2009

Computing Medoids in Large Spatial Datasets

Kyriakos MOURATIDIS
Singapore Management University, kyriakos@smu.edu.sg

Dimitris PAPADIAS
Hong Kong University of Science and Technology

Spiros PAPADIMITRIOU
IBM TJ Watson Research Center

DOI: https://doi.org/10.1201/9781420073980

Computing Medoids in Large Spatial Datasets

Kyriakos Mouratidis

Dimitris Papadias

Spiros Papadimitriou

CONTENTS

8.1 INTRODUCTION

In this chapter, we consider a class of queries that arise in spatial decision making and resource allocation applications. Assume that a company wants to open a number of warehouses in a city. Let P be the set of residential blocks in the city. P represents customer locations to be potentially served by the company. At the same time, P also comprises the candidate warehouse locations because the warehouses themselves must be opened in some residential blocks. In this context, an analyst may ask any of the following questions:

Q1. ***k*-Medoid query:** If the number k of warehouses is known, in which residential blocks should they be opened, so that the average distance from each location in P to its closest warehouse is minimized?

Q2. **Medoid-aggregate query:** If the average distance should be around a given value, what is the smallest number of warehouses (and their locations) that best approximates this value?

Q3. **Medoid-optimization query:** If the warehouse opening/maintenance overhead and the transportation cost per mile are given, what is the number of warehouses (and their locations) that minimizes the total cost?

The warehouse locations correspond to the *medoids*. Since the k-medoid problem (Q1) is NP-hard (Garey and Johnson, 1979), research has focused on approximate algorithms, most of which are suitable only for datasets of small and moderate sizes. On the contrary, this chapter focuses on very large databases. In addition to conventional k-medoids, we introduce and solve the alternative queries Q2 and Q3, which have practical relevance.

To formalize, given a set P of data points, we wish to find a set of medoids $R \subseteq P$, subject to certain optimization criteria. The average (*avg*) Euclidean distance $\|p - r(p)\|$ between each point $p \in P$ and its closest medoid $r(p) \in R$ is denoted by

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

Letting $|R|$ represent the cardinality of R, the *k-medoid query* can be formally stated as: "Given dataset P and integer parameter k, find $R \subseteq P$, such that $|R| = k$ and $C(R)$ is minimized." Figure 8.1 shows an example, where the dots represent points in P (e.g., residential blocks), $k = 3$ and $R = \{h, o, t\}$. The three medoids h, o, t are candidate locations for service facilities (e.g., warehouses or distribution centers), so that the average distance $C(R)$ from each block to its closest facility is minimized.

The *medoid-aggregate* (MA) query is defined as: "Given P and a value T, find $R \subseteq P$, such that $|R|$ is minimized and $C(R) \approx T$." In other words, k is not specified in advance. Instead, a target value T for the average distance is given, and we want to select a minimal set R of medoids, such that $C(R)$ best approximates T. Finally, the *medoid-optimization* (MO) query is formalized as: "Given P and a cost function f that is monotonically increasing with both the number of medoids $|R|$ and with $C(R)$,

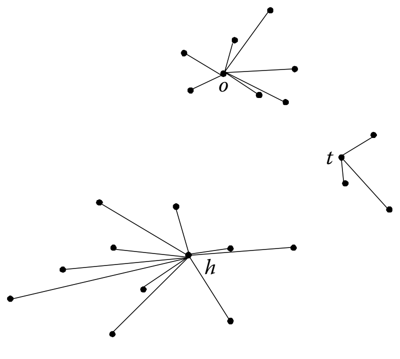


FIGURE 8.1 Example of 3-medoids.

find $R \subseteq P$ such that $f(C(R), |R|)$ is minimized." For example, in Q3 above, function f may be defined as $f(C(R), |R|) = C(R) + Cost_{pm} \times |R|$, where $Cost_{pm}$ is the opening/maintenance cost per warehouse. The goal is to achieve the best tradeoff between the number of warehouses and the average distance achieved.

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

$$C(R) = max_{p \in P} \|p - r(p)\| .$$

For instance, the company in our example may want to minimize the maximum distance (instead of the average one) between the residential blocks and their closest warehouse, potentially achieving a desired $C(R)$ with the minimal set of warehouses (MA), or minimizing a cost function (MO).

In this chapter, we present *Tree-based PArtition Querying* (TPAQ) (Mouratidis, Papadias, and Papadimitriou, 2008), a methodology that can efficiently process all of the previously mentioned 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 processing cost (seconds as opposed to hours). The rest of the chapter is organized as follows. Section 2 reviews related work. Section 3 introduces key concepts and outlines the general TPAQ framework. Section 4 considers k-medoid queries, while Section 5 and Section 6 focus on MA and MO queries, respectively. Section 7 presents experimental results and Section 8 concludes the chapter.

8.2 BACKGROUND

Although TPAQ can be used with any data partition method, we assume R*-trees (Beckmann, et al., 1990) due to their popularity. Section 8.2.1 overviews R*-trees and their application to nearest neighbor queries. Section 8.2.2 presents existing algorithms for k-medoids and related problems.

8.2.1 R-Trees and Nearest Neighbor Search

We illustrate our examples with the R*-tree of Figure 8.2 that contains the data points of Figure 8.1, 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 (Theodoridis, Stefanakis, and Sellis, 2000). Additionally, in practice, nodes at the same level contain a similar number of data

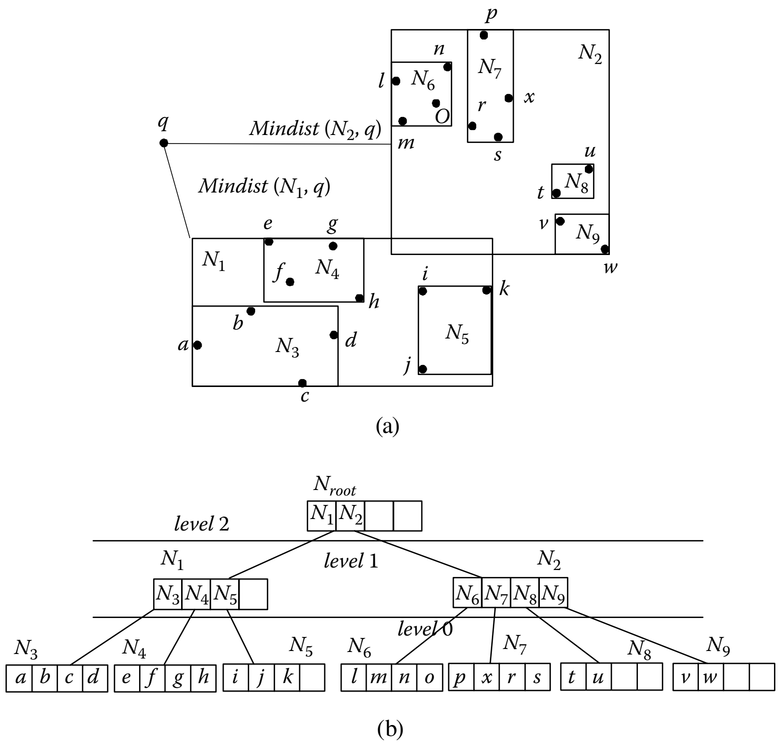


FIGURE 8.2 R-tree example.

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 data 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.

The R-tree family of indexes has been used for spatial queries such as range search, nearest neighbors, and spatial joins. A nearest neighbor (NN) query retrieves the data point 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 8.2 shows the *mindist* between *q* and nodes N_1 and N_2. The algorithm of Roussopoulos, Kelly, and Vincent (1995), shown in Figure 8.3, 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. Let *bestNN* be the best NN found thus far (e.g., *bestNN*=*e*) and *bestDist* be its distance from *q* (e.g., *bestDist* = $\|e - q\|$). Subsequently, the algorithm only visits entries whose minimum distance is less than *bestDist*. In

```
Algorithm NN (q,N)
1.  If N is a leaf node
2.  For each point $p \in N$
3.  If $||p-q|| < bestDist$
4.  $best NN = p$; $bestDist = ||p-q||$
5.  Else // N is an internal node
6.  For each child $N_i$ of N do
7.  If $mindist(q, N_i) < bestDist$
8.  NN($q, N_i$)
```

FIGURE 8.3 The *NN* algorithm. (From Roussopoulos, N., Kelly, S., and Vincent, F. Nearest neighbor queries. *SIGMOD*, 1995.)

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 (1999) propose a best-first NN algorithm that is I/O optimal (i.e., it only visits nodes that may contain NNs) and incremental (the number k of NNs does not need to be known in advance).

