=1,...,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 both the *avg* and the *max* case. TPAQ-MA first descends the R-tree of P down to an appropriate partitioning level. Next, it estimates the value of |R| that achieves the average distance $C(R)$ closest to T and returns the corresponding medoid set R.

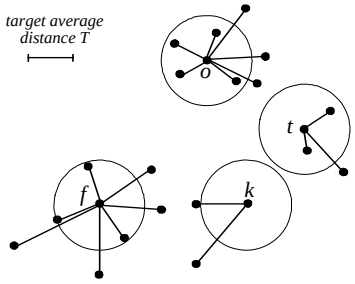


Figure 5.1: MA query example in the *avg* case

Consider first the *avg* case. The initial step of TPAQ-MA is to determine the partitioning level. The algorithm selects for partitioning the top-most level whose *minimum possible distance* (MPD) does not exceed 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 of 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[2] in its MBR, and that the corresponding medoid is at $e.c$. The average distance $\overline{C}(e)$ between $e.c$ and the points in e is given by the following lemma.

[2] This is a reasonable assumption for low-dimensional R-trees [TSS00].

Lemma 5.1: 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.

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 5.1.

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

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

TPAQ-MA applies the *InitEntries* function to select the top-most level that has MPD ≤ T. The pseudo-code of *InitEntries* is the same as shown in Figure 4.1, 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=$\bar{C}(N_{root})$, which is 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 N_3 to N_9. The MPD is less than T, and level 1 is selected for partitioning. *InitEntries* 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 best 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 $|E|$. TPAQ-MA assumes that $C(R)$ decreases as $|R|$ increases[3], and performs binary search in order to find the value of $|R|$ that yields the average distance closest 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 5.1. 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 5.2.

Algorithm **TPAQ-MA** (P, T)
1. Initialize an empty *list*
2. Set E = set of the entries at the topmost level with MPD≤T
3. *low*=1; *high*=$|E|$
4. while *low*≤*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*

Figure 5.2: The *TPAQ-MA* algorithm

In the example of Figure 2.1, 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 Figure 5.3. 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 decides that $|R|=4$ yields a value of $\bar{C}(S)$ that best approximates T. Finally, similar to TPQA, TPAQ-MA performs a NN search at the center $s.c$ of the slots corresponding to $|R|=4$, and returns the retrieved points (f, k, t and o) as the result.

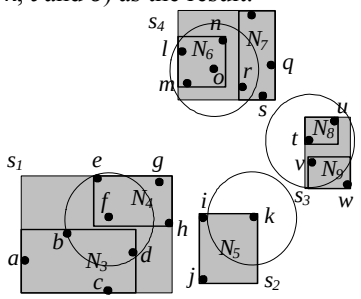


Figure 5.3: Entries and final slots

[3] Although this assumption is true for optimal medoid sets, it may not always hold for approximate solutions, in which case TPAQ-MA may be trapped in a local minimum. Nevertheless, violations of the assumption occur in very large medoid sets and do not have a significant effect on the quality of the solution.

Consider now the *max* version of the MA problem. *InitEntries* chooses for partitioning the top-most level with minimum possible distance (MPD) less than or equal to T. The MPD of a level is an estimated upper bound for the maximum distance $C(R)$, assuming that we return a medoid at the center of each of the level's entries. Given an R-tree entry e and assuming that we can find a medoid at $e.c$ (i.e., the crossing point of its MBR diagonals), then the maximum possible distance of any point in e from the medoid is half the MBR diagonal length. Therefore, the MPD of a level is computed as the half of the maximum entry diagonal in the level. In other words, $\overline{C}(e)=D/2$ (where D is the diagonal of e), and MPD=$max_{e \in E}\overline{C}(e)$ (where E is the set of entries in the given level).

Similar to the *avg* case, in order to determine the number of medoids that best approximate the target distance T, we perform binary search. If E is the set of entries in the partitioning level, then the candidate $|R|$ values range between 1 and $|E|$. For each considered $|R|$, we use the *max* slotting algorithm (described in Section 4). Let S be the set of slots for a value of $|R|$. To estimate the achieved $C(R)$ (i.e., to compute $\overline{C}(S)$), we assume that the maximum distance within each slot s equals the radius of the minimum circle enclosing the entry centers in s. For example, if level 1 is selected for partitioning and $|R|=3$, the slotting produces the grouping shown in Figure 4.5. $C(R)$ is estimated as the maximum radius of the three circles, that is, $\overline{C}(S)=max\{r_1, r_2, r_3\}=r_1$. Formally, if *MincircRadius*(s) is the radius of the smallest circle enclosing $e.c$ $\forall e \in s$, then $\overline{C}(S)$ = $max_{s \in S}$ *MincircRadius*(s). When the binary search terminates, we retrieve the medoids corresponding to the best value of $|R|$. The algorithm of Figure 5.2 directly applies to *max* MA queries, by using the *max* versions of MPD and $\overline{C}(S)$, and by implementing line 6 with the *max* slotting algorithm.

6. Medoid-Optimization Queries

In real-world scenarios, opening a facility has some cost. Thus, users may wish to find a good tradeoff between overall cost and coverage (i.e., the average or maximum distance between clients and their closest facilities). If the relative importance of these conflicting factors is given by a user-specified cost function $f(C(R), |R|)$, the aim of a MO query is to find the medoid set R that minimizes f. The TPAQ methodology applies to this problem, provided that f is increasing on both $C(R)$ and $|R|$. Consider the example of Figure 1.1 in the *avg* case, and let $f(C(R), |R|)$ be $C(R) + Cost_{pm} \cdot |R|$, where $Cost_{pm}$ is the cost per medoid. Assume that we know a priori all the optimal i-medoid sets R^i and the corresponding $C(R^i)$, for i=1,...,23. If the plot of $f(C(R^i), |R^i|)$ versus $|R^i|$ is as shown in Figure 6.1, then the optimal $|R|$ is 3 and the result of the query is {*h*, *o*, *t*} (as in Figure 1.1). TPAQ-MO is based on the observation that $f(C(R^i), |R^i|)$ has a single minimum. Hence, it applies a gradient descent technique to decide the partitioning level and the optimal number of medoids $|R|$.

