

Chapter 2

SURVEY OF EXISTING APPROACHES

In this chapter, we give an overview of existing indexes for metric spaces. Other relevant surveys on indexing techniques in metric spaces can be found in [Chávez et al., 2001b] or [Hjaltason and Samet, 2003a]. In the interests of a systematic presentation, we have divided the individual techniques into four groups. In addition we also present some techniques for approximate similarity search. Specifically, techniques which make use of ball partitioning will be found in Section 1, while Section 2 describes indexing approaches based on generalized hyperplane partitioning. A significant group of indexing methods computes distances to characteristic objects and then uses these results to organize the data. Such methods are reported in Section 3. In order to maximize performance, many approaches synergically combine several of the basic principles into a single index. The most important of these hybrid approaches are reported in Section 4. Finally, Section 5 treats the important topic of approximate similarity search, which trades some precision in search results for significant improvements in performance.

1. Ball Partitioning Methods

The advantage of ball partitioning is that it requires only one pivot and, provided the median distance d_m is known, the resulting subsets contain the same amount of data. Such a simple concept has naturally attracted a lot of attention and resulted in numerous indexing approaches being defined. In the following, we survey the most important of them. The first three structures assume discrete metric functions with a relatively small domain of values. The other methods can also be applied for continuous functions.

1.1 Burkhard-Keller Tree

Probably the first solution to support searching in metric spaces was that presented in [Burkhard and Keller, 1973]. It is called the Burkhard-Keller Tree, BKT. The tree assumes a discrete distance function and is built recursively in the following manner: From an indexed dataset X , an arbitrary object $p \in X$ is selected as the root node of the tree. For each distance $i \geq 0$, subsets $X_i = \{o \in X, d(o, p) = i\}$ are defined as groups of all objects at distance i from the root p . A child node of root p is built for every non-empty set X_i . All child nodes can be recursively repartitioned until it is no longer possible to create a new child node. When a child node is being divided, some object o_j from the set X_i is chosen as a representative of the set. A leaf node is created for every set X_i provided X_i is not repartitioned again. A set X_i is no longer split if it contains only a single object. Objects chosen as roots of subtrees (stored in internal nodes) are called pivots.

The algorithm for range queries is simple. The range search for query $R(q, r)$ starts at the root node of the tree and it compares its object p with the query object q . If p satisfies the query, that is if $d(p, q) \leq r$, the object p is returned. Subsequently, the algorithm enters all child nodes o_i such that

$$\max\{d(q, p) - r, 0\} \leq i \leq d(q, p) + r \quad (2.1)$$

and proceeds recursively downward. Observe that Equation 2.1 cuts out some branches of the tree. The inequality is a direct consequence of the lower bounds provided by Lemma 1.2 (pg. 31). In particular, by applying the lemma with $r_l = i$ and $r_h = i$, we find that the distance from q to an object o in the inspected tree branch is at least $\max\{d(q, p) - i, i - d(q, p), 0\}$. Thus, we visit the branch i if and only if $\max\{d(q, p) - i, i - d(q, p), 0\} \leq r$.

Figure 2.1b shows an example where the BKT is constructed from objects of the space illustrated in Figure 2.1a. Objects p , o_1 , and o_4 are selected as roots of subtrees, so-called pivots. The range query is given by the object q and radius $r = 2$. The search algorithm discards some branches and the accessed branches are emphasized in the figure. Obviously, if the radius of range query grows the number of accessed subtrees (branches) increases. This leads to higher search costs, which are usually measured in terms of the number of distance computations. During the range query evaluation, the algorithm traverses the tree and determines distances to pivots in internal nodes. Thus, the increasing number of accessed subtrees leads to a growing number of distance computations because pivots in individual nodes are different.

BKTs are linear in space $\mathcal{O}(n)$ and the construction complexity measured in terms of the number of distance computations is $\mathcal{O}(n \log n)$. Search time complexity, also measured in terms of distance computations, is $\mathcal{O}(n^\alpha)$, where α is a real number satisfying $0 < \alpha < 1$ which depends on the search radius and the structure of the tree, see [Chávez et al., 2001b].

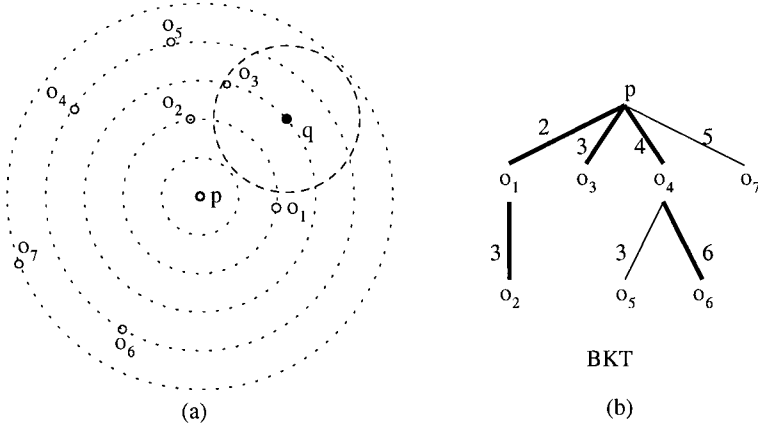


Figure 2.1. (a) An example of a metric space and a range query, (b) BKT built over the sample space.

1.2 Fixed Queries Tree

The Fixed Queries Tree, FQT, originally presented in [Baeza-Yates et al., 1994], is a modification of the BKT. In contrast to BKTs, where pivots on individual levels are different, Fixed Queries Trees use a single pivot for all nodes at the same level (see Figures 2.1b and 2.2a). All objects in a given dataset X are stored in leaves and internal nodes are used for navigation during the search (or insertion). The range search algorithm is the same as for the BKT. The advantage of this structure is a reduced number of distance computations, because even if more than one subtree has to be accessed to evaluate a query, only one distance computation between the query object and a specific pivot per level is computed. The experiments presented in [Baeza-Yates et al., 1994] confirm that FQTs need fewer distance computations than BKTs.

Figure 2.2a shows an example of an FQT built over the data of Figure 2.1a with objects p and o_4 as pivots on corresponding levels. Observe that all objects are stored in leaves, including the objects selected as pivots. The branches highlighted represent the process that evaluates the query $R(q, 2)$.

The space complexity is superlinear because the objects selected as pivots are duplicated, so the complexity varies from $\mathcal{O}(n)$ to $\mathcal{O}(n \log n)$. The number of distance computations required to build the tree is $\mathcal{O}(n \log n)$. The search complexity is $\mathcal{O}(n^\alpha)$, where α in the range $0 < \alpha < 1$ depends on the query radius and the object distribution in the metric space.

A variant of the FQT, called the Fixed-Height Fixed Queries Tree, FHFQT, is proposed in [Baeza-Yates et al., 1994, Baeza-Yates, 1997]. This structure has all its leaf nodes at the same level, i.e., leaves are at the same depth h . In other words, shorter paths are extended by additional paths. The enlargement of the

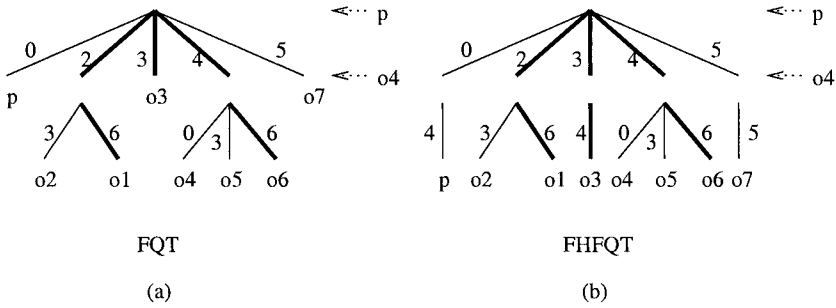


Figure 2.2. Examples of (a) FQT and (b) FHFQT built over objects of the data space depicted in Figure 2.1a.

tree can actually improve search performance, because the search process in the extended paths can be stopped before reaching the leaf. Note the distance computation to pivots for the extended paths does not typically imply extra costs, because such distances are computed due to the search needs of other (non-extended) paths. If we increase the height of the tree by thirty, we only add thirty more distance computations for the entire similarity search. We may introduce many new node traversals, but these are very cheap operations. However, thirty pivots filter out many objects, so the final candidate set is much smaller. This approach to filtering is explained in Section 7.6 of Chapter 1. For convenience, see Figure 2.2b where an example of the FHFQT is provided.

The space complexity of the FHFQT is superlinear and lies somewhere between $\mathcal{O}(n)$ and $\mathcal{O}(nh)$, where h is the height of the tree. The FHFQT is constructed with $\mathcal{O}(nh)$ distance computations. Search complexity is claimed to be constant $\mathcal{O}(h)$, that is the number of distance evaluations computed to h pivots. The extra CPU time is proportional to the number of traversed nodes and remains $\mathcal{O}(n^\alpha)$, where $0 < \alpha < 1$ depends upon the query radius and the indexed space. The extra CPU time is spent on comparing distance values (integers) and in traversing the tree. In practice, the optimal tree height $h = \log n$ cannot always be achieved due to the space limitations.

1.3 Fixed Queries Array

The Fixed Queries Array, FQA, is presented in [Chávez et al., 2001a, Chávez et al., 1999b]. Though the structure of FQA is strongly related to the FHFQT, it is not a tree structure. First, the FHFQT with height h is built on a given dataset X . If the root-to-leaf paths of the FHFQT are traversed in order from left to right and placed in an array, the result is the FQA. Each column consists of h numbers representing distances to every pivot utilized in the FHFQT. In fact, the sequence of h numbers is the path from the root of FHFQT to its leaf. The FQA structure simply stores the database objects lexicographically sorted

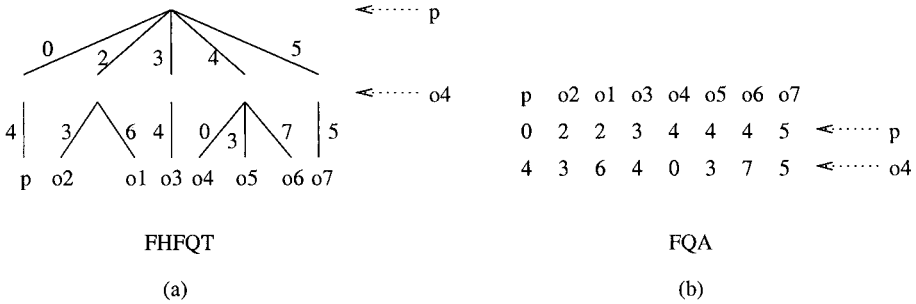


Figure 2.3. (a) An example of the FHFQT tree, (b) FQA built from the FHFQT.

by this sequence of distances. Specifically, the objects are initially sorted with respect to the first pivot and those at the same distance are sorted with respect to the second pivot and so on. For illustration, Figure 2.3b shows the FQA array constructed from the FHFQT in Figure 2.3a.