8.2.2 *k*-Medoids and Related Problems

A number of approximation schemes for k-medoids and related problems appear in the literature (Arora, Raghavan, and Rao, 1998). Most of them, however, are largely theoretical in nature. Kaufmann and Rousseeuw (1990) propose *partitioning around medoids* (PAM), a practical algorithm based on the hill climbing paradigm. PAM (illustrated in Figure 8.4) 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 data point. 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) (Kaufmann and Rousseeuw, 1990) alleviates the problem by generating random samples from P and executing PAM on them. Ng and Han (1994) 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.

```
Algorithm **PAM** ($P, k$)
1.  Initialize $R_0 = \{r_1, r_2, ..., r_k\}$ to a random subset of $P$ with $k$ elements, and set $i = 0$
2.  Repeat
3.  $bestNeighbor = R_i$
4.  For each position $j = 1$ to $k$ do
5.  For each point $p \in P$ do
6.  $R_i' = R_i - \{r_j\} \cup \{p\}$
7.  If $C(R_i') < C(bestNeighbor)$
8.  $bestNeighbor = R_i'$
9.  $R_{i+1} = bestNeighbor$; $i = i + 1$
10. Until $R_i = R_{i-1}$ // no improvement was made
11. Return $R$
```

FIGURE 8.4 The *PAM* algorithm. (From Kaufman, L. and Rousseeuw, P. *Finding Groups in Data*. Wiley-Interscience, 1990.)

Assuming that P is indexed with an R-tree, Ester, Kriegel, and Xu (1995a,b) developed *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. FOR is more efficient than CLARANS, but 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 Ester, Kriegel, and Xu use R-trees with only 50,559 points and 1,027 leaf nodes).

To the best of our knowledge, no existing method for the *max* case is suitable for disk-resident data. For in-memory processing, the *k-centers algorithm* (CTR) of Gonzales (1985) answers *max* k-medoid queries in $O(k \times |P|)$ time with an approximation factor of 2; i.e., 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 is shown in Figure 8.5. 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 i-th medoid is the one that has the maximum distance from any point in R_{i-1}. Finally, set R_k is returned as the result. The algorithm is simple and works well in practice. However, its adaptation to large datasets would be very expensive in terms of both CPU and I/O cost, since in order to find the i-th 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

```
Algorithm CTR (P, k)
1. Choose a point $p \in P$ randomly, and set $R_1 = \{p\}$
2. For $i = 2$ to $k$ do
3.     Let $p$ be the point in $P - R_{i-1}$ that is furthest from any medoid in $R_{i-1}$
4.     $R_i = R_{i-1} \cup \{p\}$
5. Return $R_k$
```

FIGURE 8.5 The CTR algorithm for *max k*-medoids. (From Gonzalez, T. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38: 293–306, 1985.)

spatial region Q (i.e., range for a new facility), an 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 an 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. (2006) propose an exact method for this problem. This technique is complementary to the proposed algorithms because it can be used to increase the cardinality of an existing medoid set when there is a need for incremental processing (e.g., the company of our example may decide to open an additional warehouse in a given area).

The k-medoid problem is related to clustering. Clustering methods designed for large databases include DBSCAN (Ester et al., 1996), BIRCH (Zhang, Ramakrishnan, and Livny, 1996), CURE (Guha, Rastogi, and Shim, 1998), and OPTICS (Ankerst et al., 1999). However, the objective of clustering in general and of these techniques in particular is inherently different. Extensive work on medoids and clustering has been carried out in the areas of statistics (Hartigan, 1975; Kaufman and Rousseeuw, 1990; Hastie, Tibshirani, and Friedman, 2001), machine learning (Pelleg and Moore, 1999, 2000; Hamerly and Elkan, 2003), and data mining (Ester et al., 1996; Fayyad et al., 1996). However, the focus there is on assessing the statistical quality of a given clustering, usually based on assumptions about the data distribution (Hastie et al., 2001; Kaufman and Rousseeuw, 1990; Pelleg and Moore, 2000). Only few approaches aim at dynamically discovering the number of clusters (Pelleg and Moore, 2000; Hamerly and Elkan, 2003). Besides tackling problems of a different nature, these algorithms are computationally intensive and unsuitable for disk-resident datasets.

8.3 FRAMEWORK OVERVIEW AND BASIC 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 a 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 chapter.

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 coordinates 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 8.6. 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 Welzl (1991), which 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 8.6 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.

8.4 *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 8.2, 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, $w_1 = w_2 = N/2 = 11.5$ (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 8.7 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 conceptually splitting leaf level nodes.

```
Function InitEntries (P, k)
1.  Load the root of the R-tree of P
2.  Initialize list = {e}, where e = ($N_{root}.c$, |P|, $N_{root}$)
3.  While list contains fewer than k extended entries do
4.  Initialize an empty list next_level_entries
5.  For each e = (c, w, N) 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 ($N_i.c$, $w_i$, $N_i$) into next_level_entries
10. Set list = next_level_entries
11. Return list
```

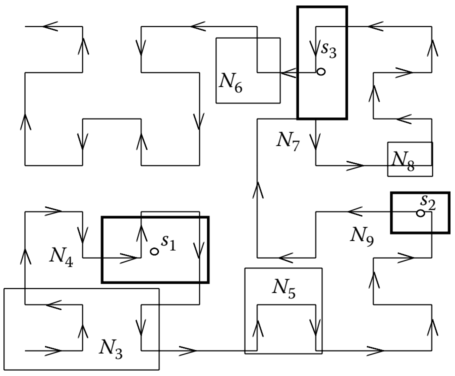
FIGURE 8.7 The *InitEntries* function.

The next step merges the k' initial entries in order to obtain exactly k groups. First, 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 multidimensional space into a linear order. Among several alternatives, Hilbert curves best preserve the locality of points (Korn, Pagel, and Faloutsos, 2001; Moon et al., 2001). 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 8.8a shows the level 1 MBRs (for the R-tree of Figure 8.2) 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 slots, based on proximity. 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 slot s_1 using the *InsertEntry* function of Figure 8.6. The center of s_1 is updated to the midpoint of N_3 and N_4's centers, as illustrated in Figure 8.8b. TPAQ proceeds in this manner, until the final slots and weighted centers are computed as shown in Figure 8.8c.

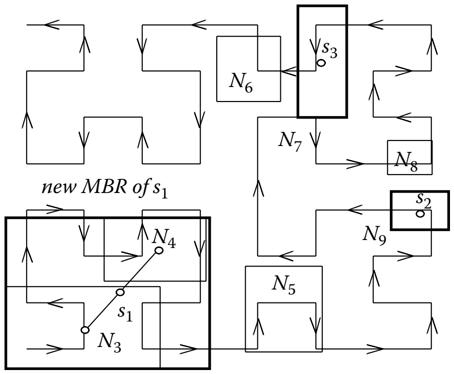
After grouping all entries into exactly k slots, we find one medoid per slot by performing an NN query. The query point is the slot's weighted center $s.c$, and the search space is the set of entries $s.e$. Since all the levels of the R-tree down to the partition level have already been loaded in memory, 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 8.9 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 (Lo and Ravishankar, 1995, 1998; Mamoulis and Papadias, 2003). 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 points that belong to different datasets.

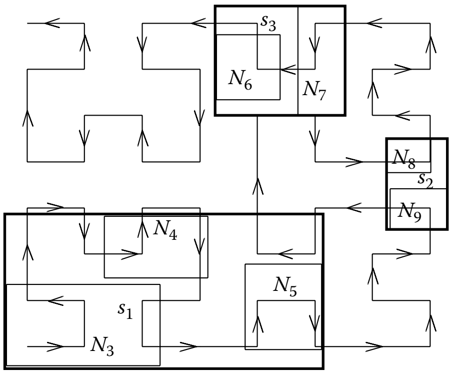
TPAQ follows similar steps for the *max* case. The function *InitEntries* proceeds as before, but without computing the expected cardinality for entries and slots; in the *max* version of the problem, we use only the geometric centroids of the R-tree entries. Let E be the set of entries in the partitioning level. We apply the CTR algorithm (described in Section 8.2.2) to select k slot seeds among the entry centers $e.c$ in E. Then, we insert the remaining entries in E one by one into the slot with the



(a) Hilbert seeds



(b) Insertion of N_3



(c) Final slot contents

FIGURE 8.8 Insertion of entries into slots.

```
Algorithm TPAQ ($P$, $k$)
1.  Initialize a set $S = \emptyset$, and an 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
```

FIGURE 8.9 The *TPAQ* algorithm.

closest center. Finally, we perform an 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 8.2, level 1 is chosen as the partitioning level. Among the entries of level 1, assume that CTR 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 8.10. The three circles

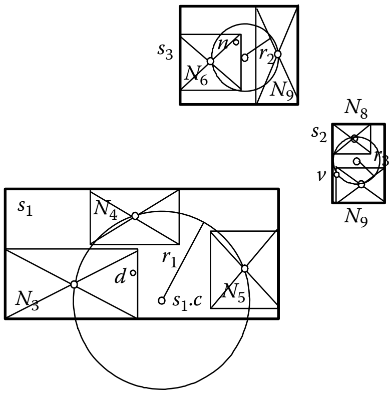


FIGURE 8.10 3-medoids in the *max* case.