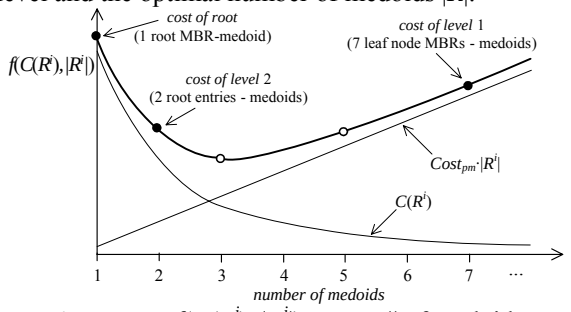
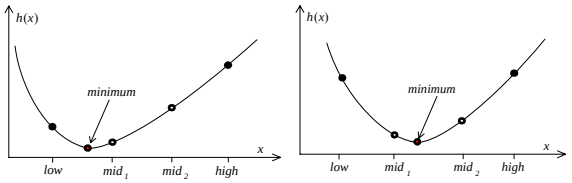


Figure 6.1: $f(C(R^i), |R^i|)$ versus # of medoids

In both the *avg* and *max* cases, TPAQ-MO initially descends the R-tree of P and for each candidate level, it computes its *cost*. We define the cost of a level as the value f(MPD, $|E|$), where E is the set of its entries. TPAQ-MO selects for partitioning the top-most level whose cost is greater than the cost of the previous one (i.e., at the first detected increase in the curve of Figure 6.1). If the MPD estimations are accurate, then the medoid set that minimizes f has size $|R|$ between 1 and $|E|$ (the number of entries at the partitioning level). The traversal of the R-tree down to the appropriate level is performed by the *InitEntries* function of Figure 4.1 by modifying the while-condition in line 3 to: "the cost of the current level is less than the cost of the previous one". In Figure 2.1, *InitEntries* compares the costs of the root entry (1 medoid) and level 2 (two medoids – one for each root entry). Since the cost of level 2 is less than that of the root, it proceeds with level 1, whose cost is larger than that of level 2. Thus, level 1 is selected for partitioning and *InitEntries* returns the set of extended entries from N_3 to N_9.

Given the set of entries E at the partitioning level, the next step of TPAQ-MO is to compute the optimal value for $|R|$, which lies between 1 and $|E|$. To perform this task, TPAQ-MO uses a gradient descent method which considers $O(\log_{3/2}|E|)$ different values for $|R|$. Consider the example of Figure 6.2, where we want to find the value $x_{opt} \in [low, high]$ that minimizes a given function $h(x)$. We split the search interval into three equal sub-intervals, defined by $mid_1=(2 \cdot low+high)/3$ and $mid_2=(low+2 \cdot high)/3$. Next, we compute $h(mid_1)$ and $h(mid_2)$. Assuming that $h(mid_1) < h(mid_2)$, we distinguish two cases; either $x_{opt} \in [low, mid_1]$ (as shown in Figure 6.2a), or $x_{opt} \in [mid_1, mid_2]$ (Figure 6.2b). In other words, the search interval is restricted to $[low, mid_2]$.

Symmetrically, if $h(mid_1) > h(mid_2)$, then the search interval becomes $[mid_1, high]$. Otherwise, if $h(mid_1) = h(mid_2)$, the search is restricted to interval $[mid_1, mid_2]$. The x_{opt} can be found by recursively applying the same procedure to the new search interval. If x_{opt} is an integer, then the search terminates in $O(\log_{3/2}(high\text{-}low))$ steps.



(a) $x_{opt} \in [low, mid_1]$ (b) $x_{opt} \in [mid_1, mid_2]$

Figure 6.2: Computing the minimum of a function h

We use the above technique to determine the optimal value of $|R|$, starting with low=1 and $high$=$|E|$. For each considered $|R|$, we compute the set of slots S in the way presented in Section 4, and estimate the corresponding $C(R)$ as the quantity $\bar{C}(S)$ discussed in Section 5. The gradient descent method returns the value of $|R|$ that minimizes $f(\bar{C}(S), |R|)$. Finally, the result of TPAQ-MO is the set of points retrieved by a NN search at the center of each slot $s \in S$ of the corresponding partitioning. Figure 6.3 shows the overall TPAQ-MO algorithm. The algorithm works for both *avg* and *max* MO queries, by using the corresponding MPD and $\bar{C}(S)$ functions, and the appropriate slotting algorithms. In our running example, for the *avg* case, level 1 is the partitioning level, and $|R|$=3 is selected as the best medoid set size. The slots and the returned medoids (i.e., h, o and t) are the same as in Figure 4.3.

Algorithm **TPAQ-MO** (P, f)
1. Initialize an empty *list*
2. Set E = set of the entries at the topmost level with cost greater than that of the previous level
3. low=1; $high$=$|E|$
4. while low+2<$high$ do
5. mid_1=(2·low+$high$)/3; mid_2=(low+2·$high$)/3
6. Group the entries in E into mid_1 slots
7. S_1 = the set of created slots
8. Group the entries in E into mid_2 slots
9. S_2 = the set of created slots
10. If $f(\bar{C}(S_1), mid_1) < f(\bar{C}(S_2), mid_2)$
11. Set $high$=mid_2 and S=S_1
12. Else, if $f(\bar{C}(S_1), mid_1) > f(\bar{C}(S_2), mid_2)$
13. Set low=mid_1 and S=S_2
14. Else, if $f(\bar{C}(S_1), mid_1) = f(\bar{C}(S_2), mid_2)$
15. Set low=mid_1, $high$=mid_2 and S=S_1
16. For each $s \in S$ do
17. Perform a NN search at $s.c$ on the points under $s.E$
18. Append the retrieved point to *list*
19. Return *list*

Figure 6.3: The *TPAQ-MO* algorithm

7. Discussion

All the above medoid queries have a *bichromatic* version, in which the candidate medoids belong to a dataset M which is different from that of the data points P. Set M may be a subset of P, or a set completely disjoint with P. For instance, the locations for potential franchise branches may be restricted to industrial buildings. The definitions of k-medoid, MA and MO queries (for both the *avg* and *max* versions) remain the same, but the reported medoid set R is a subset of M. TPAQ can easily capture bichromatic queries by performing its final step (i.e., NN queries) on M instead of P. Since the other steps remain the same, the performance is similar to the conventional (monochromatic) case.