The range search algorithm is inherited from the FHFQT. Each internal node of the FHFQT corresponds to a range of elements in the FQA. Child nodes have a range of elements which is a subrange of their parents' range in the array. Naturally, there is a similarity between the FQA approach, the suffix trees, and the suffix arrays [Frakes and Baeza-Yates, 1992]. Navigation in the tree algorithm of the FHFQT is simulated by the binary search through the new range inside the current one.

The FQA is able to use more pivots than the FHFQT, which improves efficiency and search pruning. The authors of [Chávez et al., 2001a] show that the FQA outperforms the FHFQT. The space requirements are $n \cdot h \cdot b$ bits, where b is the number of bits used to store one distance. The number of distance computations evaluated during the search is $\mathcal{O}(h)$. As proved in [Baeza-Yates and Navarro, 1998], the extra CPU complexity of the FHFQT is $\mathcal{O}(n^\alpha)$. The FQA has $\mathcal{O}(n^\alpha \log n)$ extra complexity, where $0 < \alpha < 1$. The extra CPU time is due to the binary search of the array.

All the search structures presented above (BKT, FQT, FHFQT, and FQA) were designed for discrete metric functions, since a separate child is needed for any specific distance value. If we apply them to the continuous case, the tree degenerates to a flat tree of height one, and the search algorithm in effect performs a sequential scan.

In order to properly transform the continuous case to the discrete, we must segment the domain of potential distance values into a small set of subranges. Two discretizing schemata for the FQA have been proposed in [Chávez et al., 1999b, Chávez et al., 2001a]. The former divides the range of possible values into slices of identical width, the result being labeled a Fixed Slices Fixed

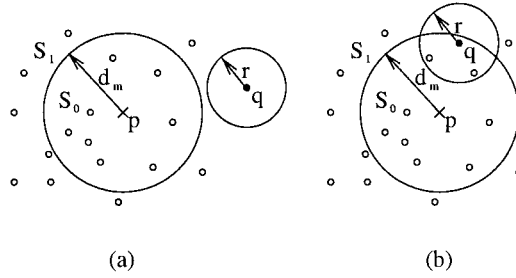


Figure 2.4. Examples of range queries: (a) S_0 is not accessed, (b) both subsets must be visited.

Queries Array. Such partitioning may lead to empty slices where no database object is accommodated. This, then, has motivated a more recent approach in which the entire range is divided into slices, each containing the same number of database objects. In other words, the domain is divided into fixed quantiles. The resulting FQA is called the Fixed Quantiles Fixed Queries Array.

1.4 Vantage Point Tree

The Vantage Point Tree (VPT) [Yianilos, 1993] is expressly designed for continuous distance functions, but discrete distance functions are also supported with virtually no modifications. It is based on the ball partitioning principle described in Section 5 of Chapter 1, which divides a set S into subsets S_1 and S_2 based upon a chosen object p called a vantage point or pivot, and the median distance d_m from p to the objects in S . Starting with the whole set of objects X and recursively applying this partitioning procedure leads to a balanced binary tree. Applying the median to divide a dataset into two subsets can be replaced by a strategy which instead employs the *mean* of distances from p to all objects in $X \setminus \{p\}$. This method, called the *middle point* in [Chávez et al., 2001b], may yield better performance for high-dimensional vector data. A disadvantage of the middle point strategy is that it may produce an unbalanced tree, impacting negatively on search algorithm efficiency.

The search algorithm for a range query $R(q, r)$ traverses the VPT from root to leaves. For each internal node, it evaluates the distance $d(q, p)$ between the pivot p and the query object q . If $d(q, p) \leq r$, the pivot p is reported to output. For internal nodes, the algorithm must also decide which subtrees to access. Doing so requires establishing lower bounds on the distances from q to objects in the left and right subtrees. If the query radius r is less than the lower bound, the algorithm does not visit the corresponding subtree. Figure 2.4a provides an example of a situation in which the inner ball region need not be accessed, whereas Figure 2.4b shows an example in which both subtrees must be checked. The lower bounds are established using Lemma 1.2 (pg. 31). More precisely, applying the equation and setting $r_l = 0$ and $r_h = d_m$, we have that the distance

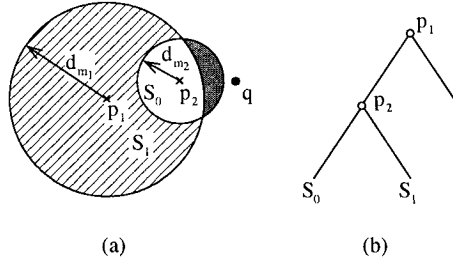


Figure 2.5. An example of VPT with two pivots p_1 and p_2 : (a) the 2-D overview and (b) the corresponding tree representation.

from q to any object in the left branch is at least $\max\{d(q, p) - d_m, 0\}$. Likewise, setting $r_l = d_m$ and $r_h = \infty$ we get that the distance from q to an object in the right subtree is at least $\max\{d_m - d(q, p), 0\}$. Thus, we enter the left branch if $\max\{d(q, p) - d_m, 0\} \leq r$ and the right branch if $\max\{d_m - d(q, p), 0\} \leq r$. Note both subtrees can be visited simultaneously.

The ball partitioning principle applied in VPTs does not guarantee that the ball region around pivot p_2 will be completely inside the ball region around pivot p_1 , which is the parent of p_2 . For convenience, see Figure 2.5 where the situation is depicted for a query object q . In general, it is possible that the lower bound from q to a child node is smaller than the lower bound from q to the child's parent node, that is

$$\max\{d(q, p_2) - d_{m2}, 0\} < \max\{d(q, p_1) - d_{m1}, 0\}.$$

But this will not affect the behavior or correctness of the search algorithm – objects rooted in the subtree of p_2 are not closer than $\max\{d(q, p_1) - d_{m1}, 0\}$, even though the lower bounds may claim the opposite. In other words, objects in the left subtree of p_2 (the set S_0) are somewhere in the white area inside the ball region of p_2 and not in the shaded region (see Figure 2.5). On the other hand, objects in the right branch (the set S_1) must be in the hatch-marked area and not outside the ball region around p_1 .

In constructing the VPT, many distance computations between pivots and objects are evaluated. For every object o in a leaf, distances are computed to each pivot p on the path from root to leaf. This information can be used to construct a more efficient search algorithm. The idea is employed in so-called VP^s trees, which are variants of VPTs proposed in [Yianilos, 1993]. Distances computed during insertion of objects are remembered and stored in the structure of the VP^s tree. They are then used in the range search algorithm as follows:

- if $|d(q, p) - d(p, o)| > r$ holds, we discard the object o without actually computing the distance $d(q, o)$,

- if $(d(q, p) + d(p, o)) \leq r$ holds, we directly include the object o in the query response set, again without computing the distance $d(q, o)$.

Given the distances $d(q, p)$ and $d(p, o)$, Lemma 1.1 (pg. 29) forms the lower and upper bounds of the actual distance between q and o :

$$|d(q, p) - d(p, o)| \leq d(q, o) \leq d(q, p) + d(p, o).$$

Thus the previous two pruning conditions are in fact direct consequences of Lemma 1.1.

Another variant of the VPT, also proposed in [Yianilos, 1993], is called the VP^{sb} tree. This tree is a further extension of the VP^s tree, where each leaf node is conceived as a bucket, that is, a unit of storage able to accommodate more than one object.

1.4.1 Multi-Way Vantage Point Tree

Figure 2.4b shows an elementary situation in which the search algorithm of the VPT must enter both subtrees and examine all objects. If such a situation occurs in many tree nodes, the global efficiency of the search deteriorates. In [Bozkaya and Özsoyoglu, 1997], the authors have tried to approach this problem by extending the binary VPT to a k -ary tree, with $k > 2$. The tree uses $k - 1$ thresholds (percentiles) $d_{m_1}, \dots, d_{m_{k-1}}$ in place of the single median d_m to partition the dataset into k subsets via spherical cuts. The modified tree is called the Multi-Way Vantage Point Tree, mw-VPT. Unfortunately, experiments reveal the performance of mw-VPTs is not always better because the spherical cuts become too thin. Take, for example, the case of high-dimensional domains where distances between any pair of objects are practically the same. The search algorithm leads to more branches of the tree being accessed during query execution. If i of k children of a node have to be searched then i distance computations are evaluated at the next level because all distances between the query object q and each pivot of the accessed children have to be determined – the VPT keeps a different pivot for each internal node at the same level.

Another extension of the VPT is called the Optimistic Vantage Point Tree, presented in [Chiueh, 1994]. This paper formulates algorithms for nearest neighbor queries and reports exhaustive performance tests on a database of image features.

These VPTs require $\mathcal{O}(n)$ space, the construction time for a balanced tree is $\mathcal{O}(n \log n)$, and search time complexity is $\mathcal{O}(\log n)$. The author of [Yianilos, 1993] claims this is only valid for very small query radii – too small to be interesting. The construction time of mw-VPT is $\mathcal{O}(n \log_k n)$ in terms of distance computations. The space complexity is the same, i.e., $\mathcal{O}(n)$. Likewise search time complexity is $\mathcal{O}(\log_k n)$.

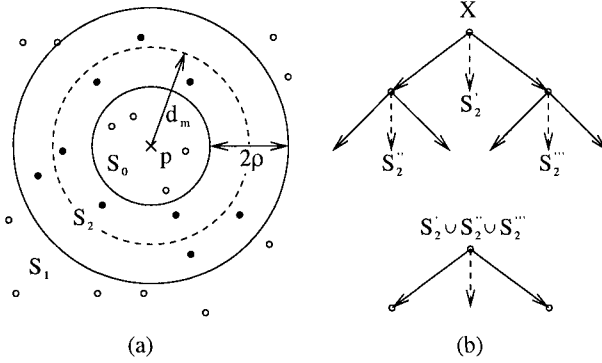


Figure 2.6. (a) An example of bp_ρ function with excluded points emphasized, (b) the VPF consisting of two trees.