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 $R = \{d, v, n\}$.

8.5 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 $C(R) = T$. 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. 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 8.4, MPD is equal 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, and that the corresponding medoid is at $e.c$. The average distance $\bar{C}(e)$between $e.c$ and the points in e is given by the following lemma.

Lemma 8.1: 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 8.1.

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:

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

* This is a reasonable assumption for low-dimensional R-trees (Theodoridis et al., 2000).

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 8.7, 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 entries N_3 to N_9. The MPD is less than T, and level 1 is selected for partitioning. *InitEntries* returns a list containing seven 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 value T. If E is the set of entries in the partitioning level, the candidate values for $|R|$ range between 1 and $|E|$. TPAQ-MA assumes that $C(R)$ decreases as $|R|$ increases, 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 8.4. 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 $e \in s$ is equal to 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 8.1. 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 TPAQ-MA algorithm is shown in Figure 8.11.

In the example of Figure 8.2, 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 four slots, as shown in Figure 8.12. 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 TPAQ, TPAQ-MA performs an 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.

Consider now the *max* version of the MA problem. *InitEntries* chooses for partitioning the top-most level with 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, $\bar{C}(e) = D/2$ (where D is the diagonal of e), and MPD = $max_{e \in E}\bar{C}(e)$ (where E is the set of entries in the given level).

```
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 C̄(S) < T, set high = mid
9.  Else, set low = mid
10. For each s∈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 8.11 The *TPAQ-MA* algorithm.

Similar to the *avg* case, in order to determine the number of medoids that best approximate the target distance T, we perform a 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 8.4). Let S be the set of slots for a value of $|R|$. To estimate the achieved $C(R)$ [i.e., to

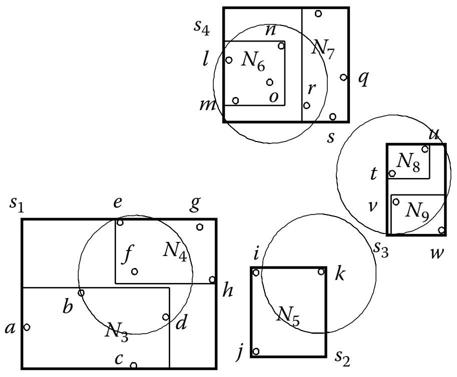


FIGURE 8.12 Entries and final slots.

compute $\overline{C}(S)$], we assume that the maximum distance within each slot s is equal to 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 8.10. $C(R)$ 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 8.11 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.

8.6 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 an 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 8.1 in the *avg* case, and let $f(C(R), |R|)$ be $C(R) + Cost_{pm} \times |R|$, 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 $f(C(R^i), |R^i|)$ vs. $|R^i|$ is shown in Figure 8.13, then the optimal $|R|$ is 3 and the result of the query is $\{h, o, t\}$ (as in Figure 8.1). 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 $|R|$.

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 $f(\text{MPD}, |E|)$, 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

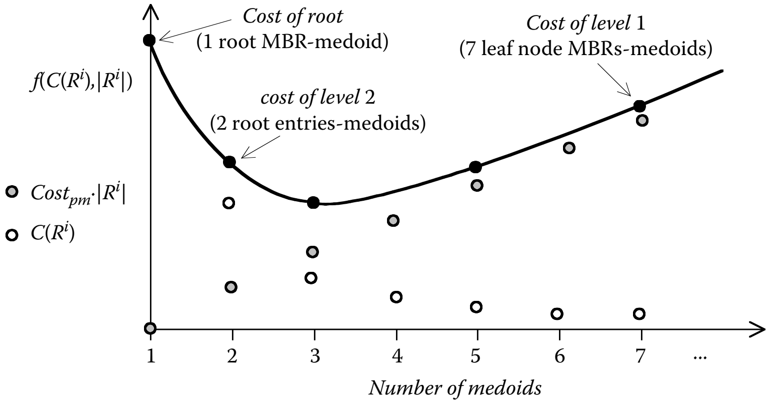
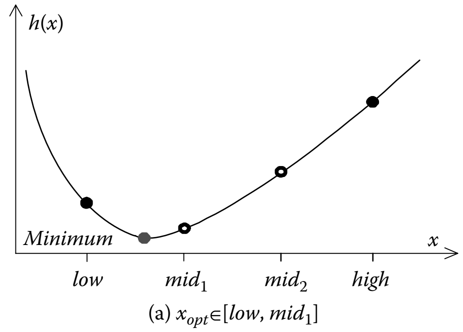


FIGURE 8.13 $f(C(R^i), |R^i|)$ versus number of medoids.

detected increase in the curve of Figure 8.13). 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 8.7 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 8.2, *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 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 8.14, 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 8.14a), or $x_{opt} \in [mid_1, mid_2]$ (Figure 8.14b). 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 $[mid_1, mid_2]$.



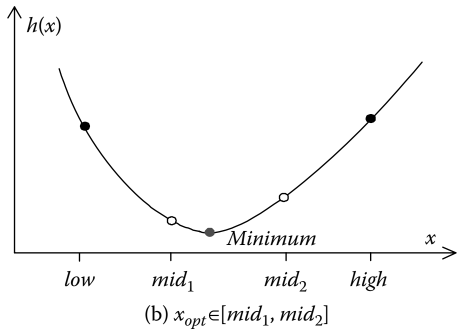


FIGURE 8.14 Computing the minimum of a function h.

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.

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 8.4, and estimate the corresponding $C(R)$ as the quantity $\bar{C}(S)$ discussed in Section 8.5. 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 an NN search at the center of each slot $s \in S$ of the corresponding partitioning. TPAQ-MO is illustrated in Figure 8.15. The algorithm works for both *avg* and *max* MO queries, by using the corresponding MPD and $\bar{C}(S)$ functions, and the appropriate slotting strategies. In our running example, for the *avg* case, level 1 is the

```
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.  $mid_1 = (2 \cdot low + high)/3$; $mid_2 = (low + 2 \cdot high)/3$
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 8.15 The *TPAQ-MO* algorithm.

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 8.8.

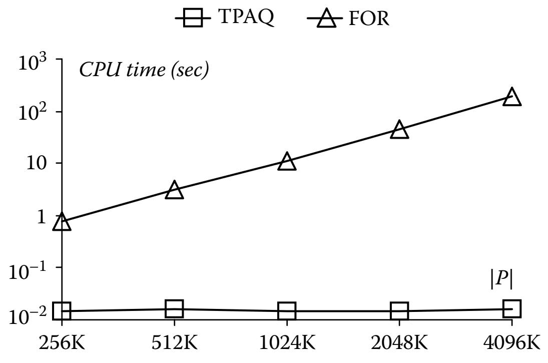
8.7 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of the proposed methods for k-medoid, medoid-aggregate, and medoid-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 $\alpha = 0.8$, and have cardinality 256K, 512K, 1M, 2M and 4M points (with 1M being the default). The real dataset (LA) contains 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 (Berchtold, Keim, and Kriegel, 1996) with a 2Kbyte page size. For the experiments, we use a 3GHz Pentium CPU.

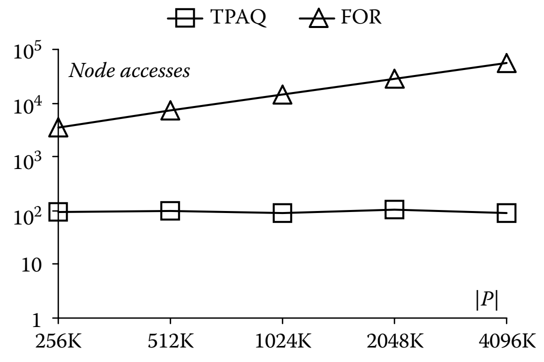
8.7.1 k-MEDOID QUERIES

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 TPAQ, we use the depth-first algorithm of Roussopoulos et al. (1995) to retrieve the nearest neighbor of each computed slot center. 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. (1995a) suggest setting *numlocal* = 2 and *maxneighbors* = $k \times (M - k)/800$, where M is the number of leaf nodes in the R-tree of P. With these parameters, FOR terminates in several hours for most experiments. Therefore, we set *maxneighbors* = $k \times (M - k)/(8000 \times \log M)$ and keep *numlocal* = 2. These values speed up FOR considerably, while the deterioration of the resulting solutions is small (with respect to the suggested values of *numlocal* and *maxneighbors*). 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 data points and medoids; that is, 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 $|P|$ and number of medoids k; for k, the tested values are from 1 to 512, and its default is 32. In each experiment we fix either parameter (i.e., $|P|$ or k) to its default value and vary the other one.