In *weighted* queries, each data point is assigned a non-negative number indicating its importance (e.g., the weight of a residential block could be the number of its residents). Processing in the *max* case is identical to its un-weighted counterpart. However, the application of TPAQ to *avg* weighted queries requires an *aggregate* R*-tree [TP04], or any other aggregate data partition method. The aggregate R*-tree has the same structure and update algorithms as the regular R*-tree, except that each entry also stores the sum of weights of the data points in its sub-tree. The only necessary modification to TPAQ is using the sum of weights instead of the estimated entry cardinality $e.w$. For k-medoid queries, this affects the *InsertEntry* function (i.e., the calculation of the new slot center and weight upon the insertion of an entry), while for MA and MO it also affects the computation of the MPD and $\bar{C}(S)$ estimates.

Even for un-weighted *avg* queries, the use of aggregate R-trees can improve the accuracy of TPAQ. If each R-tree entry additionally contains the number and geometric centroid (i.e., the average x and y coordinate) of the points in its sub-tree, we can replace the estimations of $e.w$ and $e.c$ with these (i.e., the exact) values, respectively, leading to higher accuracy. The algorithmic modifications to TPAQ are similar to the weighted case.

In several practical scenarios, a medoid query may include additional capacity/service constraints. For instance, an application may require that a facility (i.e., medoid) can only serve up to a maximum number of clients (data points). Another application may require that each facility is assigned roughly the same number of clients (i.e., about $|P|/|R|$, where $|R|$ is the number of medoids). We refer to this class of problems as *maximum capacity* medoids. To deal with maximum capacity queries, TPAQ returns in addition to the medoids, an assignment of the data points to them. In particular,

as a first step, TPAQ retrieves the medoids in the way described in the previous sections (depending on the problem type; i.e., *k*-medoid, MA or MO). In a second step, TPAQ computes the assignment of the data points similar to the method of [AHA92].

The algorithm of [AHA92] computes a weight a_i for each medoid so that if each point p is assigned according to distance function[4] $pow(p, r(p)) = ||p-r(p)||^2 - a_i$ (where a_i is the weight of $r(p)$), then the capacity constraints are satisfied. Initially, all weights are equal. Depending on the number of points assigned to each medoid, their weights respectively decrease or increase. This hill climbing process is repeated until all constraints are satisfied. Note that the technique cannot be used as is, since it requires reading the entire dataset at every iteration. To avoid this problem, we can use the expected cardinalities and centroids of the entries at the partitioning level; we estimate the number of points assigned to each medoid assuming that the *e.w* points of every entry *e* are assigned to medoid *r*(*e.c*) (according to the *pow* function). To conclude the discussion about maximum capacity queries, the output of TPAQ is the set of medoids *R* and their corresponding weights a_i (i = 1,...,|*R*|). Given this information, the assignment of the data points is implicit.

TPAQ is targeted to static datasets. However, it can easily capture *dynamic* instances of the problem; when a batch of updates (insertions and deletions) takes place in dataset *P*, we can re-use the previous result to compute the new medoid set. In particular, if the updates do not affect the partitioning level (i.e., the number and MBRs of its entries remain the same), then we have to perform anew only the final step of TPAQ (i.e., NN queries at slot centers *s.c*). Actually, the NN search can be avoided for slots *s* where (i) the currently reported medoid is not deleted or (ii) an inserted point lies closer to *s.c*. In case (i) the medoid for *s* remains the same, while in case (ii) the new medoid is the closest inserted point to *s.c*. A NN search in *s* is required only if the current medoid is deleted and no new point lies closer to *s.c*. Note that if the updates affect the partitioning level, then we have to re-compute the medoids from scratch.

The above method works well when updates are infrequent, but it is slow for very dynamic datasets. To cope with high update rates, it is a common practice to store the data in main memory (e.g., [MXA04, YPK05, MHP05]). In this setting, a straightforward adaptation of TPAQ would be to use an in-memory R-tree, and evaluate the query as discussed above. This, however, would be expensive due to the slow R-tree updates. To overcome this problem, we could index the data points with a B-tree, sorted on their Hilbert values. In the case of *k*-medoid queries, we continuously report every m^{th} point as a medoid, where $m = |P|/k$. For MA (MO) queries, when they are first installed at the system, we can determine |*R*| using a binary search (hill climbing) method similar to Section 5 (6). Since computing *C*(*R*) for each considered |*R*| is expensive, we can use an estimate. Particularly, we may apply a regular grid, and maintain on-the-fly for each cell the number of data points falling therein. We estimate *C*(*R*) assuming that all points inside a cell are assigned to the medoid closest to its centroid. To maintain the medoids in subsequent update cycles, if the estimate of *C*(*R*) deviates from parameter *T* (if the estimated value of cost function *f* deviates from its previous value) by a percentage larger than some threshold, then we re-compute |*R*| as described above. Otherwise, we simply report the new |*R*| medoids.

8. Experimental Evaluation

In this section we evaluate the performance of the proposed methods for *k*-medoid, medoid-aggregate and optimization queries. For each of these three problems, we first present our experimental results for *avg*, and then for *max*, using both synthetic and real datasets. The synthetic ones (SKW) follow a Zipf distribution with parameter α=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 [BKSS90] whose block size ranges between 1 and 4Kbytes. For the experiments we use a 3GHz Pentium CPU.

8.1. *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 [RKV95] 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. [EKSX95a] suggest setting *numlocal* = 2 and *maxneighbors* = $k \cdot (M-k)/800$, where *M* is the number of leaf nodes in the R-tree of

[4] Formally, this is called the *power function* and results in a space partitioning known as the *power diagram* (a variation of the Voronoi diagram).