1.5 Excluded Middle Vantage Point Forest

The Excluded Middle Vantage Point Forest, VPF, presented in [Yianilos, 1999], is another structure based on the ball partitioning principle. The motivation for the VPF comes from the following observation: Though the search time of the VPT [Yianilos, 1993] is sublinear, its performance depends upon not only the dataset, that is the distance distribution in X , but also on the choice of specific query object q . The VPF structure supports the worst-case sublinear search time for queries with a fixed radius up to the maximum ρ , so performance does not depend on the query object distribution. The VPF introduces a new concept of excluding objects at *middle distances* by modifying the ball partitioning technique. This principle has already been described in Section 5 of Chapter 1. For convenience, we repeat the key formula below.

$$bp_\rho(o) = \begin{cases} 0 & \text{if } d(o, p) \leq d_m - \rho \\ 1 & \text{if } d(o, p) > d_m + \rho \\ 2 & \text{otherwise} \end{cases} \quad (2.2)$$

Figure 2.6a depicts an example of the bp_ρ function, in which a dataset has been divided into two sets S_0, S_1 , with the exclusion set S_2 containing objects excluded from the partitioning process. A binary tree is built recursively by repartitioning S_0 and S_1 . The resulting exclusion sets S_2 are used to create another binary tree via the same principle. This procedure is repeated, and a forest of VPTs is produced. Figure 2.6b provides an example of the VPF. The first tree is built on the dataset X . All exclusion sets of the first tree, i.e., $\{S_2', S_2'', S_2'''\}$, are organized in the second tree. This process continues until the exclusion sets are not empty.

Excluding objects at distances near the threshold d_m has the outcome that no more than one branch of any internal node must be followed if the query

radius is less than or equal to ρ . The following tree is searched if and only if the excluded area must be visited. It is correct to have the search algorithm enter only a single subtree (left or right) because every pair of objects (x, y) such that x belongs to the left subtree and y belongs to the right, must be at a distance greater than 2ρ , that is, $d(x, y) > 2\rho$. To prove this, consider the definition of the bp_ρ function in Equation 2.2. This implies $d(x, p) \leq d_m - \rho$ and $d(y, p) > d_m + \rho$. Since the triangle inequality holds between x, y, p , we get $d(x, y) + d(x, p) \geq d(y, p)$. Combining these inequalities and simplifying, we arrive at the desired formula, $d(x, y) > 2\rho$.

The VPF is linear in $\mathcal{O}(n)$ space, with a construction time of $\mathcal{O}(n^{2-\alpha})$, where $\mathcal{O}(n^{1-\alpha})$ is the number of trees in the VPF. Similarity queries are answered in $\mathcal{O}(n^{1-\alpha} \log n)$ distance computations. In a parallel environment with $\mathcal{O}(n^{1-\alpha})$ processors, search complexity is logarithmic, $\mathcal{O}(\log n)$. The parameter $0 < \alpha < 1$ depends on ρ , the dataset, and the distance function. Unfortunately, to achieve a greater value of α , the ρ parameter must be quite small.

2. Generalized Hyperplane Partitioning Approaches

In this section, we survey methods based on an approach which is orthogonal to ball partitioning. Specifically, we focus on Bisector trees and variants on them called the Monotonous Bisector Trees and Voronoi Trees. Next, we discuss properties of Generalized Hyperplane Trees. All these techniques share a common architecture based upon generalized hyperplane partitioning.

2.1 Bisector Tree

Probably the first indexing structure to use generalized hyperplane partitioning was the Bisector Tree (BST), proposed in [Kalantari and McDonald, 1983]. The BST is a binary tree built recursively over a dataset X as follows: Two pivots p_1, p_2 are selected at each node and a hyperplane partition is applied. Objects nearer the pivot p_1 than p_2 form the left subtree, while the objects closer to p_2 create the right subtree. For each of the pivots, *covering radii* are established and stored in respective nodes. The covering radius is the maximum distance between the pivot and any object in its subtree. The search algorithm for range query $R(q, r)$ enters a subtree if $d(q, p_i) - r$ is not greater than the covering radius r_i^c of p_i . Thus, we can prune a branch if the query does not intersect the ball centered at p_i with covering radius r_i^c . The pruning condition $d(q, p_i) - r \leq r_i^c$ is correct because its modification $d(q, p_i) - r_i^c \leq r$ is a direct consequence of the lower bound of Lemma 1.2 (pg. 31) with substitutions $r_l = 0$ and $r_h = r_i^c$. From the definition of the range query, $d(q, o)$ is upper-bounded by the query radius r .

A variant of the BST, called the Monotonous Bisector Tree (MBT), has been proposed in [Noltemeier et al., 1992b, Noltemeier et al., 1992a]. The idea

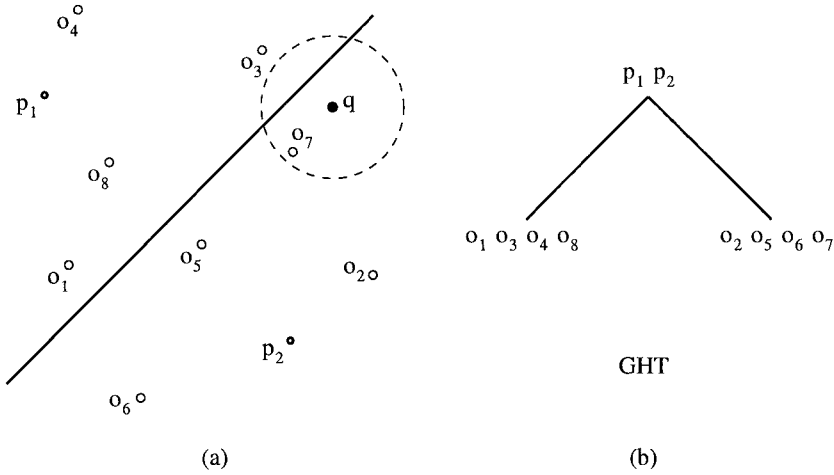


Figure 2.7. Generalized Hyperplane Tree (GHT): (a) a range query requiring access to both subsets of the hyperplane partition, (b) the corresponding structure of the tree.

behind this structure is that one of the pivots of each internal node other than the root node is inherited from its parent node. Specifically, pivots representing the left and the right subtrees are copied to the corresponding child internal nodes, respectively. This technique results in a structure with fewer pivots, and thus fewer distance computations are needed to execute a query.

BSTs are linear in space $\mathcal{O}(n)$ and require $\mathcal{O}(n \log n)$ distance computations to construct the tree. Search complexity is not analyzed by the authors.

An improvement on the BST called the Voronoi Tree (VT) is proposed in [Dehne and Noltmeier, 1987]. The VT uses two or three pivots in each internal node and also has the property that the covering radii are reduced as we move downwards in the tree. This provides better packing of objects in subtrees. The author of [Noltmeier, 1989] shows that balanced VTs can be obtained using an insertion algorithm similar to that of B-trees [Comer, 1979].

2.2 Generalized Hyperplane Tree

The Generalized Hyperplane Tree (GHT) proposed in [Uhlmann, 1991] is very similar to the BST in that both partition the dataset recursively via the generalized hyperplane principle. The difference is that the GHT does not use covering radii as a pruning criterion during the search operation. Instead, the GHT uses the hyperplane between pivots p_1 and p_2 to decide which subtrees to visit. Figure 2.7 depicts an example of the GHT. In (a), the partitioning is indicated and a range query specified. The corresponding tree structure can be seen in (b). At search time, we traverse the left subtree if $d(q, p_1) - r \leq d(q, p_2) + r$. The right subtree is visited if $d(q, p_1) + r \geq d(q, p_2) - r$ holds.

Again, note that it is possible to enter both subtrees. Observe also that the first inequality comes from Lemma 1.4 (pg. 34) and from the fact that $d(q, o) \leq r$, i.e., from the constraint given in the query specification. The second inequality is based on the same prerequisites, however, Lemma 1.4 is used in reverse, that is, the assumption about the position of o is $d(o, p_1) \geq d(o, p_2)$. A modification of the GHT that adopts the idea of reusing one pivot from the parent node, applied in MBTs, is presented in [Bugnion et al., 1993].

The space complexity of GHTs is $\mathcal{O}(n)$ and $\mathcal{O}(n \log n)$ distance computations are needed to construct the tree, the same as with BSTs. Unfortunately, search complexity was not analyzed by the authors. [Uhlmann, 1991] argues that GHTs should work better than VPTs in high-dimensional vector spaces, but no proof is provided.

3. Exploiting Pre-Computed Distances

When distance computations become expensive, a sound objective is to reduce their number to a minimum. To give efficient answers to similarity search queries, [Shasha and Wang, 1990] have suggested using pre-computed distances between data objects. For a datafile of n objects, a table of size $n \times n$ is used to store distances between data objects once computed. Pairwise distances which are not stored are estimated as intervals using the pre-computed distances. Distances unknown in advance will be, e.g., those from a query object to database objects. This technique of storing and using pre-computed distances may be effective for datasets of small cardinality. But space requirements and search complexity become overwhelming for larger files.

In this section, we discuss other techniques based on a matrix of distances between objects in a metric space. Specifically, we present the Approximating and Eliminating Search Algorithm and its linear variant. We also briefly mention other modifications or improvements, such as TLAESA, ROAESA, and Spaghettis.

3.1 AESA

The Approximating and Eliminating Search Algorithm (AESA), presented in [Vidal, 1986, Vidal, 1994], uses a matrix of distances between database objects which have been computed during the creation of the AESA structure. The structure is simply an $n \times n$ matrix holding the distances between all pairs of n database objects. Due to the symmetry property of metric functions only that half of the matrix lying below the diagonal need to be stored, resulting in $n(n-1)/2$ distances. Unlike the methods of previous sections, every object in the AESA plays the role of pivot.