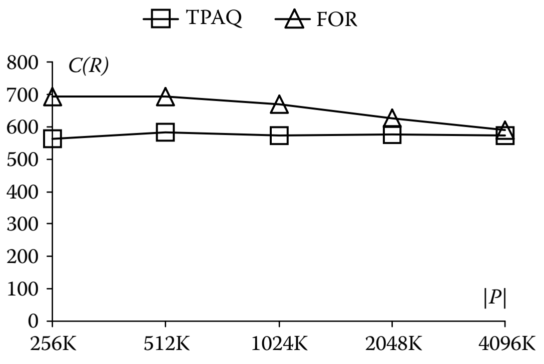
We first measure the effect of $|P|$ in the *avg* case. Figure 8.16a shows the CPU time of TPAQ and FOR for SKW, when $k = 32$ and $|P|$ ranges between 256K and 4M. TPAQ is 2 to 4 orders of magnitude faster than FOR. Even for $|P| = 4$M points, our method terminates in less than 0.04 sec (while FOR needs more than 3 min). Figure 8.16b shows the I/O cost (number of node accesses) for the same experiment. FOR is approximately 2 to 3 orders of magnitude more expensive than TPAQ because 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



(a) CPU time



(b) Node accesses



(c) Average distance

FIGURE 8.16 Performance versus |P| (SKW, *avg*).

improvements of TPAQ come with no compromise in answer quality. Figure 8.16c 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 a higher $|P|$ implies more possible "paths" to a local minimum. To summarize, the results of Figure 8.16 verify that TPAQ scales gracefully with the dataset cardinality and incurs much lower cost than FOR, without sacrificing medoid quality.

The next set of experiments studies the performance of TPAQ and FOR in the *avg* case, when k varies between 1 and 512, using an SKW dataset of cardinality $|P| = 1$M. Figure 8.17a compares the CPU time of the methods. In all cases, TPAQ is three orders of magnitude faster than FOR. It is worth mentioning that for $k = 512$ our method terminates in 2.5 sec, while FOR requires approximately 1 hour and 20 min. 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 8.17b 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 (1) it needs to descend more R-tree levels in order to find one with a sufficient number (i.e., k) of entries, and (2) 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 8.17c 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, Figure 8.18 repeats the above experiment for the real dataset LA. TPAQ outperforms FOR by orders of magnitude in terms of both CPU time (Figure 8.18a) and number of node accesses (Figure 8.18b). Regarding the average distance $C(R)$, the methods achieve similar results (Figure 8.18c), with TPAQ being the winner.

Next, we focus on *max* k-medoid queries. We perform the same experiments as in the *avg* case, with identical test ranges and default values for $|P|$ and k. Figure 8.19 compares TPAQ and FOR on 32-medoid queries over SKW datasets of varying cardinality. As in Figure 8.16, our method significantly outperforms FOR in terms of both CPU and I/O cost because FOR reads the entire input dataset and its CLARANS component is much more expensive than our *max* slotting algorithm. TPAQ is also considerably better on the quality of the retrieved medoids (Figure 8.19c). This is expected because FOR is originally designed for the *avg* k-medoid problem. FOR converges to poor 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 toward 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 point in a sparse area can lead to a large $C(R)$.

Figure 8.20 and Figure 8.21 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

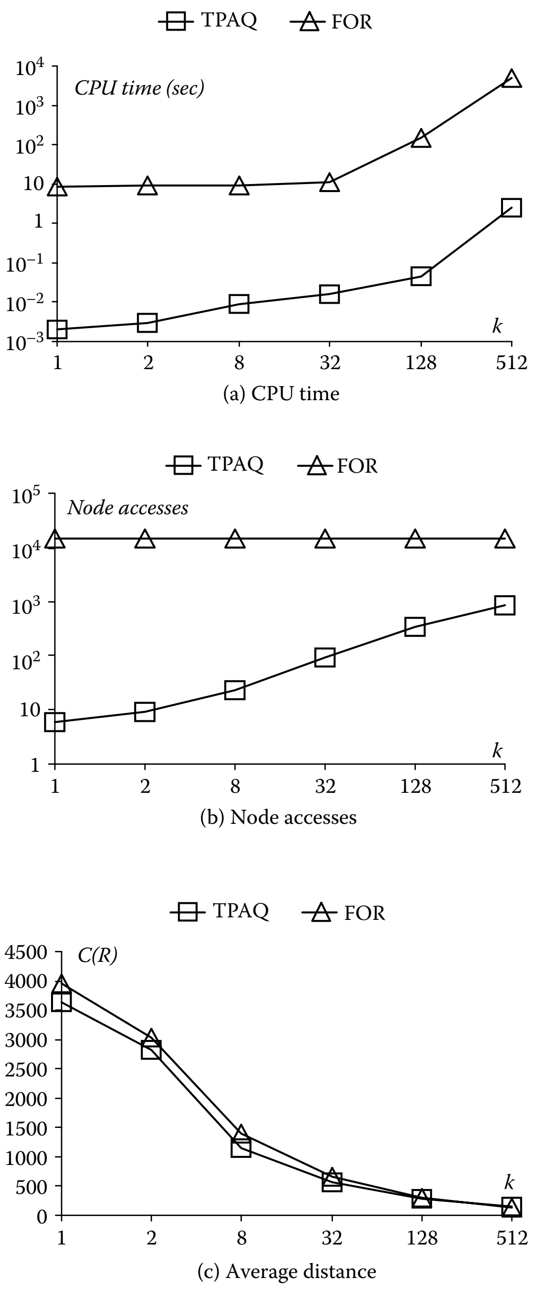
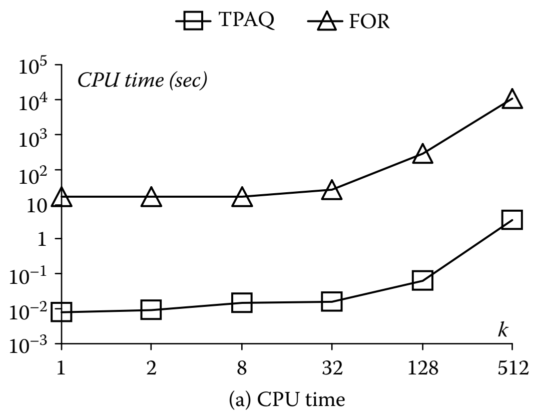
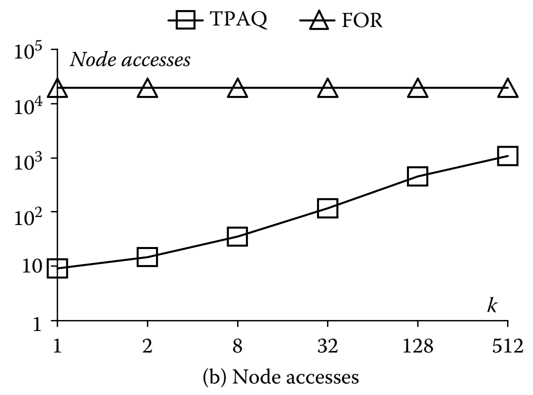


FIGURE 8.17 Performance versus *k* (SKW, *avg*).





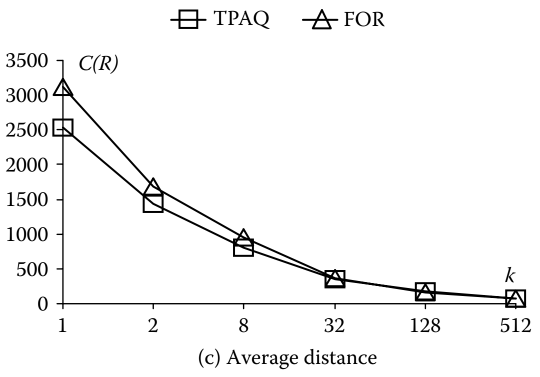


FIGURE 8.18 Performance versus *k* (LA, *avg*).