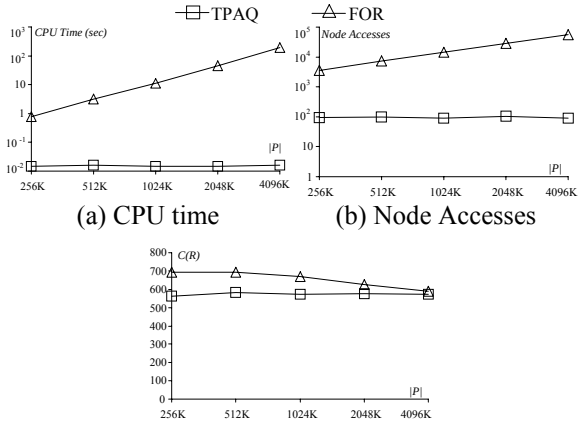
P. With these parameters, FOR terminates in several hours for most experiments. Therefore, we set *maxneighbors* = $k\cdot(M-k)/(8000\cdot\log M)$ and keep *numlocal* = 2. These values speed up FOR considerably, while the deterioration of the resulting solutions, with respect to the suggested values of *numlocal* and *maxneighbors*, is small. Regarding the *max* case, there is currently no other algorithm for disk-resident data. For the sake of comparison however, we adapted FOR to *max k*-medoid queries by defining $C(R)$ to be the maximum distance between objects and medoids; i.e., the CLARANS component of FOR exchanges the current medoid set R_i with a neighbor one R_i', only if the maximum distance achieved by R_i' is smaller than that of R_i. All FOR 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 CLARANS. The algorithms are compared for different data cardinality $|P|$, number of medoids k and block size. Table 8.1 summarizes the parameters 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.

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

Table 8.1: Parameter values

We first measure the effect of $|P|$ in the *avg* case. Figure 8.1a shows the running time of TPAQ and FOR for SKW, when k=32 and $|P|$ ranges 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 8.1b 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 tree level.

The cost improvements of TPAQ come with no compromise in answer quality. Figure 8.1c 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 of Figure 8.1 verify that TPAQ scales gracefully with the dataset cardinality and incurs much lower cost than FOR, without sacrificing the medoid quality.

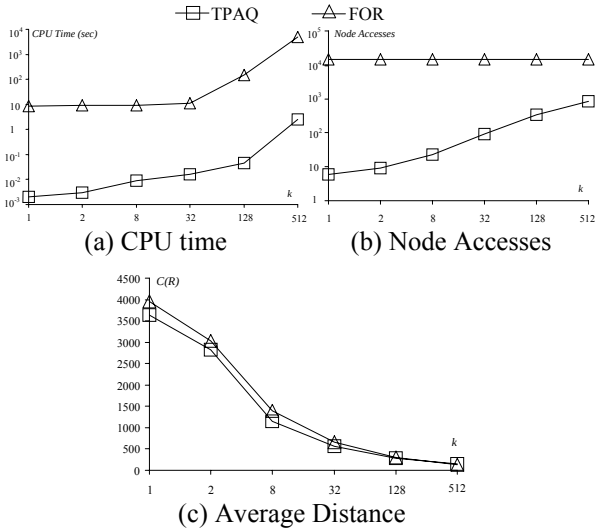


(a) CPU time (b) Node Accesses

(c) Average Distance

Figure 8.1: Performance versus $|P|$ (SKW, *avg*)

The next set of experiments studies the performance of TPAQ and FOR in the *avg* case, when k varies between 1 and 512, using a SKW dataset of cardinality $|P|$ = 1M. Figure 8.2a 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.



(a) CPU time (b) Node Accesses

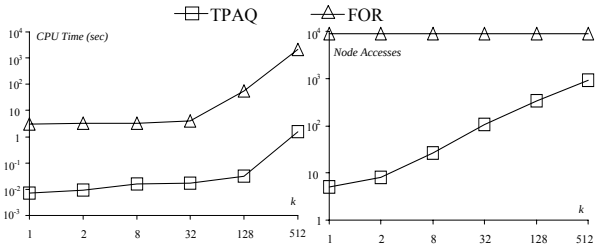
(c) Average Distance

Figure 8.2: Performance versus k (SKW, *avg*)

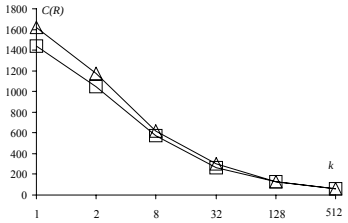
Figure 8.2b shows the effect of k on the I/O cost. The node accesses of FOR are constant and equal to the

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 8.2c compares the accuracy of the methods. TPAQ achieves lower $C(R)$, for all values of k.

In order to confirm the generality of our observations, Figures 8.3 and 8.4 repeat the above experiment for real datasets NA and LA. TPAQ outperforms FOR by orders of magnitude in terms of both CPU time (Figures 8.3a and 8.4a for NA and LA, respectively) and number of node accesses (Figures 8.3b and 8.4b). 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, 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).

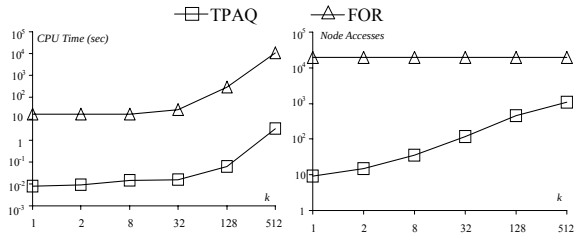


(a) CPU time (b) Node Accesses

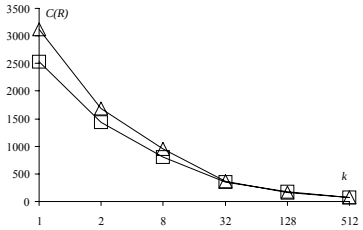


(c) Average Distance

Figure 8.3: Performance versus k (NA, *avg*)