The search operation for a range query $R(q, r)$ (and similarly for nearest neighbor queries) picks an object p at random and uses it as a pivot. The distance

from q to p is evaluated and used for pruning some objects. An object o can be pruned if $|d(q, p) - d(p, o)| > r$, i.e., if the lower bound from Lemma 1.1 on page 29 is greater than the query radius r . Note again that this pruning condition only utilizes distances which have already been evaluated. The algorithm then chooses another pivot from among the still remaining objects. The choice of pivot is influenced by the lower bound $|d(q, p) - d(p, o)|$. Since we want to maximize the pruning effect, we must maximize the lower bound resulting in the choice of the closest object p to q [Vidal, 1986]. The new pivot is used in the pruning condition to further eliminate some non-discarded objects. The process is repeated until the set of non-discarded objects is small enough. Finally, the remaining objects are checked directly with q , i.e., distances $d(q, o)$ are evaluated and objects satisfying $d(q, o) \leq r$ are returned.

According to experiments presented in [Vidal, 1994], AESA performs an order of magnitude better than competing methods and it is argued that it has a constant query time with respect to the size of database ($\mathcal{O}(1)$). This superior performance is obtained at the expense of quadratic space complexity $\mathcal{O}(n^2)$ and quadratic construction complexity. The extra CPU time is spent scanning the matrix, and ranges from $\mathcal{O}(n)$ up to $\mathcal{O}(n^2)$. However, we should note that one distance computation is much more expensive than one scan through the matrix. Although its performance is promising, AESA is applicable only for small datasets. If, by contrast, range queries with large radii, or nearest neighbor queries with high k are specified, AESA tends to require $\mathcal{O}(n)$ distance computations, the same as a trivial linear scan.

3.2 Linear AESA

The main drawback of the AESA approach being quadratic in space is solved in the Linear AESA (LAESA) structure [Micó et al., 1992, Micó et al., 1994]. This works around the problem by storing distances from objects to only a fixed number m of pivots. Thus, the distance matrix is $n \times m$ rather than the $n(n-1)$ entries used in the AESA. However, this has its price: a new problem arises in choosing appropriate pivots. In [Micó et al., 1994], the pivot selection algorithm attempts to choose pivots that are as far away from each other as possible, in keeping with the observations noted in Section 10.5 of Chapter 1.

The search procedure is nearly the same as in the AESA, except for the fact that some objects will not be the pivots. Thus, we cannot choose the next pivot from non-discarded objects up to now, because we might have eliminated some pivots. First, the search algorithm eliminates objects using all pivots. Then, all remaining objects are directly compared with the query object q . More details can be found in [Hjaltason and Samet, 2000], which also provides a description of the nearest neighbor search algorithm.

The space complexity and construction time of LAESA are $\mathcal{O}(mn)$, while search complexity is $m + \mathcal{O}(1)$. The extra CPU time can be reduced by a

modification called Tree LAESA (TLAESA), proposed in [Micó et al., 1996]. TLAESA builds a GHT-like structure using the same m pivots, with the extra CPU time being between $\mathcal{O}(\log n)$ and $\mathcal{O}(mn)$. The AESA and LAESA approaches are compared in [Rico-Juan and Micó, 2003].

3.3 Other Methods

A structure similar to the LAESA is proposed in [Shapiro, 1977]. It also stores mn distances in a matrix $n \times m$. However, the search procedure for $R(q, r)$ queries is slightly different. Database objects (o_1, \dots, o_n) are sorted according to their distance from the first pivot p_1 . The search starts with the object o_i such that $|d(p_1, o_i) - d(p_1, q)|$ is minimized, for $i = 1, \dots, n$. Note that this is the lower bound on $d(q, o_i)$ defined by Lemma 1.1. In other words, we start with an object potentially closest to q . The object o_i is checked against all pivots p_j ($j = 1, \dots, m$) and if $|d(p_j, o_i) - d(q, p_j)| > r$ is true for any p_j , then o_i cannot qualify for $R(q, r)$. Observe that distances $d(p_j, o_i)$ are stored in the matrix and the distances $d(q, p_j)$ are computed only once at the beginning of the query evaluation. If o_i is not eliminated by this condition, the distance $d(q, o_i)$ must be computed to decide whether o_i qualifies or not. The search continues with objects $o_{i+1}, o_{i-1}, o_{i+2}, o_{i-2}, \dots$ until the pruning conditions $|d(p_1, o_{i+c}) - d(q, p_1)| > r$ and $|d(p_1, o_{i-c}) - d(q, p_1)| > r$ are valid.

Another improvement on LAESA is a method called Spaghettis, introduced in [Chávez et al., 1999a]. This approach also stores mn distances, organized in m arrays of length n . Each array of distances to a pivot is sorted according to the distances it contains. The order of any two objects o_i, o_j may be inconsistent from one array to another, since distances to the corresponding pivots may differ e.g. $d(p_1, o_i) < d(p_1, o_j)$ and $d(p_2, o_i) > d(p_2, o_j)$. Thus, permutations of objects must be stored with respect to the preceding array. During the range search, m intervals are defined on individual arrays, $[d(q, p_1) - r, d(q, p_1) + r], \dots, [d(q, p_m) - r, d(q, p_m) + r]$. All objects that qualify for the query will belong to the intersection of all these intervals. Each object in the first interval is checked to see whether it is a member of all other intervals – the stored permutations are used for traversing through arrays of distances. Finally, the non-discarded objects are compared with the query object for qualification. The extra CPU time is reduced to $\mathcal{O}(m \log n)$.

Both AESA and LAESA have an overhead of $\mathcal{O}(n)$, measured in terms of computations other than distance evaluations (i.e., searching the matrix). The Reduced Overhead AESA (ROAESA) from [Vilar, 1995] applies heuristics to eliminate unneeded traversals of the matrix. However, this technique is only applicable to nearest neighbor queries, and the range search algorithm is not accelerated. A variant of LAESA, designated the Approximating k -LAESA (Ak-LAESA), is presented in [Moreno-Seco et al., 2003]. This variant pro-

vides a faster algorithm for kNN queries particularly designed for classification purposes.

4. Hybrid Indexing Approaches

Indexing methods which employ pre-computed distances provide promising performance boosts in terms of computational costs. Their disadvantage lies in their enormous space requirements. A straightforward remedy is to combine both the partitioning principle and the pre-computed distances technique into a single index structure. Basically this entails having search algorithms take advantage of stored pre-computed distances while traversing a hierarchy-like structure built using partitioning principles.

Such an approach is applied to Multi Vantage Point Trees, presented later in this section. We also tackle slightly different approaches based on Voronoi diagrams, namely the Geometric Near-neighbor Access Tree and the Spatial Approximation Tree. Finally, we also provide the reader with a short summary of the M-tree, a disk-based access structure which has become very popular. The M-tree and its variants are discussed in-depth in Chapter 3. In addition, we briefly mention the new concept of similarity hashing, which is again analyzed in greater depth in the next chapter.

4.1 Multi Vantage Point Tree

The Multi Vantage Point Tree (MVPT) [Bozkaya and Özsoyoglu, 1997, Bozkaya and Özsoyoglu, 1999] is an extension of the VPT. The motivation behind the MVPT is to cut down on the number of pivots used to construct a tree, since computing distances between a query object and pivots brings significant search costs. One source of motivation is the FQT described in Section 1.2. Another interesting approach to helping reduce distance computations is based on storing distances between pivots and objects in leaf nodes – such distances are computed in the course of tree construction. The extra information kept in leaves is then exploited by a sort of filtering algorithm, explained in detail in Section 7.6 of Chapter 1. The filtering algorithm dramatically decreases the number of distance computations needed to answer similarity queries.

The MVPT uses two pivots in each internal node, instead of one as in the VPT. Thus, each internal node can be considered to be two VPT levels collapsed into one node. There is one significant difference. While VPTs use different pivots at lower levels, MVPTs apply only one. Thus all children at the lower level employ the same pivot. This allows for fewer pivots while still preserving the fanout, or degree of branching. Figure 2.8 depicts a situation where a VPT is collapsed into an MVPT. Observe that some sets are partitioned using pivots that are not members of the sets. This never occurs in VPTs. In Figure 2.8b, so is the set around p_1 which is divided using p_2 and the radius d_{m_3} . In this

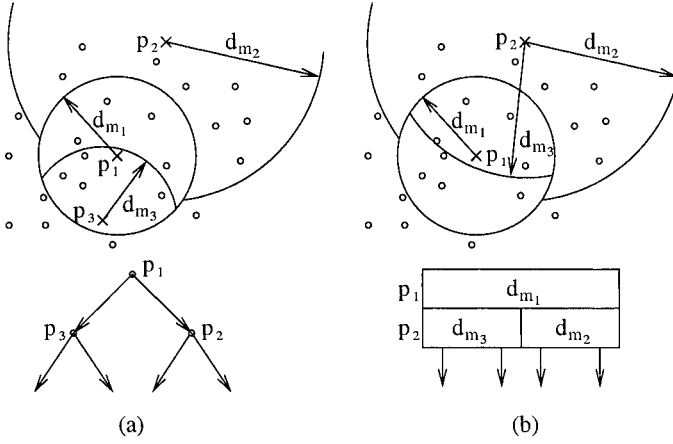


Figure 2.8. Comparison of the VPT and MVPT structures: (a) VPT with three pivots for partitioning to four sets, (b) MVPT using only two pivots.

case, each pivot leads to two subsets, which implies that the fanout of an MVPT node is 2^2 . Since a pivot can generally partition data into m subsets, an internal node can root m^2 child nodes. In addition, MVPT can employ k pivots in each internal node, which implies a fanout of m^k . Moreover, each object in the leaf node is associated with a list of distances to the first l pivots, which are used for additional pruning at search time.

Since no objects are duplicated, space complexity is $\mathcal{O}(n)$ – objects chosen as pivots appear only in internal nodes. However, MVPTs need some extra space to keep l pre-computed distances for each object in leaves. Construction time complexity is $\mathcal{O}(nk \log_{m^k} n)$, where $\log_{m^k} n$ is the height of the balanced tree. Search complexity is $\mathcal{O}(k \log_{m^k} n)$, but is valid only for very small query radii. In the worst case, search complexity will be $\mathcal{O}(n)$. The authors of [Bozkaya and Özsoyoglu, 1999] show experimentally that MVPTs outperform VPTs, which they mainly attribute to the greater number of pivots in internal nodes rather than the increased fanout m . The largest performance boosts are achieved by storing more pre-computed distances in leaves.