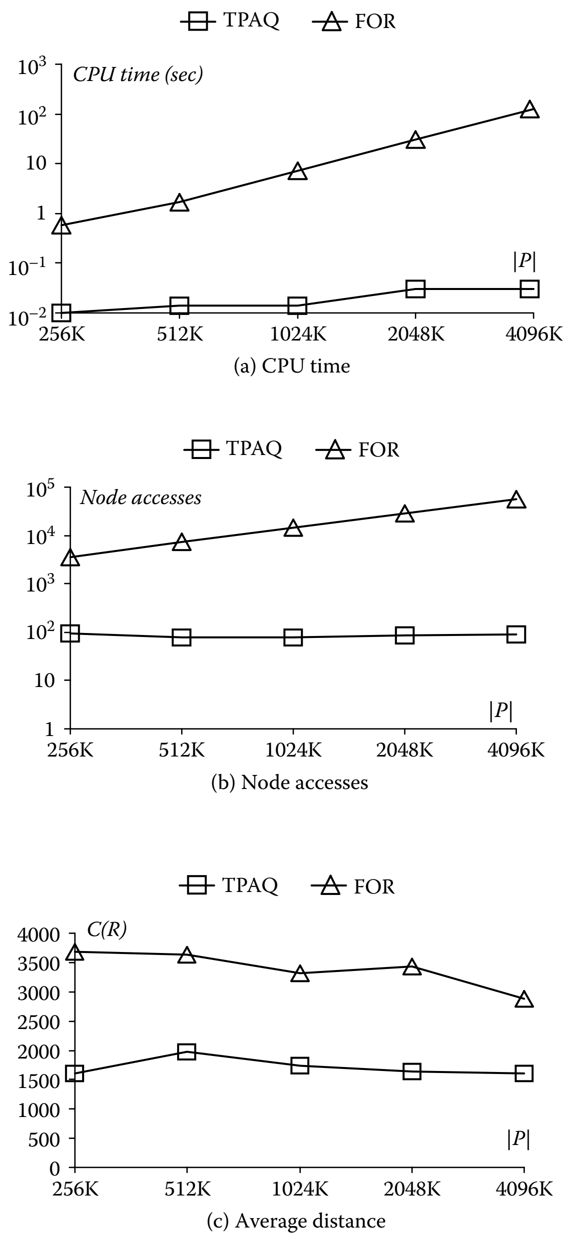
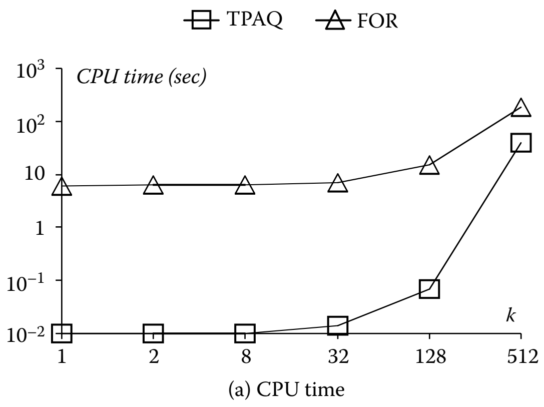
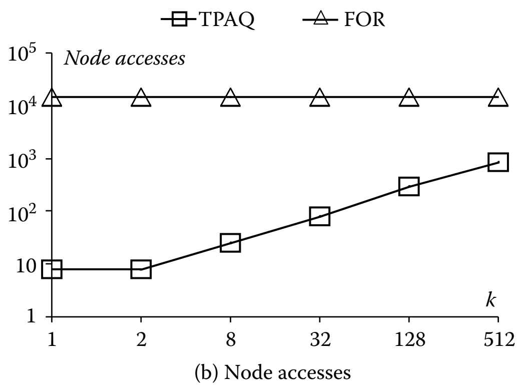


FIGURE 8.19 Performance versus $|P|$ (SKW, *max*).





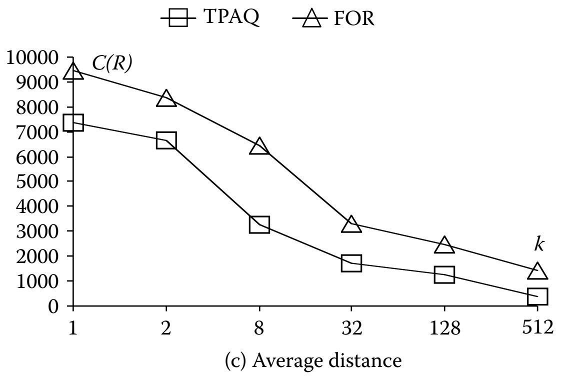


FIGURE 8.20 Performance versus *k* (SKW, *max*).

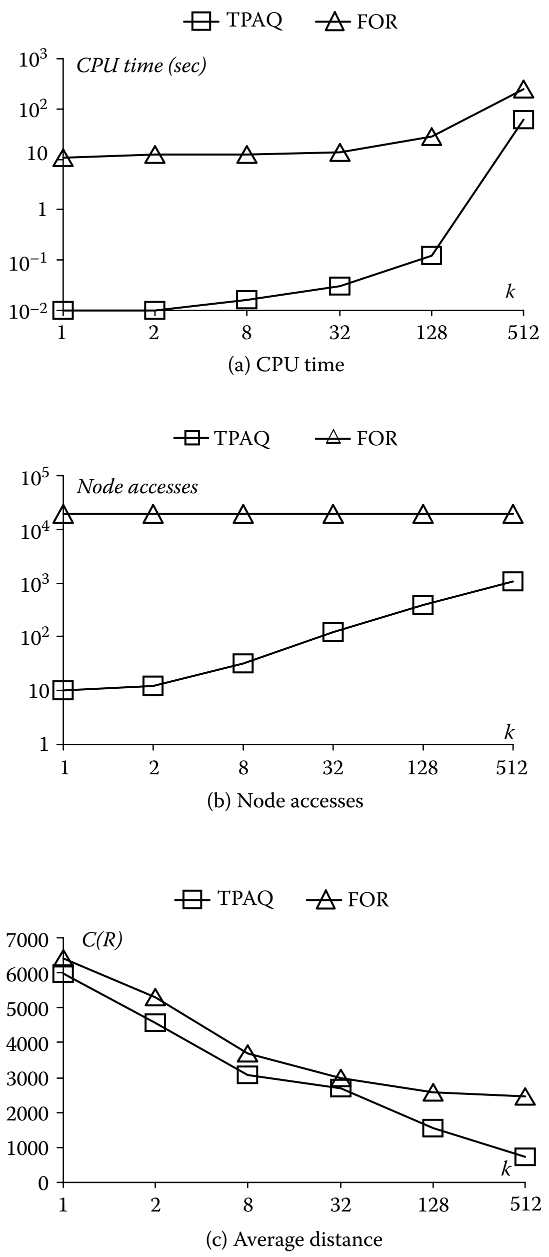


FIGURE 8.21 Performance versus *k* (LA, *max*).

of k incur higher I/O costs for TPAQ for the reasons explained in the context of Figure 8.17b. FOR performs a constant number of node accesses because it always reads the entire dataset. Regarding the quality of the returned medoid sets, our algorithm achieves much lower maximum distance $C(R)$.

8.7.2 Medoid-Aggregate Queries

In this section we study the performance of TPAQ-MA, starting with the *avg* case. We use datasets SKW (with 1M points) 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 8.22a 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)

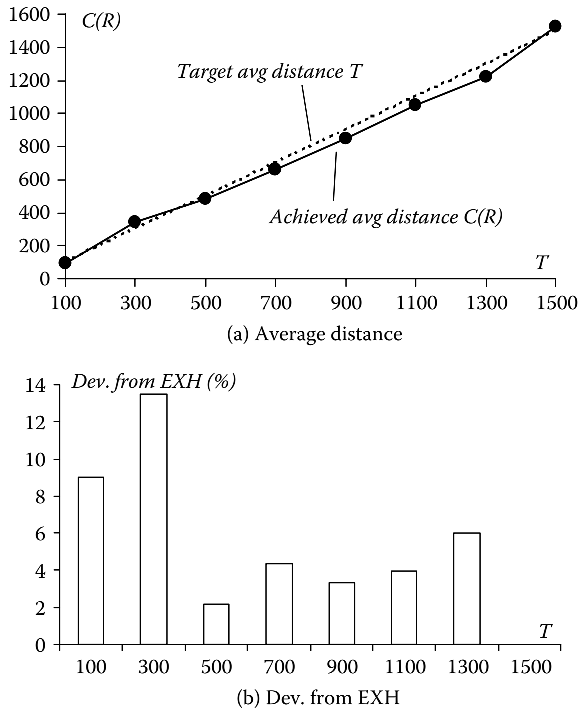
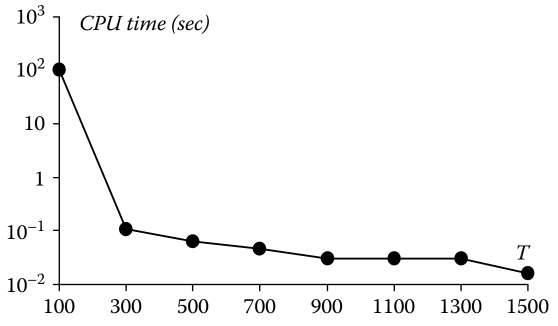
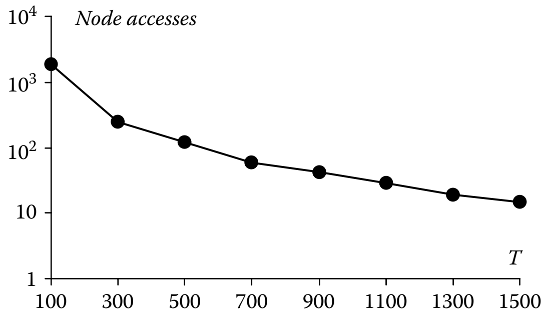


FIGURE 8.22 Performance versus T (SKW, *avg*).