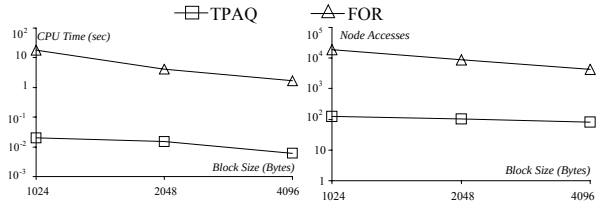
(a) CPU time (b) Node Accesses



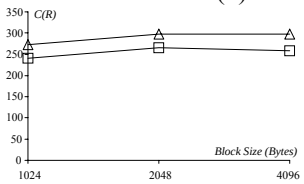
(c) Average Distance

Figure 8.4: Performance versus k (LA, *avg*)

Figures 8.5a and 8.6a show the running time of TPAQ and FOR on 32-medoid *avg* 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 8.5b and 8.6b). Independently of the block size, TPAQ incurs much fewer node accesses than FOR. Finally, Figures 8.5c and 8.6c illustrate the effect of the block size on the quality of the retrieved medoid set. In all cases, the average distance achieved by TPAQ is lower than that of FOR.

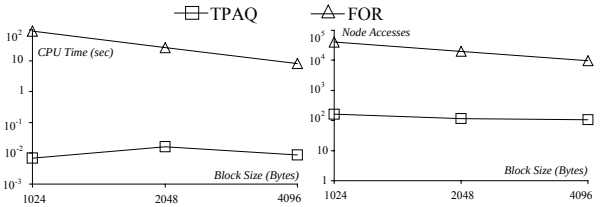


(a) CPU time (b) Node Accesses

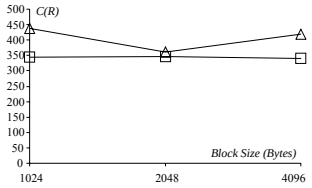


(c) Average Distance

Figure 8.5: Performance versus block size (NA, *avg*)



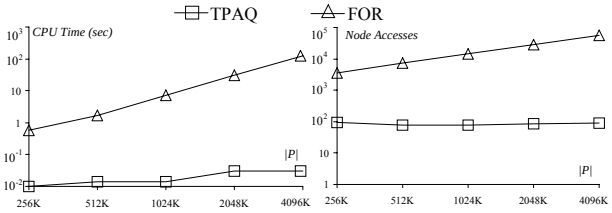
(a) CPU time (b) Node Accesses



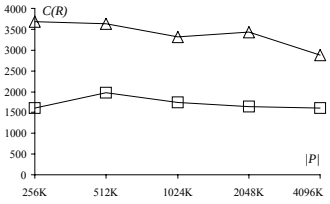
(c) Average Distance

Figure 8.6: Performance versus block size (LA, *avg*)

Next, we focus on *max* *k*-medoid queries. We perform the same experiments as in the *avg* case, with identical ranges and default values for the examined parameters (shown in Table 8.1). For brevity, we omit the results for NA. Figure 8.7 compares TPAQ and FOR on 32-medoid queries over SKW datasets of varying cardinality. As in Figure 8.1, our method outperforms FOR by orders of magnitude in both CPU and I/O cost. This is due to the fact that FOR reads from the disk the entire input dataset, and that its CLARANS component is much more expensive than our *max* slotting algorithm. Concerning the quality of the retrieved medoid sets (Figure 8.7c), TPAQ is better with a large margin. This is expected, since FOR was originally designed for the *avg* *k*-medoid problem. FOR converges to bad local minima when CLARANS considers swapping a current medoid with another representative because it selects the latter randomly among the set of representatives. Since the representatives follow the data distribution, the choices of CLARANS are biased towards dense areas of the workspace. Even though this behavior is desirable in *avg* *k*-medoid queries, it is clearly unsuitable for the *max* case, because even a single object at a sparse area can lead to a large $C(R)$.



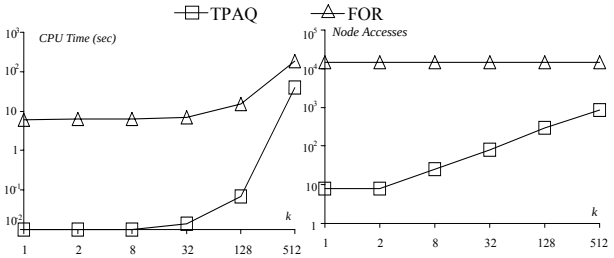
(a) CPU time (b) Node Accesses



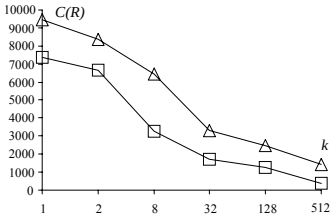
(c) Average Distance

Figure 8.7: Performance versus $|P|$ (SKW, *max*)

Figures 8.8 and 8.9 examine the effect of *k* on TPAQ and FOR over the SKW and LA datasets. The CPU cost of both methods increases with *k*. Larger values of *k* incur higher I/O cost for TPAQ for the reasons explained in the context of Figure 8.2b. FOR performs a constant number of node accesses since it always reads the entire dataset. Regarding the quality of the returned medoid sets, our algorithm achieves much lower maximum distance $C(R)$. The diagrams for NA are similar and omitted.

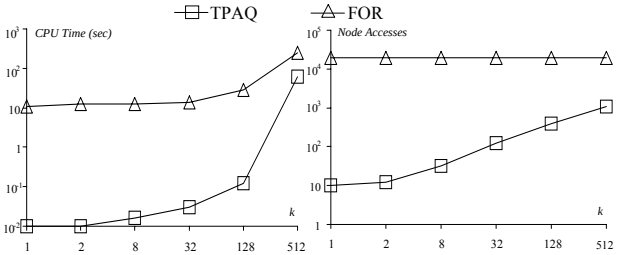


(a) CPU time (b) Node Accesses

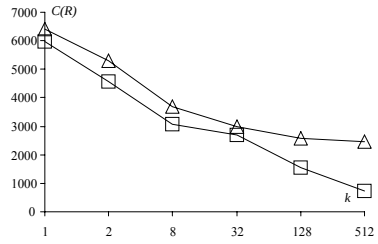


(c) Average Distance

Figure 8.8: Performance versus *k* (SKW, *max*)