4.2 Geometric Near-neighbor Access Tree

The Geometric Near-neighbor Access Tree (GNAT), proposed by [Brin, 1995], uses m pivots in each internal node. Specifically, a set of pivots $P = \{p_1, \dots, p_m\}$ is chosen and the dataset X is split into S_1, \dots, S_m subsets, depending on the shortest distance to a pivot in P . In other words, for any object $o \in X - P$, o is a member of the set S_i if and only if $d(p_i, o) \leq d(p_j, o)$ for all $j = 1, \dots, m$. Thus, applying this procedure recursively we build an m -ary tree. Figure 2.9 shows a simple example of the first level of a GNAT structure.

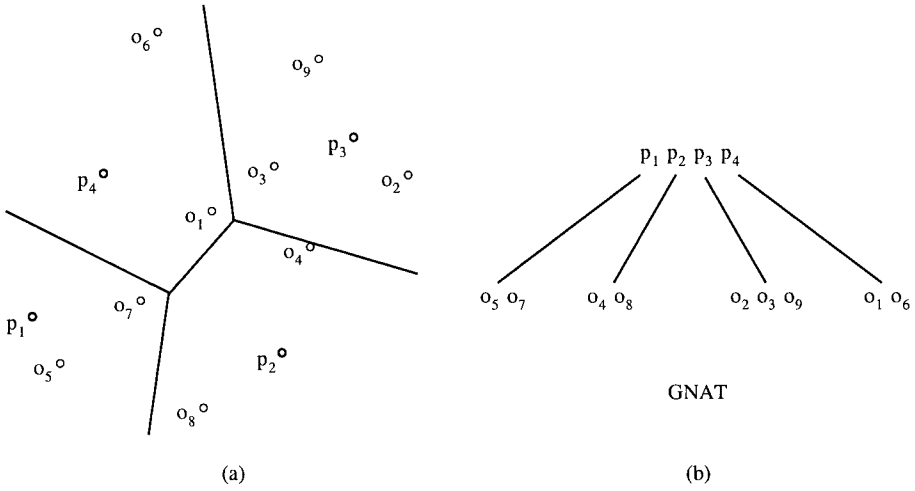


Figure 2.9. The Geometric Near-neighbor Access Tree: (a) an example of partitioning, (b) the corresponding tree.

Observe the close relationship between this idea and the Voronoi-like partitioning of vector spaces [Aurenhammer, 1991]. Each subset S_i corresponds to a cell in the Voronoi diagram – GNAT calls this cell the Dirichlet domain. The parameter m is adjusted according to the level of the tree. In fact, the number of children (i.e., the value of m) should be proportional to the number of data objects allocated in the node.

Besides applying the m -ary partitioning principle, the GNAT also retains objects' distances to their respective pivots. This enables additional pruning during the search, resulting in a range search algorithm quite different from the one used for the GHT. In each internal node, an $m \times m$ table consisting of distance ranges is stored. Specifically, the minimum and maximum distances between each pivot p_i and the objects of each subset S_j are stored. Formally, the range $[r_l^{ij}, r_h^{ij}]$, $i, j = 1, \dots, m$, is defined as follows:

$$r_l^{ij} = \min_{o \in S_j \cup \{p_j\}} d(p_i, o),$$

$$r_h^{ij} = \max_{o \in S_j \cup \{p_j\}} d(p_i, o).$$

Note that the lower bound r_l^{ii} for pivot p_i itself is equal to zero, since the minimum is at distance $d(p_i, p_i) = 0$. Figure 2.10 illustrates two ranges. The first $[r_l^{ij}, r_h^{ij}]$ is defined for pivot p_i and set S_j around pivot p_j , while the second is $[r_l^{jj}, r_h^{jj}]$ for pivot p_j itself.

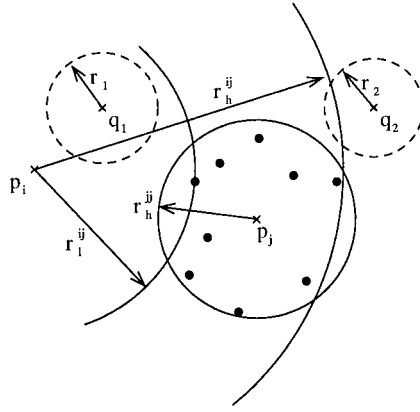


Figure 2.10. An example of the pruning effect of ranges in GNAT for two queries $R(q_1, r_1)$ and $R(q_2, r_2)$.

The range search algorithm for query $R(q, r)$ proceeds depth-first. In each internal node N , the distances between q and the pivots of N are computed and subtrees not containing qualifying objects are eliminated. After all distances from q to pivots have been computed, the algorithm visits all subtrees that remain. Starting with the set of pivots P , the procedure applied in each internal node is described in the following steps: First, pick one pivot p_i from P (repeatedly, but do not pick the same pivot twice) and compute the distance $d(p_i, q)$. If $d(p_i, q) \leq r$ holds, the pivot p_i is returned in the query result. Afterwards, for all $p_j \in P$ we remove p_j from P if $d(q, p_i) - r > r_h^{ij}$ or $d(q, p_i) + r < r_l^{ij}$. The inequalities are direct consequences of the lower bound $\max\{d(q, p_i) - r_h^{ij}, r_l^{ij} - d(q, p_i)\} \leq r$ of Lemma 1.2 (pg. 31) with $d(q, o) \leq r$. When all pivots in P are examined, the subtrees of the node N corresponding to the remaining pivots in P are visited. Note that a pivot p_j can be discarded from P before its distance to q has been evaluated. Figure 2.10 depicts a situation in which two range queries $R(q_1, r_1)$ and $R(q_2, r_2)$ are given. In this example, only the range $[r_l^{ij}, r_h^{ij}]$ is sufficient for the query $R(q_1, r_1)$ to discard p_j . However, the other query requires the additional range $[r_l^{jj}, r_h^{jj}]$ to prune the subtree around p_j .

The space complexity of the GNAT index structure is $\mathcal{O}(nm^2)$, because tables consisting of m^2 elements are stored in each internal node. GNAT is built in $\mathcal{O}(nm \log_m n)$ time. The search complexity was not analyzed by the authors, but experiments in [Brin, 1995] reveal that the GNAT outperforms the GHT and VPT structures.

4.3 Spatial Approximation Tree

The indexes which have been described so far all use a partitioning principle to recursively divide the data space into subsets. For example, the GHT and GNAT are inspired by the Voronoi-like partitioning. In the following, we introduce the Spatial Approximation Tree, the sa-tree (SAT), proposed in [Navarro, 1999, Navarro, 2002]. The SAT is also based on the Voronoi diagrams, but in contrast to the GHT and GNAT it tries to approximate the structure of the *Delaunay graph*. Given a Voronoi diagram, a Delaunay graph, defined in [Navarro, 2002], is a graph where each node represents one cell of the Voronoi diagram and where nodes are connected with edges if the corresponding Voronoi cells are directly neighboring cells. In other words, the Delaunay graph is a representation of relations between cells in the Voronoi diagram. In the following, we use the term object for a node of the Delaunay graph and vice versa.

The search algorithm for the nearest neighbor of a query object q starts with an arbitrary object (node in the Delaunay graph) and proceeds to a neighboring object closer to q as long as it is possible. If we reach an object o where all neighbors of o are further from q than o , the object o is the nearest neighbor of q . The correctness of this simple algorithm is obvious. Unfortunately, it is possible to show that without more information about a given metric space $\mathcal{M} = (\mathcal{D}, d)$, knowledge of the distances between objects in a finite set $X \subseteq \mathcal{D}$ does not uniquely determine the Delaunay graph for X (for further details see [Navarro, 2002, Hjalton and Samet, 2003a]). Thus, the only way to ensure the search procedure is correct is to use a complete graph, that is, the graph containing all edges between all pairs of objects in X . However, such a graph is not suitable for searching because the decision as to which edge should be traversed from the starting object requires computing distances from the query to all remaining objects in X . This boils down to a linear scan of all objects in the database and thus, from a searching point of view, is useless.

For a dataset X , the SAT is defined as follows: An arbitrary object p is selected as the root of the tree and the smallest possible set $N(p)$ of all its neighbors is determined so that:

$$o \in N(p) \Leftrightarrow \forall o' \in N(p) \setminus \{o\} : d(o, p) < d(o, o').$$

The intuition behind this definition is that for a valid set $N(p)$ (not necessarily the smallest), each object of $N(p)$ is closer to p than to any other object in $N(p)$ and all objects in $X \setminus N(p)$ are closer to an object in $N(p)$ than to p . Figure 2.11b shows an example of SAT built on a dataset depicted in Figure 2.11a. The object o_1 has been selected as the root node. The set of neighbors for o_1 is $N(o_1) = \{o_2, o_3, o_4, o_5\}$. Note that object o_7 cannot be included in $N(o_1)$ since o_7 is closer to o_3 than to o_1 .

To build the tree, a child node is defined for every neighbor and the objects nearest the child are structured in the same way as defined above. The distance

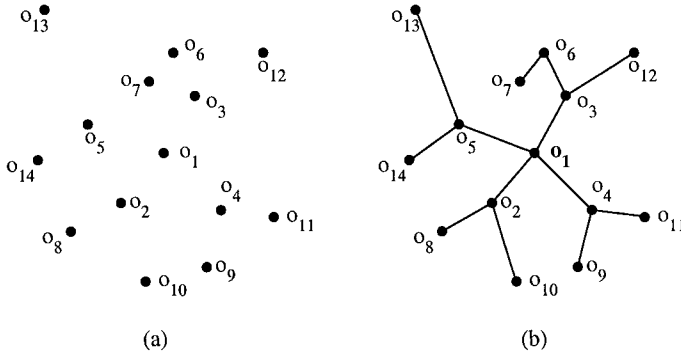


Figure 2.11. An example of SAT: (a) the dataset, (b) SAT structure with the root o_1 .

to the furthest object o from p is also stored in each node, i.e., for the root node, $\max_{o \in X} \{d(p, o)\}$. In conventional terminology, it is the covering radius r^c .