(c) CPU time



(d) Node accesses

FIGURE 8.22 (Continued).

very well. Figure 8.22b 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 is equal to 13.4%. Interestingly, for $T = 1500$, TPAQ-MA returns exactly the same result as EXH with $|R| = 5$. Figure 8.22c and Figure 8.22d illustrate the CPU time and the node accesses of our method, respectively. For $T = 100$, both costs are relatively high (100.8 sec and 1839 node accesses) compared to larger values of T. The reason is that when $T = 100$, partitioning takes place at level 1 (i.e., the 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 sec and reads fewer than 251 pages.

Figure 8.23 repeats the above experiment for the LA dataset. Figure 8.23a and Figure 8.23b 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 8.22; TPAQ-MA partitions

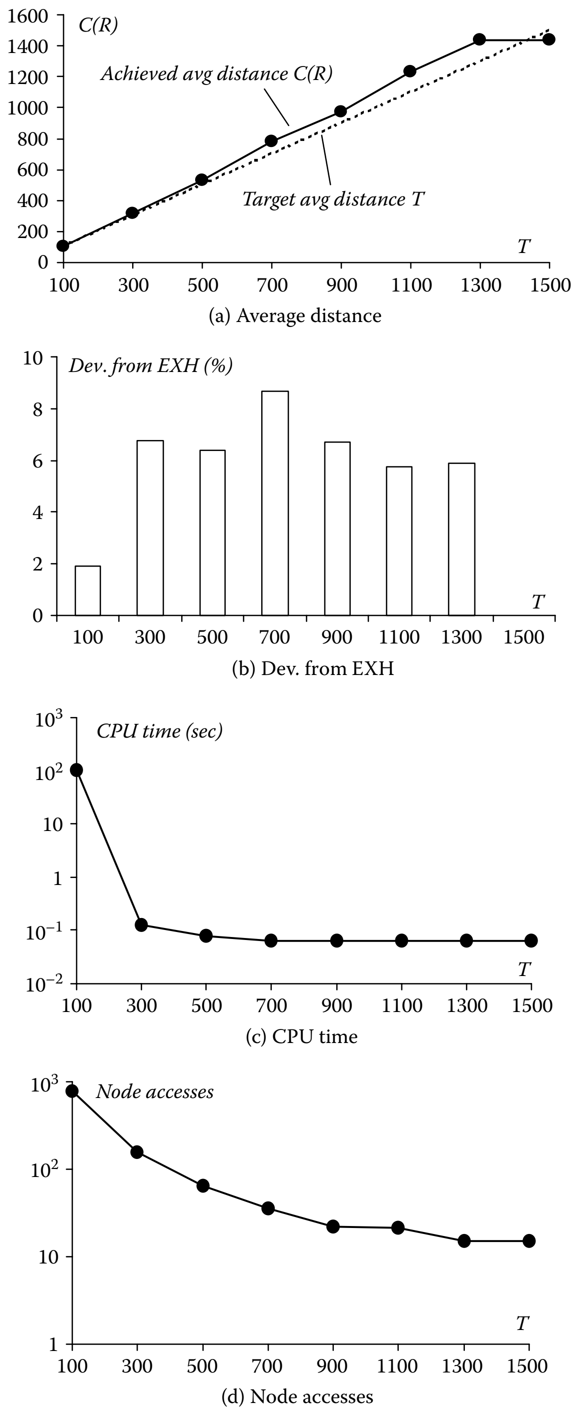


FIGURE 8.23 Performance versus T (LA, *avg*).

19,186 entries into slots and extracts $|R| = 296$ medoids, taking in total 105.6 sec and performing 781 node accesses.

In Figure 8.24 and Figure 8.25 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 several hours to terminate. As shown in Figure 8.24a and Figure 8.25a, the maximum distance of TPAQ-MA is close to the desired value T. In general, the deviation from EXH (illustrated in Figure 8.24b and Figure 8.25b) 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 sec in all cases, and incurs a small number of node accesses.

8.7.3 Medoid-Optimization Queries

Finally, we experiment on the performance of TPAQ-MO, using datasets SKW (with 1M points) and LA. We process medoid-optimization queries with $f(C(R), |R|) = C(R) + Cost_{pm} \times |R|$, 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 (1) computes all the medoid sets with $|R|$ from 1 to $|E|$, by performing partitioning into slots in the same level as TPAQ-MO, (2) calculates the (average or maximum) distance $C(R)$ achieved for each considered set, and (3) returns the one that minimizes function f.

First, we experiment on *avg* MO queries using the SKW dataset. Figure 8.26a 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. Figure 8.26b and Figure 8.26c 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 CPU time is between 147 and 157 sec 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 sec 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).

In Figure 8.27 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 Figure 8.27b and Figure 8.27c. 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 8.26.

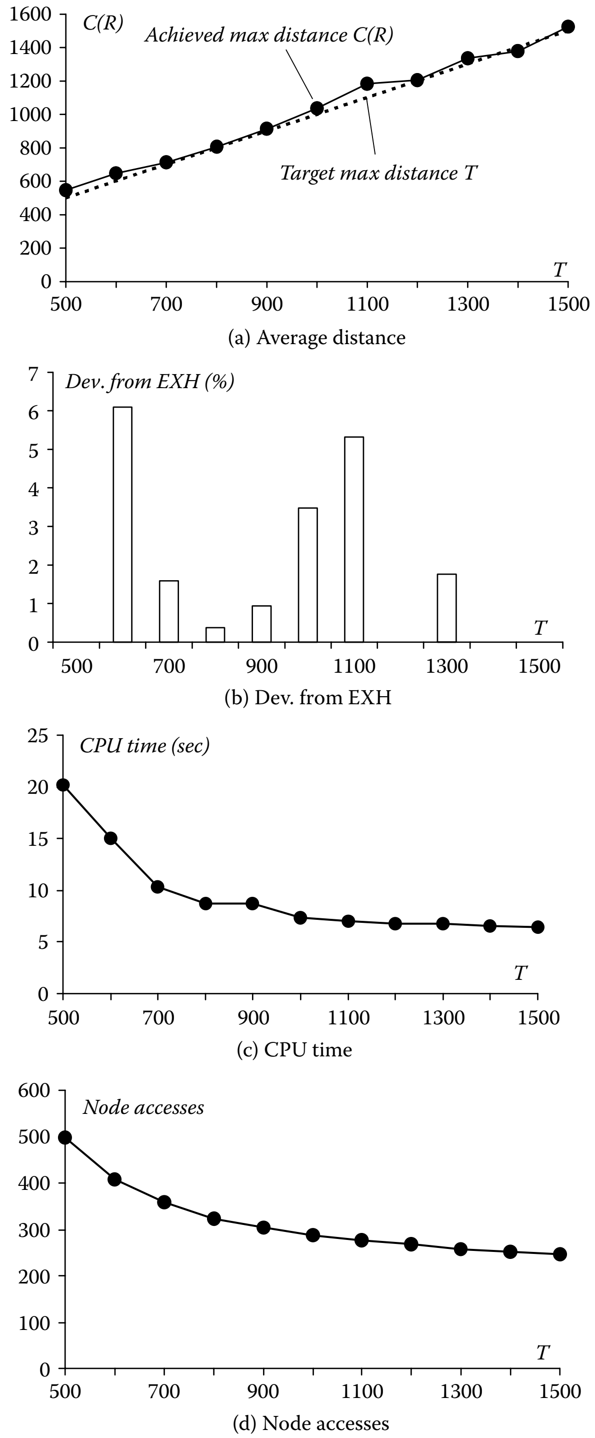


FIGURE 8.24 Performance versus T (SKW, *max*).