(a) CPU time (b) Node Accesses

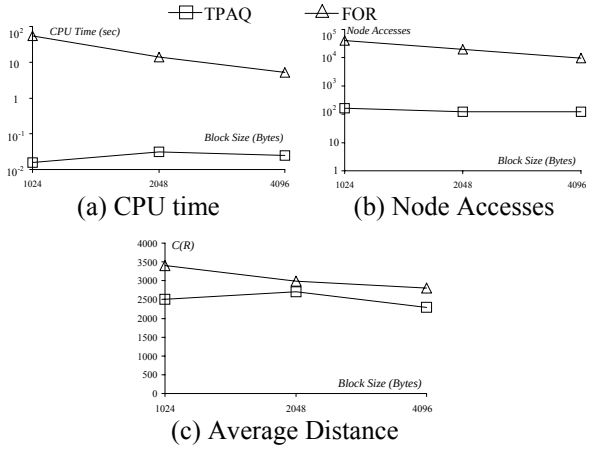


(c) Average Distance

Figure 8.9: Performance versus *k* (LA, *max*)

The final experiment for *k*-medoid queries measures the effect of the block size on the performance of the algorithms in the *max* case. Figure 8.10 presents the results for 32-medoid queries over the LA dataset, where the block size varies between 1, 2 and 4 KB. The CPU cost of FOR drops for higher block size, since there are fewer leaf nodes and CLARANS runs over fewer representatives. On the other hand, similar to Figure 8.6, the running time of TPAQ is higher for

block size 2K, because the partitioning level contains more entries than in the 1K and 4K cases. The number of node accesses for both algorithms drops for larger blocks, because the R-tree contains fewer nodes. Figure 8.10c illustrates the obtained maximum distance; TPAQ achieves better $C(R)$ in all cases.



(a) CPU time (b) Node Accesses

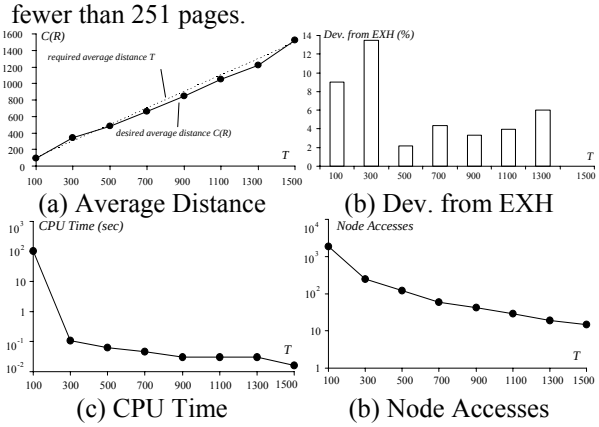
(c) Average Distance

Figure 8.10: Performance vs. block size (LA, *max*)

8.2. Medoid-Aggregate Queries

In this section we study the performance of TPAQ-MA, starting with the *avg* case. We use datasets SKW (with 1M objects) and LA, and vary T 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. 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.

Figure 8.11a shows the $C(R)$ for TPAQ-MA versus T on SKW. Clearly, the average distance returned by TPAQ-MA approximates the desired distance (dotted line) very well. Figure 8.11b 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 8.11c and 8.11d 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 at level 1 (leaf level, which contains 14,184 entries) and returns $|R|$=1272 medoids, incurring many computations and I/O operations. In all the 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.

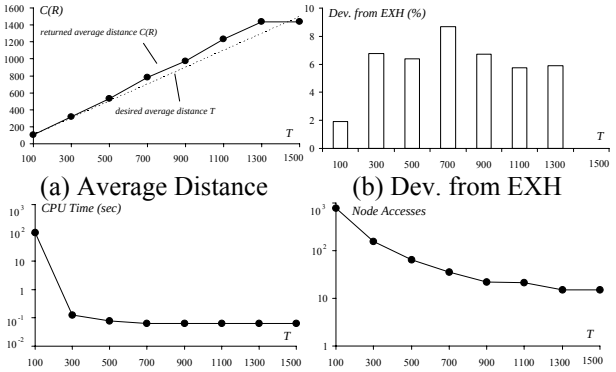


(a) Average Distance (b) Dev. from EXH

(c) CPU Time (b) Node Accesses

Figure 8.11: Performance versus T (SKW, *avg*)

Figure 8.12 repeats the above experiment for the LA dataset. Figures 8.12a and 8.12b 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 8.11; TPAQ-MA partitions 19,186 entries into slots and extracts $|R|$=296 medoids, taking in total 105.6 seconds and performing 781 node accesses.



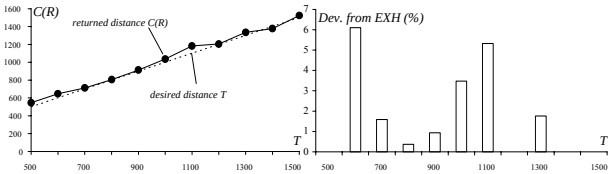
(a) Average Distance (b) Dev. from EXH

(c) CPU Time (b) Node Accesses

Figure 8.12: Performance versus T (LA, *avg*)

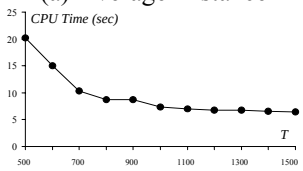
In Figures 8.13 and 8.14 we examine the performance of TPAQ-MA in the *max* case, using datasets SKW and LA. We compare again with the

EXH algorithm. It is implemented as explained in the beginning of the subsection, the difference being that now it uses the *max k*-medoid TPAQ algorithm. For *max*, the range of T is from 500 to 1500. We do not use the same range as in the previous two experiments (i.e., 100 to 1500), because for T<500 the number of required medoids becomes very high and EXH requires numerous hours to complete. As shown in Figures 8.13a and 8.14a, the maximum distance of TPAC-MA is close to the desired value T. In general, the deviation from EXH (illustrated in Figures 8.13b and 8.14b) is low, and in the worst case it reaches 6.1% for SKW and 11.6% for LA. The algorithm terminates in less than 21 seconds in all cases, and incurs a small number of node accesses.