As argued in [Navarro, 2002], the construction of $N(p)$ is NP-complete, so a heuristics is proposed which builds the set $N(p)$ in a way which may not be minimal. The method of selecting the set of neighbors influences the shape of the resulting tree. When the set is not minimal the fanout of the tree increases, which impacts upon search costs. The heuristics starts with an object p , a set $S = X \setminus \{p\}$, and initially empty set $N(p)$. We first sort the members of S with respect to their distance to p . Next, we pick an object o from S and add it to $N(p)$ if it is closer to p than any other object in $N(p)$. In this fashion, we incrementally construct a suitable set of neighbors.

The range search algorithm for the query $R(q, r)$ starts at the root node and traverses the tree, visiting all non-discardable subtrees. Recall that at the node p , we have the set of all neighbors $N(p)$. The algorithm first finds the closest object $o_c \in N(p) \cup \{p\}$ to q . Then, it discards all subtrees $o_d \in N(p)$ such that

$$d(q, o_d) > 2r + d(q, o_c). \quad (2.3)$$

Such a pruning criterion is correct and is a consequence of Lemma 1.4 (pg. 34) with substitutions $p_1 = o_d$ and $p_2 = o_c$. In particular, we get

$$\max\left\{\frac{d(q, o_d) - d(q, o_c)}{2}, 0\right\} \leq d(q, o).$$

Providing that $d(q, o) \leq r$ (the range query constraint) and q is closer to o_c than to o_d we get $(d(q, o_d) - d(q, o_c))/2 \leq r$. The branch o_d can easily be pruned if $(d(q, o_d) - d(q, o_c))/2 > r$, which is exactly what we desired.

The reason we select the closest object o_c to q is we want to maximize the lower bound of Lemma 1.4. When the current node is not the root of tree, we can even improve the pruning effect. Figure 2.12 depicts a sample SAT

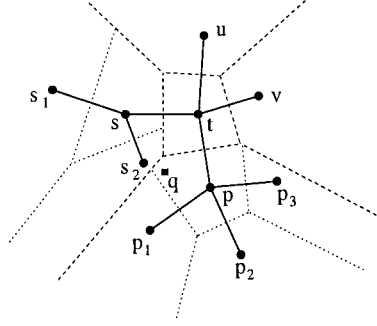


Figure 2.12. A sample of the SAT structure.

with root t , current node p (with neighbors p_1, p_2, p_3), and query object q . The dashed lines represent the boundaries between Voronoi cells of the first level of the SAT. The dotted lines depict the same for the second level. Assuming the current node is p , the algorithm presented above selects p among $\{p, p_1, p_2, p_3\}$ as the closest object to q , even though the object s_2 is closer. If we choose s_2 as the closest object, we further strengthen the pruning effect. However, this requires modifying the procedure for picking the closest object as follows: Select the closest object o_c from p 's ancestor, including its neighbors and their associated neighbors, i.e., $o_c \in \bigcup_{o \in A(p)} (N(o) \cup \{o\})$. Here, $A(p)$ consists of the ancestors of p and its neighbors – in the figure, $A(p) = \{t, p, s, u, v\}$. Finally, the covering radius r^c of each node is used to further reduce search costs. We do not visit a node p if $d(q, p) > r^c + r$. This expression is derived from the lower bound in Lemma 1.2 (pg. 31) with $r_l = 0, r_h = r^c$ and from the fact that $d(q, o) \leq r$. The search algorithm is correct and returns all qualifying objects regardless of the strategy of selecting the closest object o_c . In other words, the strategy only influences the efficiency of pruning, see Equation 2.3.

The tree is built in $\mathcal{O}(n \log n / \log \log n)$ time, takes $\mathcal{O}(n)$ space and its search complexity is $\Theta(n^{1-\Theta(1/\log \log n)})$. The SAT is designed as a static structure. More details can be found in [Navarro, 1999, Hjalton and Samet, 2003a, Navarro, 2002]. A dynamic version of SAT is presented in [Navarro and Reyes, 2002].

4.4 M-tree

A dynamic structure called the Metric Tree (M-tree) is proposed in [Ciaccia et al., 1997b]. It can handle data files that change size dynamically, which becomes an advantage when insertions and deletions of objects are frequent. In contrast to other metric trees, the M-tree is built bottom-up by splitting its fixed-size nodes. Each node is constrained by sphere-like (ball) regions of the metric space. A leaf node entry contains an identification of the data object,

its feature value used as an argument for computing distances, and its distance from a routing object (pivot) that is kept in the parent node. Each internal node entry keeps a child node pointer, the covering radius of the ball region that bounds all objects indexed below, and its distance from the associated pivot. Obviously, the distance to the parent pivot has no meaning for the root. The pruning effect of search algorithms is achieved by using the covering radii and the distances from objects to their pivots in parent nodes.

Dynamic properties in storage structures are highly desirable but typically have a negative effect on performance. Furthermore, the insertion algorithm of the M-tree is not deterministic, i.e., inserting objects in different order results in different trees. That is why the *bulk-loading* algorithm has been proposed in [Ciaccia and Patella, 1998]. The basic idea of this algorithm works as follows: Given a set of objects, the initial clustering produces k sets of relatively close objects. This is done by choosing k distant objects from the set and making them representative samples. The remaining objects get assigned to the nearest sample. Then, the bulk-loading algorithm is invoked for each of these k sets, resulting in an unbalanced tree. Special refinement steps are applied to make the tree balanced.

The idea of M-trees was later extended by [Traina, Jr. et al., 2000b] in a metric tree structure called the Slim-tree. In order to get control over the overlap between metric regions, the fat-factor is defined and systematically used. The concept of fat-factor has been described in detail in Section 10.4 of Chapter 1. Slim-trees also use new insertion and split algorithms which result in improved performance. Slim-trees and many other variants of M-trees are described in Chapter 3.

4.5 Similarity Hashing

Similarity Hashing (SH), as proposed in [Gennaro et al., 2001] is built upon a completely different principle. It is a multi-tier hashing structure, consisting of search-separable sets on each tier, organized in buckets. The structure supports easy insertion and bounded search costs, because at most one bucket need to be accessed at each level for range queries up to a pre-defined value of the search radius. At the same time, the number of distance computations is always significantly reduced by the use of pre-computed distances obtained at insertion time. Buckets of static files can be arranged in such a way that I/O costs never exceed the cost of scanning a compressed sequential file. Experimental results demonstrate the performance of SH is superior to other available tree-based structures.

The similarity hashing approach is exploited in the so-called the D-index structure [Dohnal et al., 2003a]. The D-index applies excluded middle partitioning to hashed organizations. In contradistinction to VPF, navigation along the tree branches is unnecessary, and each storage bucket is directly accessible.

In principle, the concept of similarity hashing is not necessarily restricted to the excluded middle partitioning principle. [Dohnal et al., 2001] define another three ρ -split functions that are able to achieve the same effect, i.e., to produce sets separable up to a pre-defined distance radius ρ . Based on well-known geometric concepts, these methods are called the *elliptic*, *hyperbolic*, and *pseudo-elliptic* ρ -split functions. The second section of Chapter 3 deals with the D-Index and its variants suitable for similarity joins, and further extends the description of similarity hashing.

5. Approximate Similarity Search

Some applications can benefit from a very fast response to similarity queries even when it is obtained at the expense of precision in results. The fundamental concepts have already been discussed in Section 9 of Chapter 1. In the following, we survey some interesting approaches that have been proposed in the literature.

First, we briefly cover certain approximation techniques that exploit space transformations. Then we provide a more extensive presentation of techniques which reduce the subset of data that must be examined. Most of these techniques were originally applied to vector spaces, but some can also be used in generic metric spaces.

5.1 Exploiting Space Transformations

Space transformations are convenient to use for approximate similarity search. This has already been mentioned in Section 9 of Chapter 1. Obviously the transformations must satisfy the constraints described in Section 8 of Chapter 1. The general strategy is as follows: First, the original space is transformed. Then all search requests are executed in the projected space. Some false hits may be returned – but approximate similarity search algorithms do not apply the final cleansing phase which is necessary for obtaining exact results.

An approach to dimensionality reduction specifically designed for approximate similarity searching has been proposed in [Egecioglu and Ferhatosmanoglu, 2000]. The authors propose a dimensionality reduction technique that offers an easy way to compute the inner product between vectors approximately. Given a vector $\vec{z} = (z_1, \dots, z_d)$, let $\psi_p(\vec{z})$ denote L_p norm to the p -th power. Then $\psi_p(\vec{z}) = (\|\vec{z}\|_p)^p = [L_p(\vec{z}, \vec{0})]^p$, where $\vec{0} = (0, \dots, 0)$. The inner product of two vectors $\langle x, y \rangle$ can be approximated with the estimate of its m -power as $\langle \vec{x}, \vec{y} \rangle^m \approx b_1 \psi_1(\vec{x}) \psi_1(\vec{y}) + \dots + b_m \psi_m(\vec{x}) \psi_m(\vec{y})$, where $m < d$ and b_1, \dots, b_m are parameters that should be tuned to obtain a good approximation. This technique saves disk space by storing the m -dimensional vector $(\psi_1(\vec{x}), \dots, \psi_m(\vec{x}))$ instead of the d -dimensional vector \vec{x} , given that the approximate inner product can be computed using it. It also allows the

Euclidean distance $\|\vec{x} - \vec{y}\|_2$ to be also approximated, given that

$$\|\vec{x} - \vec{y}\|_2 = \sqrt{\psi_2(\vec{x}) + \psi_2(\vec{y}) - 2\langle \vec{x}, \vec{y} \rangle}.$$

In [Ogras and Ferhatosmanoglu, 2003], this approximation method is further refined as follows: The d -dimensional space is divided into orthogonal subspaces S_1, \dots, S_s each having l dimensions, $l = d/s$. Let xP_i be the projection of vector \vec{x} in the subspace S_i . The Euclidean distance between \vec{x} and \vec{y} can be computed exactly as $\|\vec{x} - \vec{y}\|_2 = \|\vec{x}P_1 - \vec{y}P_1\|_2 + \dots + \|\vec{x}P_s - \vec{y}P_s\|_2$. If the individual $\|\vec{x}P_i - \vec{y}P_i\|_2$ are separately approximated using the approximate inner product technique, the approximation of the entire Euclidean distance which results is more precise. The authors note that the basic inner product approximation retains information on the magnitude of vectors only. A refined technique, also based on the space decomposition, is able to additionally retain information about the shape of approximated vectors, i.e., their direction.