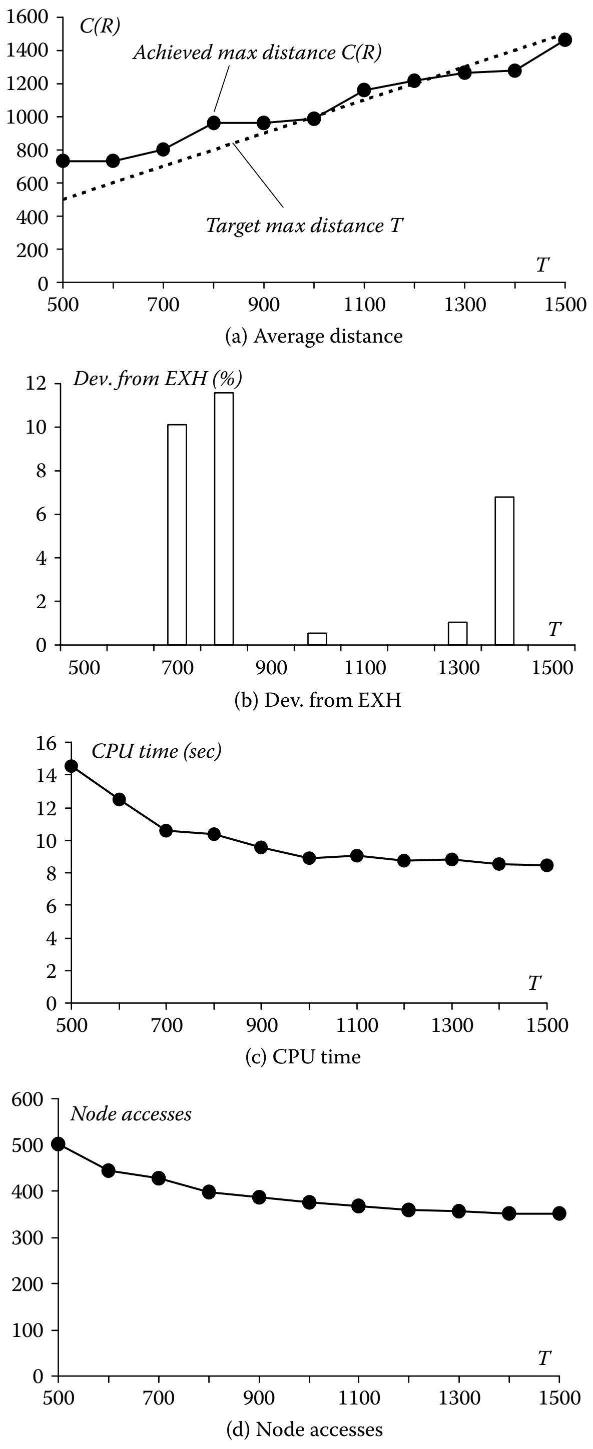


FIGURE 8.25 Performance versus T (LA, *max*).

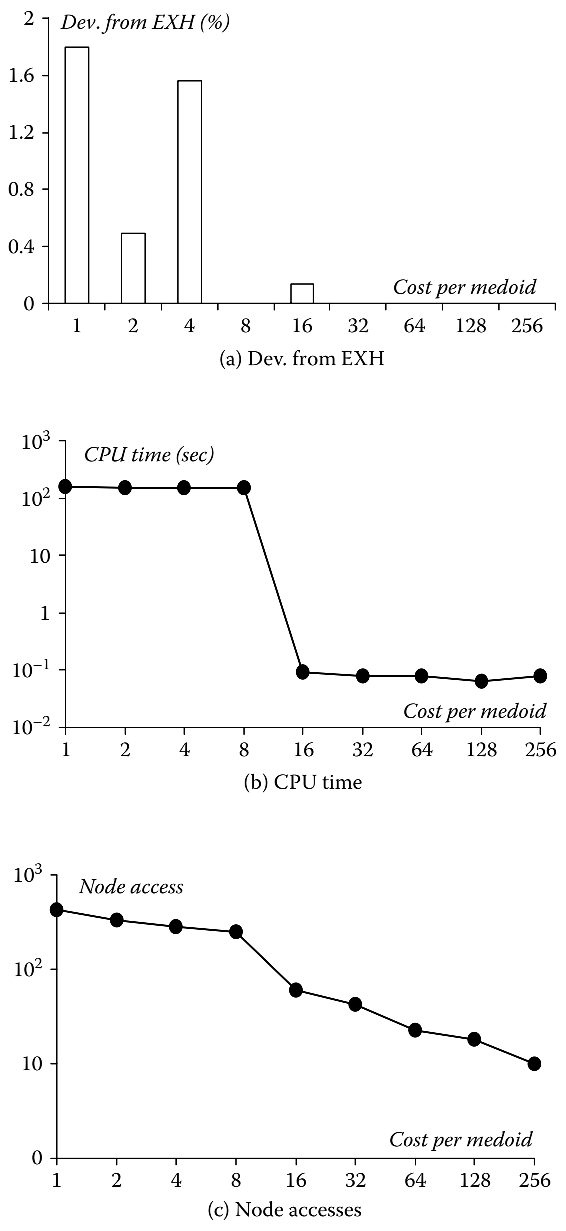


FIGURE 8.26 Performance versus $Cost_{pm}$ (SKW, *avg*).

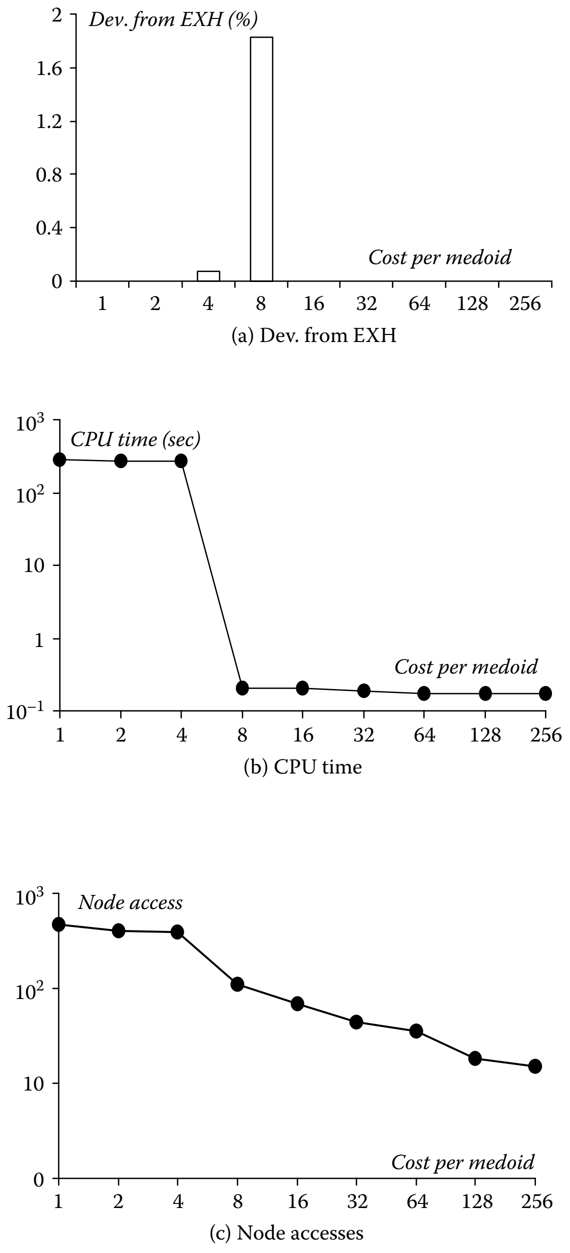


FIGURE 8.27 Performance versus $Cost_{pm}$ (LA, *avg*).