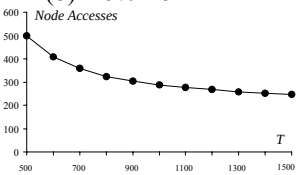


(a) Average Distance

(b) Dev. from EXH

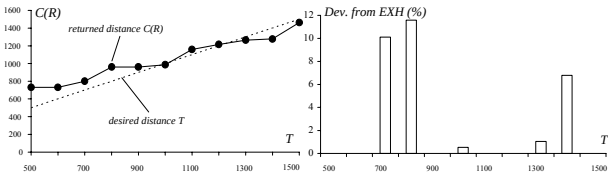


(c) CPU Time



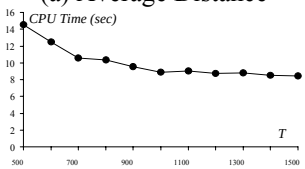
(b) Node Accesses

Figure 8.13: Performance versus T (SKW, *max*)

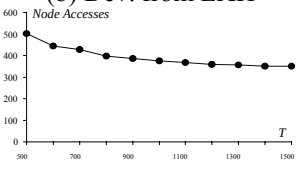


(a) Average Distance

(b) Dev. from EXH



(c) CPU Time



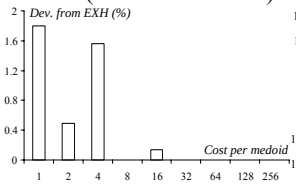
(b) Node Accesses

Figure 8.14: Performance versus T (LA, *max*)

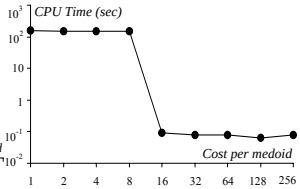
8.3. Medoid-Optimization Queries

Finally, we experiment on the performance of TPAQ-MO, using datasets SKW (with 1M objects) and LA. We process optimization queries with $f(C(R), |R|) = C(R) + Cost_{pm} \cdot |R|$, where $Cost_{pm}$ is the cost per medoid and ranges between 1 and 256. TPAQ-MO is again compared with an exhaustive algorithm (EXH), which in the MO case, (i) computes all the medoid sets with $|R|$ from 1 to $|E|$, by performing partitioning into slots in the same level as TPAQ-MO, (ii) calculates the (average or maximum) distance $C(R)$ achieved for each considered set, and (iii) returns the one that minimizes function f.

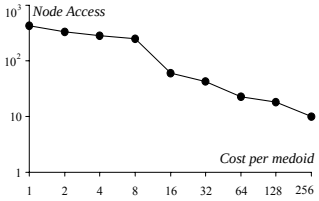
First, we experiment on *avg* MO queries. Figure 8.15a plots the deviation percentage (between the values of f achieved by TPAQ-MO and EXH) as a function of the cost $Cost_{pm}$ per medoid. The deviation does not exceed 1.8% in any case. Interestingly, TPAQ-MO returns exactly the same medoid sets as EXH for many values of $Cost_{pm}$, verifying the effectiveness of the gradient descent technique and the accuracy of the estimators described in Section 6. Figures 8.15b and 8.15c show the CPU and I/O costs of the algorithm. In both charts, the cost of TPAQ-MO is much higher when $Cost_{pm} \leq 8$. In these cases the running time is between 147 and 157 seconds and the number of node accesses ranges between 251 and 430. The returned medoid sets have size $|R|$ between 33 and 174. On the other hand, when $Cost_{pm} > 8$ the CPU time is less than 0.1 seconds and the incurred node accesses are fewer than 60. The answer contains from 3 to 24 medoids. This large difference is explained by the fact that when $Cost_{pm} \leq 8$ partitioning takes place in level 1 (with 14,184 entries), while for $Cost_{pm} > 8$ the partitioning level is level 2 (with 203 entries).



(a) Dev. from EXH



(b CPU Time

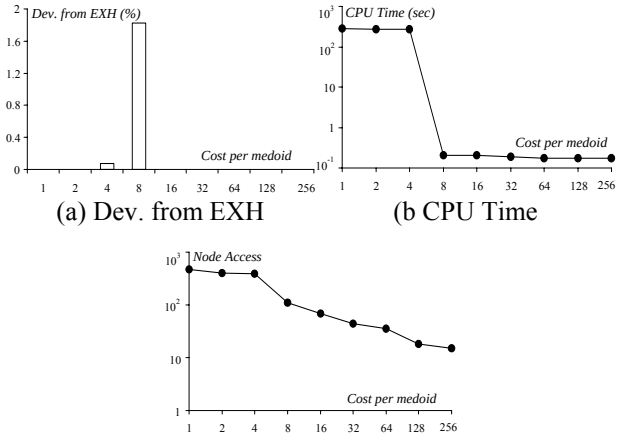


(c) Node Accesses

Figure 8.15: Performance versus $Cost_{pm}$ (SKW, *avg*)

In Figure 8.16 we repeat the above experiment for the LA dataset. The performance of TPAQ-MO is very similar to the SKW case. The deviation of TPAQ-MO from EXH is 0.07% and 1.82% for $Cost_{pm}$ equal to 4 and 8, respectively. For all the other values of $Cost_{pm}$ our algorithm retrieves the same medoid set as EXH. The cost of TPAQ-MO is plotted in Figures 8.16b and 8.16c. There is a large difference in both the CPU time and the node

accesses for $Cost_{pm} \leq 4$ and $Cost_{pm} > 4$. The reason for this behavior is the same as in Figure 8.15.