A further approach to space transformation is presented in [Weber and Böhm, 2000], based on so-called Vector Approximation files, VA-files. The VA-file [Weber et al., 1998] reduces the size of multi-dimensional vectors by quantizing the original data objects. It demands a nearest neighbor search performed in two steps. Initially, the approximated vectors are scanned to identify candidate vectors. Then, in the second step, the candidate vectors are visited in order to find the actual nearest neighbors. The approximate search variant on this algorithm basically omits the second step of the exact search. A modification of the VA-files approach has been proposed in [Ferhatosmanoglu et al., 2000] in which the VA-file building procedure is improved by initially transforming the data into a more suitable domain using the Karhunen-Loeve transform, KLT. An approximate search algorithm based on the modified VA-file approach is proposed in [Ferhatosmanoglu et al., 2001]. The performance improvement offered by techniques based on VA-files is significant. However, they are applicable to vector spaces only.

A final approach which falls into this category is FastMap [Faloutsos and Lin, 1995]. This technique is also suitable for generic metric spaces, provided we have k feature-extraction functions which transform the metric space into a k -dimensional space. A similar technique which is, however, applicable directly to metric spaces is called MetricMap [Wang et al., 2000] and has already been discussed in Section 8.3 of Chapter 1.

5.2 Approximate Nearest Neighbors with BBD Trees

Suppose we have a query object q and a dataset X represented in a vector space whose distances are measured by Minkowski distance functions. Arya et al. [Arya et al., 1998] propose an approximate nearest neighbor algorithm which guarantees to find $(1+\epsilon)$ - k -approximate-nearest-neighbors. Specifically, it retrieves k objects whose distances from the query are at most $1 + \epsilon$ times

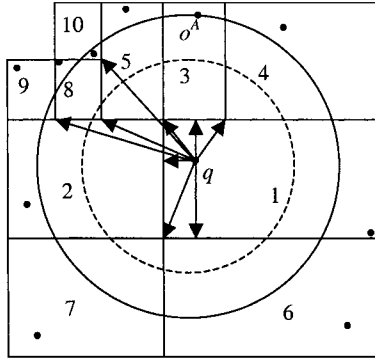


Figure 2.13. Overview of the approximate nearest neighbors search algorithm using BBD trees.

larger than that of the k -th actual nearest neighbor of q . The time complexity of this algorithm is $\mathcal{O}(k \log n)$, where n is the size of the dataset X . The parameter ϵ is used to control the tradeoff between the efficiency and quality of the approximation. The higher the value of ϵ , the higher the performance and error.

As its underlying indexing structure, the algorithm uses the *Balanced Box-Decomposition tree* (BBD) that is a variant of the Quad-tree [Samet, 1984] and is similar to other balanced structures based on box-decomposition [Bern et al., 1993, Bspamyatnikh, 1995, Callahan and Kosaraju, 1995]. A property of the BBD tree is that regions associated with nodes which have the same parent do not overlap. Node regions are recursively repartitioned until they contain only one object, thus every region associated with a leaf node contains just a single object. The tree has $\mathcal{O}(n)$ nodes and is built in $\mathcal{O}(dn \log n)$ time, where d is the number of dimensions of the vector space.

The nearest neighbor algorithm associated with this data structure proceeds as follows: Given a query object q , the tree is traversed and the unique leaf node associated with the region containing the query is found in $\mathcal{O}(\log n)$. At this point, a priority search is performed by enumerating leaf regions in increasing order of distance from the query object. The distance from an object o to a region is computed as the distance of o to the closest point that can be contained in the region. When a leaf region is visited, the distance of the associated object from q is measured and the k closest points seen so far are recorded. Let us call o_k^A the current k -th closest point. The algorithm terminates when the distance from q to the region of current leaf is larger than $d(q, o_k^A)$, that is, the current region cannot contain objects whose distance from the query object is shorter than that of o_k^A . Since all remaining leaf regions are more distant from the current region, the k objects retrieved so far are the k nearest neighbors to q .

In contrast, the approximate nearest neighbor algorithm uses a stop condition to terminate the search prematurely. Specifically, the algorithm stops as soon as the distance to the current leaf region exceeds $d(q, o_k^A)/(1 + \epsilon)$. It is easy to show that under these circumstances, o_k^A is the $(1 + \epsilon)$ - k -th-approximate-nearest-neighbor. To clarify the behavior of the precise versus approximate nearest neighbor search algorithms, look at Figure 2.13, in which data objects are represented as black spots and a query $1NN(q)$ is posed. Each object is included in a rectangular region associated with a leaf node. Given a nearest neighbor query $1NN(q)$, every region is identified by a number assigned incrementally and based on the distance of the region to the query object q . Thus, the region containing q itself is assigned the value 1, while the region farthest away is labeled 10. The algorithm starts to search in Region 1 for a potential nearest object to q . The figure illustrates the situation in which Region 3 has been accessed in the current stage of execution, and the object o^A found as the current closest object. The circumference is indicated as having radius $d(q, o^A)$. The precise algorithm will continue accessing regions that overlap the circumference and stop only after accessing Region 10, which contains the actual nearest neighbor. The approximate algorithm, by contrast, accesses only those regions which overlap the dotted circumference whose radius is $d(q, o^A)/(1 + \epsilon)$. Therefore, it terminates after accessing Region 8, missing the actual nearest neighbor.

The priority search can be performed in $\mathcal{O}(m \log n)$, where m stands for the number of regions visited. The upper bound on m depends only on the dimensionality d , ϵ of the space and the number of nearest neighbors k , for any Minkowski metric, and is defined as $2k + \lceil 1 + 6d/\epsilon \rceil^d$. Provided that d and ϵ are fixed, the algorithm finds the $(1 + \epsilon)$ - k -approximate-nearest-neighbors in $\mathcal{O}(k \log n)$ time.

Note that upper bound on m is independent of the dataset size n . However, it depends exponentially on d , so this algorithm is feasible only in low-dimensional vector spaces.

5.3 Angle Property Technique

Other two vector-space-only techniques for reducing the number of nodes accessed during nearest neighbor searches are proposed in [Pramanik et al., 1999a, Pramanik et al., 1999b]. The chief novelty of these techniques lies in their exploitation of angles formed by objects contained in a ball region, the center of this region and a query object (see Figure 2.14). These techniques have been successfully applied to SS-trees [White and Jain, 1996]. However, they are generally applicable to any access method for vector spaces which partitions the data space, restricts groups of objects with ball regions, and organizes regions hierarchically.

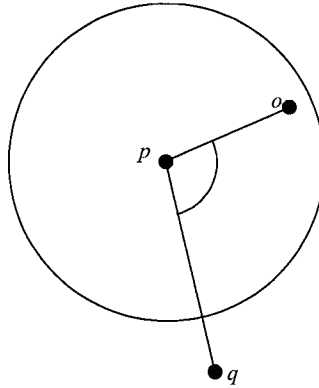


Figure 2.14. An angle between objects contained in a ball region and a query object q with respect to the center p of the ball region.

The heuristics employed in the search algorithm and proposed in [Pramanik et al., 1999a, Pramanik et al., 1999b] is justified by the following three properties of datasets in high-dimensional vector spaces:

- As dimensionality rises, the points in a ball region become almost equidistant from the region's center.
- With the increasing dimensionality, the radii of smaller child ball regions grow nearly as fast as the radius of the parent ball region, and thus their centers also tend to be close to each other.
- Given a query point and a set of points covered by a ball region, the angle between the query point and any point in the ball region will fall into an interval of angles around 90 degrees. As dimensionality grows, this interval will decrease.

Assuming regions are hierarchically structured, the algorithm uses an approximate pruning condition to decide whether a region should be accessed or not. In [Pramanik et al., 1999a], it is suggested that a region be inspected if at least one of the following conditions holds:

- The node corresponding to the region is an internal node.
- The center of the region's parent is contained in the ball region defined by the query object and the current candidate set of nearest neighbors.
- The region's center resides in the half of the parent's ball region closer to the query object, i.e., the angle between the center of the region and the query object with respect to the center of the parent's ball region is less than 90 degrees.

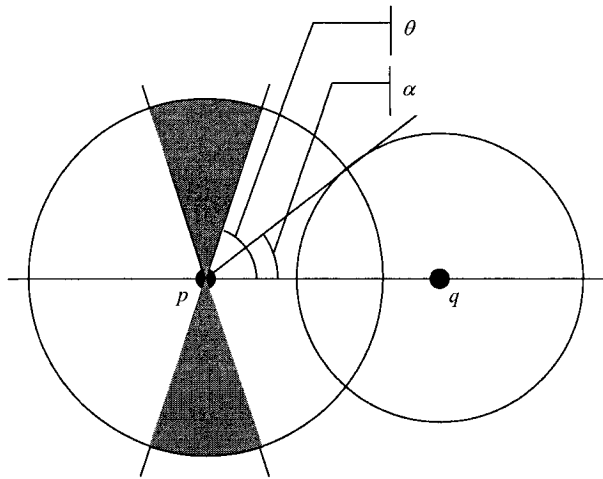


Figure 2.15. If the query region does not intersect promising portions of the data region, the region is discarded.

On reaching a leaf node, all objects of the leaf are examined and directly compared to the query object. If an object is closer to the query object than the current candidate for the k -th nearest neighbor, it is added to the response, superseding the k -th nearest neighbor.

This algorithm is, however, unable to trade performance with quality of results. In [Pramanik et al., 1999b], the algorithm is further improved by introducing a threshold angle θ to allow such a trade-off. Here is a brief sketch of how the improvement comes about:

According to the properties listed above, the area where qualifying objects are most likely to be found is close to the border of the ball region, forming an angle of about 90 degrees with the query object. Assume θ indicates the value of such an angle and the angle α is obtained by considering the query object q , the region's center p and the intersection of the query region and the region being examined (see Figure 2.15). If the angle α is greater than θ , the region is accessed, otherwise it is excluded – this is the situation depicted in the figure. Notice that if $\theta = 0$ all regions overlapping the query region are accessed and the query response-set is determined exactly.