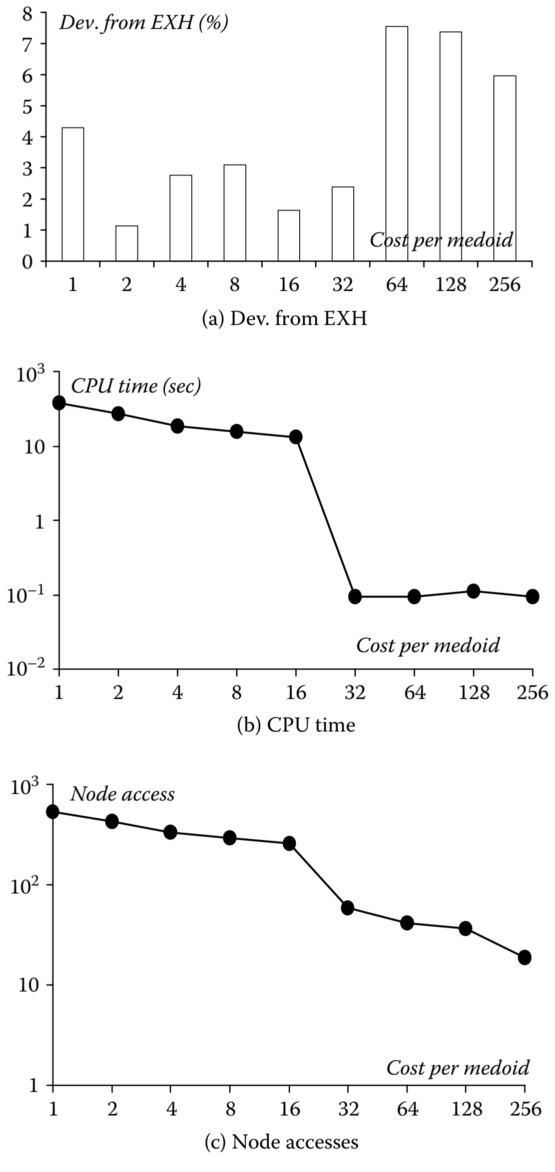
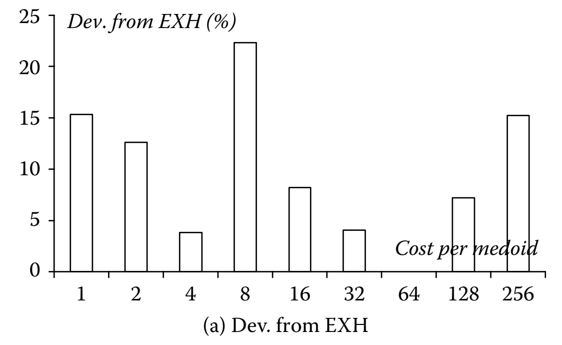
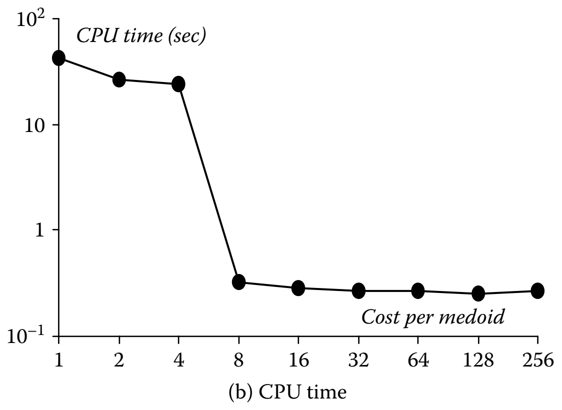


FIGURE 8.28 Performance versus $Cost_{pm}$ (SKW, *max*).

In the last two experiments we focus on *max* MO queries. Figure 8.28 and Figure 8.29 illustrate the performance of TPAQ-MO when $Cost_{pm}$ varies between 1 and 256, using datasets SKW and LA, respectively. 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





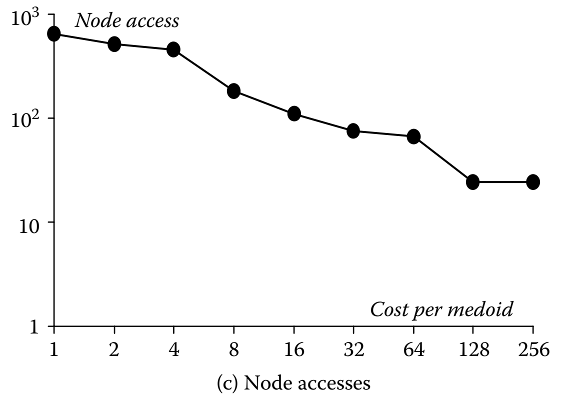


FIGURE 8.29 Performance versus $Cost_{pm}$ (LA, *max*).

areas. On the other hand, SKW (even though it is very skewed) covers the whole workspace. Concerning the CPU time of TPAQ-MO, it does not exceed 43 sec in any case. As in Figure 8.26 and Figure 8.27, both the I/O and the CPU costs drop when partitioning takes place at 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.

8.8 CONCLUSION

This chapter studies *k*-medoids and related problems in large spatial databases. In particular, we consider *k*-medoid, MA, and MO queries. We present TPAQ, a framework that efficiently processes all three query types, and is applicable to both their *avg* and *max* versions. TPAQ provides high-quality answers almost instantaneously, by exploiting the data partitioning properties of a spatial access method on the input dataset. TPAQ is a three-step methodology that works as follows. 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. Finally, it performs a NN query to retrieve one medoid for each slot. Extensive experiments with synthetic and real datasets demonstrate that (1) TPAQ outperforms the state-of-the-art method for *k*-medoid queries by orders of magnitude, while achieving results of better or comparable quality, and (2) TPAQ is also very efficient and effective in processing MA and MO queries. TPAQ relies on spatial indexing, which is known to suffer from the dimensionality curse (Korn, Pagel, and Faloutsos, 2001). A challenging direction for future work is to extend it to high-dimensional spaces, using appropriate data partition indexes (Berchtold et al., 1996).

REFERENCES

Ankerst, M., Breunig, M., Kriegel, H.P., and Sander, J. OPTICS: Ordering points to identify the clustering structure. *SIGMOD*, 1999.

Arora, S., Raghavan, P., and Rao, S. Approximation schemes for Euclidean k-medians and related problems. *STOC*, 1998.

Beckmann, N., Kriegel, H.P., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. *SIGMOD*, 1990.

Berchtold, S., Keim, D., and Kriegel, H. The X-tree: An index structure for high-dimensional data. *VLDB*, 1996.

Ester, M., Kriegel, H.P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. *KDD*, 1996.

Ester, M., Kriegel, H.P., and Xu, X. A database interface for clustering in large spatial databases. *KDD*, 1995a.

Ester, M., Kriegel, H.P., and Xu, X. Knowledge discovery in large spatial databases: focusing techniques for efficient class identification. *SSD*, 1995b.

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

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

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

Guha, S., Rastogi, R., and Shim, K. CURE: An efficient clustering algorithm for large databases. *SIGMOD*, 1998.

Hamerly, G. and Elkan, C. Learning the *k* in *k*-means. *NIPS*, 2003.

Hartigan, J.A. *Clustering Algorithms*. Wiley, 1975.

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

Hjaltason, G. and Samet, H. Distance browsing in spatial databases. *ACM TODS*, 24(2): 265–318, 1999.

Kamel, I. and Faloutsos, C. On packing r-trees. *CIKM*, 1993.

Kaufman, L. and Rousseeuw, P. *Finding Groups in Data*. Wiley-Interscience, 1990.

Korn, F., Pagel, B.U., and Faloutsos, C. On the 'dimensionality curse' and the 'self-similarity blessing'. *TKDE*, 13(1): 96–111, 2001.

Lo, M.L. and Ravishankar, C.V. Generating seeded trees from data sets. *SSD*, 1995.

Lo, M.L. and Ravishankar, C.V. The design and implementation of seeded trees: An efficient method for spatial joins. *TKDE*, 10(1): 136–151, 1998.

Mamoulis, N. and Papadias, D. Slot index spatial join. *TKDE*, 15(1): 211–231, 2003.

Moon, B., Jagadish, H.V., Faloutsos, C., and Saltz, J.H. Analysis of the clustering properties of the hilbert space-filling curve. *TKDE*, 13(1): 124–141, 2001.

Mouratidis, K., Papadias, D., and Papadimitriou S. Tree-based partition querying: a methodology for computing medoids in large spatial datasets. *VLDB Journal*, 17(4):923–945, 2008.

Ng, R. and Han, J. Efficient and effective clustering methods for spatial data mining. *VLDB*, 1994.

Pelleg, D. and Moore, A.W. Accelerating exact k-means algorithms with geometric reasoning. *KDD*, 1999.

Pelleg, D. and Moore, A.W. X-means: Extending k-means with efficient estimation of the number of clusters. *ICML*, 2000.

Roussopoulos, N., Kelly, S., and Vincent, F. Nearest neighbor queries. *SIGMOD*, 1995.

Theodoridis, Y., Stefanakis, E., and Sellis, T. Efficient cost models for spatial queries using r-trees. *TKDE*, 12(1): 19-32, 2000.

Welzl, E. Smallest enclosing disks (balls and ellipsoids). *New Results and New Trends in Computer Science*, 555: 359–370, 1991.

Zhang, D., Du, Y., Xia, T., and Tao, Y. Progressive computation of the min-dist optimal-location query. *VLDB*, 2006.

Zhang, T., Ramakrishnan, R., and Livny, M. BIRCH: An efficient data clustering method for very large databases. *SIGMOD*, 1996.