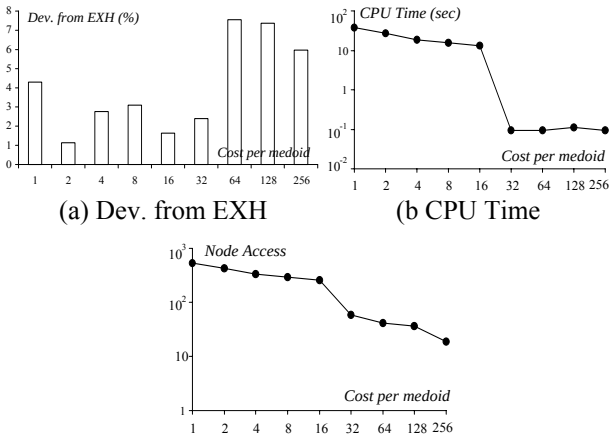


(a) Dev. from EXH (b CPU Time

(c) Node Accesses

Figure 8.16: Performance versus $Cost_{pm}$ (LA, *avg*)

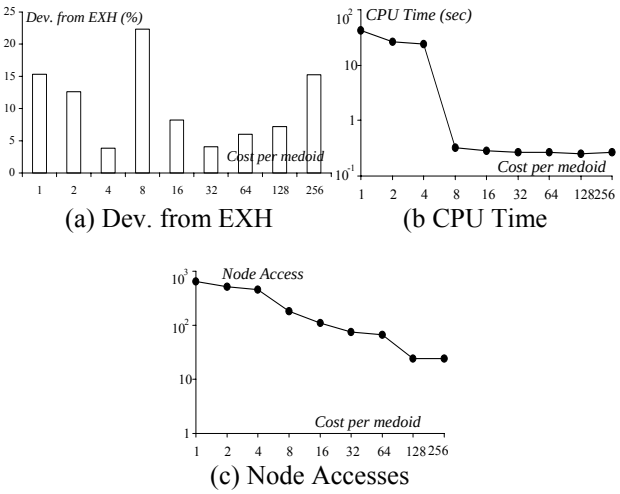
In the last two experiments we focus on *max* MO queries. Figures 8.17 and 8.18 illustrate the performance of TPAQ-MO when $Cost_{pm}$ varies between 1 and 256. The deviation from EXH is usually small. For SKW the maximum deviation is 7.5%. For LA the deviation is in general higher; on the average it is around 10% with maximum value 22.3% (for $Cost_{pm}$=8). TPAQ-MO performs worse for LA, because it contains large empty areas. On the other hand, SKW (even though it is very skewed) covers the whole workspace. Concerning the running time of TPAQ-MO, it does not exceed 43 seconds in any case. As in Figures 8.15 and 8.16, both the I/O and the CPU costs drop when partitioning takes place to a higher level. For SKW (for LA), the partitioning level is level 1 for $Cost_{pm} \leq 16$ (for $Cost_{pm} \leq 4$), while for higher $Cost_{pm}$ it is level 2.



(a) Dev. from EXH (b CPU Time

(c) Node Accesses

Figure 8.17: Performance versus $Cost_{pm}$ (SKW, *max*)



(a) Dev. from EXH (b CPU Time

(c) Node Accesses

Figure 8.18: Performance versus $Cost_{pm}$ (LA, *max*)

9. Conclusion

This paper studies *k*-medoids and related problems in large databases. In particular, we consider *k*-medoid, medoid-aggregate (MA) and medoid-optimization (MO) queries. We propose TPAQ (*Tree-based PArtition Querying*), a framework for their efficient processing that works for both their *avg* and *max* versions. TPAQ provides high-quality answers almost instantaneously, thus facilitating data analysis, especially in time-critical resource allocation applications. Our techniques exploit the data partitioning properties of an 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 and MO, this number is decided using binary search and a gradient descend method, respectively, in conjunction with some (average or maximum) distance estimators. Finally, TPAQ issues a NN query to retrieve one medoid for each slot. 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 of TPAQ for processing MA and MO queries, in both *avg* and *max* cases. In the future, we plan to extend the proposed methodology to high-dimensional spaces, using appropriate data partition indexes [BKK96].

Acknowledgments

This work was supported by grant HKUST 6184/06E from Hong Kong RGC.

References

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

[AHA92] Aurenhammer, F., Hoffmann, F., Aronov, B. Minkowski-type theorems and least-squares partitioning. *ACM Symposium on Computational Geometry*, 1992.

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

[BKK96] Berchtold, S., Keim, D., Kriegel, H. The X-tree: An Index Structure for High-Dimensional Data. *VLDB*, 1996.

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

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

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

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

[FPSU96] Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R. Advances in Knowledge Discovery and Data Mining. *AAAI/MIT Press*, 1996.

[G85] Gonzalez, T. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38: 293-306, 1985.

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

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

[H75] Hartigan, J.A. Clustering Algorithms. *Wiley*, 1975.

[HE03] Hamerly, G., Elkan, C. Learning the *k* in *k*-means. *NIPS*, 2003.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *ACM TODS*, 24(2): 265-318, 1999.

[HTF01] Hastie, T., Tibshirani, R., Friedman, J. The Elements of Statistical Learning. *Springer-Verlag*, 2001.

[KF93] Kamel, I., Faloutsos, C. On Packing R-trees. *CIKM*, 1993.

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

[LR95] Lo, M.L., Ravishankar, C.V. Generating Seeded Trees from Data Sets. *SSD*, 1995.

[LR98] Lo, M.L., Ravishankar, C.V. The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins. *TKDE*, 10(1): 136-151, 1998.

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

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

[MP03] Mamoulis, N., Papadias, D. Slot Index Spatial Join. *TKDE*, 15(1): 211-231, 2003.

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

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

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

[PM00] Pelleg, D., Moore, A.W. X-means: Extending k-means with Efficient Estimation of the Number of Clusters. *ICML*, 2000.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[TP04] Tao, Y., Papadias, D. Range Aggregate Processing in Spatial Databases. *TKDE*, 16(12), 1555-1570, 2004.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. *TKDE*, 12(1): 19-32, 2000.

[W91] Welzl, E. Smallest Enclosing Disks (Balls and Ellipsoids). *New Results and New Trends in Computer Science*, 555: 359-370, 1991.

[XWZ+05] Xing, G., Wang, X., Zhang, Y., Lu, C., Pless, R., Gill, C.: Integrated coverage and connectivity configuration for energy conservation in sensor networks. *ACM TOSN*, 1(1): 36-72, 2005.

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

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

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