5.4 Clustering for Indexing

The Clindex technique (Clustering for indexing) performs approximate similarity searches in high-dimensional vector spaces using an index structure supported by a clustering technique [Li et al., 2002]. The Clindex partitions the dataset into similar clusters, i.e., into clusters containing elements close to each

other in the space. Each cluster is represented by a separate file and all files are sequentially stored on a disk.

The Clindex technique uses a new algorithm for building clusters of objects. The algorithm starts by dividing each dimension of the d -dimensional vector space into 2^n segments, so every segment can be identified using an n -bit number. This process forms $(2^n)^d$ cells in the data space. The clustering algorithm aggregates these cells into clusters as follows: Each cell is associated with the number of objects it contains. The algorithm starts with the cells containing the largest number of objects and checks to see if they are adjacent to other clusters. If a cell is not adjacent to any cluster it is used as the seed for a new cluster. If a cell is adjacent to just one cluster, it is attached to that cluster. Finally, if the cell neighbors more than one cluster a special heuristics is applied to decide whether the clusters should be merged or to which cluster the cell belongs. This process is iterated until the remaining cells contain fewer objects than a specified threshold. Underfilled or empty cells are grouped in an *outlier* cluster and stored separately.

Once the clusters are obtained, an indexing structure is built for speeding access to them. The index is a simple encoding scheme which maps an object to a cell and a cell to its corresponding cluster. The associations between clusters and disk files are also kept.

Approximate similarity search is processed by first identifying the cluster to which the query object belongs. This is obtained by determining the cell which covers the query object and then identifying the corresponding cluster. If the query's cell is empty, a cluster cannot be obtained, so a cluster having the centroid closest to the query object is located. Once a cluster is identified, the file corresponding to it is sequentially searched, and objects qualifying for the query are returned. Of course, this search algorithm is approximate, because only one cluster is examined. In fact, it might happen that objects in non-selected clusters might also qualify for the query, so these objects are falsely dismissed by the algorithm.

5.5 Vector Quantization Index

Another approach that uses a clustering technique to organize data and process similarity queries approximately is the Vector Quantization Index (VQ-index) [Tuncel et al., 2002]. The VQ-index is based on reducing both the dataset and the size of data objects at the same time. The basic idea is to organize the dataset into subsets which are not necessarily disjoint, and then reduce the size of data by compression. The approximate search algorithm first identifies the subset to be searched. Next, it goes through its compressed content and qualifying objects are reported.

The dataset is grouped into subsets by exploiting a query history in the following way: Queries from the record of requests posed in the past are divided

into m clusters C_i ($i = 1, \dots, m$) using the k-means algorithm [MacQueen, 1967, Duda and Hart, 1973, Kaufman and Rousseeuw, 1990]. If the query history is too long, a sample is used instead. A subset S_i of the entire dataset corresponding to each cluster C_i is defined as follows:

$$S_i = \bigcup_{q \in C_i} kNN(q),$$

where $kNN(q)$ obviously represents the k objects of the dataset nearest to q . Each subset S_i contains elements of the dataset close to queries in the cluster C_i . Thus an element may belong to several different subsets. The overlap of subsets S_i versus index performance can be tuned by the choice of m and k .

The reduction in object size is obtained using the vector quantization technique [Gresho and Gray, 1992]. The objective of the vector quantization is to map an arbitrary vector from the original d -dimensional space into a reproduction vector. Reproduction vectors form a set of n representatives from the original space. The process of mapping can be decomposed into two modules – an encoder Enc and a decoder Dec . The encoder transforms the original space \mathbb{R}^d into a set $\{1, \dots, n\}$ of numbers, thus each vector $x \in \mathbb{R}^d$ gets assigned an integer $Enc(x) = c$ ($1 \leq c \leq n$). The decoder, by contrast, maps the set $\{1, \dots, n\}$ to the set of n reproduction vectors, so-called code-vectors, which in fact, approximate all possible vectors from \mathbb{R}^d , i.e., $Dec(c) = x$, $x \in \mathbb{R}^d$. For each subset S_i , a separate encoder Enc_i and decoder Dec_i is defined. S_i is compressed by representing it with the set $S_i^{enc} = \{Enc_i(x) | \forall x \in S_i\}$. The decoder function Dec_i is used to obtain the reproduction vectors corresponding to the elements in S_i^{enc} , i.e., the approximation of the original elements in S_i .

Here is an example: Having a fixed encoder Enc , several vectors can be mapped to a single number c . The best value for the corresponding code-vector is one minimizing its average distance to all vectors mapped to c . In this way, a suitable decoder function can be obtained. An approximate nearest neighbors query is processed by locating the cluster C_i nearest the query. Next, by applying the decoder function Dec_i on S_i^{enc} , the set S_i is reconstructed and sequentially searched for k nearest neighbors. A certain level of imprecision is present at both stages. In the first stage, it cannot be guaranteed that the selected subset S_i contains all objects which qualify for the given query. In the second stage, the vectors contained in the re-created set S_i might have distances to the query object significantly different from the distances of the original vectors. The approximation quality of the vector quantization technique depends on the number n of code-vectors. In practice, the number n is much smaller than the total number of vectors. Initially, the set of code-vectors is very small and huge collisions are solved by replacing the code-vector in question with two new vectors, improving the quality of the quantization. However, the experiments presented in [Tuncel et al., 2002] reveal the VQ-index is very competitive and

outperforms other techniques based on linear quantization by factor of ten to twenty, while retaining the same precision in the response.

5.6 Buoy Indexing

Another approach to approximate nearest neighbor search which is based on clustering is presented in [Volmer, 2002]. In this proposal, the dataset is partitioned into disjoint clusters bounded by ball regions. A cluster is represented by an element called a *buoy*. Clusters are gradually built by assigning objects to the cluster with the closest buoy. Radii of ball regions are defined as the distance between the buoy of a cluster and the furthest element in that cluster. This iterative optimization procedure attempts to find buoys of clusters so that radii of ball regions of these clusters are minimized. However, any other clustering algorithm that organizes the space into disjoint clusters bounded by ball regions can be used with the approximate search algorithm described below.

Imagine a dataset X with clusters $C_1, \dots, C_n \subset X$, where each C_i is bounded by a ball region $\mathcal{R}_i = (p_i, r_i)$ and p_i denotes the cluster's buoy. A precise k -nearest neighbors search algorithm accesses the clusters in the order determined by the distance between the query and the cluster's center, starting with the closest. The qualifying objects from every cluster visited are determined. This process is repeated until no better objects can be found in remaining clusters. The stop condition can be formalized as

$$\text{stop if } d(q, o_k) + r_j < d(q, p_j), \quad (2.4)$$

where q is the query object of a $kNN(q)$ query, o_k is the current k -th nearest neighbor, and p_j and r_j form a ball region $\mathcal{R}_j = (p_j, r_j)$ of a cluster to be accessed in the next step.

The proposed approximation strategy is to reduce the amount of data accessed by limiting the number of accessed clusters, i.e., modifying the stop condition. A parameter f ($0 < f \leq 1$) is introduced which specifies the clusters to be accessed. Specifically, the approximate kNN search algorithm stops when either Equation 2.4 holds, or the ratio of clusters accessed exceeds f . This technique guarantees $\lceil f \cdot n \rceil$ clusters will be accessed at a maximum, where n stands for the total number of clusters.

The results of experiments reported in [Volmer, 2002] imply query execution may be about four times faster than a linear scan, with about 95% recall ratio. The advantage of this method is that it is not limited to vector spaces only but can be applied to metric spaces as well.

5.7 Hierarchical Decomposition of Metric Spaces

There are other techniques beyond those mentioned for approximate similarity searching which have been especially designed for metric spaces. In what follows, we briefly introduce the basics of these techniques. However, in view

of their prominent role in the field of approximate similarity search, they are more extensively discussed in Chapter 4.

5.7.1 Relative Error Approximation

A technique employing a user-defined parameter as an upper bound on approximation error is presented in [Zezula et al., 1998a, Amato, 2002]. In particular, the parameter limits the relative error on distances from the query object to objects in the approximate result-set with respect to the precise results. The proposed technique can be used for both approximate nearest neighbor and range searches in generic metric spaces. Assuming a dataset organized in a tree structure, the approximate similarity search algorithm decides which nodes of the tree can be pruned even if they overlap with the query region. At the same time, it guarantees the relative error obtained on distances does not exceed the specified threshold. On a similar basis, nearest neighbor queries retrieve $(1+\epsilon)$ - k -approximate-nearest-neighbors. Details of this technique are given in Section 1 of Chapter 4.

5.7.2 Good Fraction Approximation

The technique presented in [Zezula et al., 1998a, Amato, 2002] retrieves k approximate nearest neighbors of a query object by returning k objects that statistically belong to the set of l ($l \geq k$) actual nearest neighbors of the query object. The value l is specified by the user as a fraction of the whole dataset. By using the overall distance distribution, the approximate similarity search algorithm stops when it determines that k objects currently retrieved belong to the specified fraction of objects nearest to the query. This method is discussed in detail in Section 2 of Chapter 4.

5.7.3 Small Chance Improvement Approximation

An approximate nearest neighbor search strategy proposed in [Zezula et al., 1998a] and later refined in [Amato, 2002] is based upon the pragmatic observation that similarity search algorithms for tree structures are defined as iterative processes where the result-set is improved in each iteration until no further improvement can be made. As for k -nearest neighbors queries, algorithms refine the response, which means that k objects retrieved in the current iteration will be nearer than those in the previous one. This can be explicitly measured by the distance between the current k -th object and the query object. Such a distance decreases rapidly in first iterations and it gradually slows down and remains almost stable for several iterations before the similarity search algorithm stops. The approximate similarity search algorithm exploits this behavior and stops the search algorithm when the reduction of distance to the current k -th object

slows down. A detailed description of this approach is given in Section 3 of Chapter 4.

5.7.4 Proximity-Based Approximation

A technique that uses a proximity measure to decide which tree nodes can be pruned even if their bounding regions overlap the query region is proposed in [Amato et al., 2003, Amato, 2002]. This has already been discussed in Section 10.2 of Chapter 1 from a theoretical point of view. When the proximity of a node's bounding region and the query region is small, the probability that qualifying objects will be found in their intersection is also small. A user-specified parameter is employed as a threshold to decide whether a node should be accessed or not. If the proximity value is below the specified threshold, the node is not promising from a search point of view, and thus not accessed. This method is defined for both nearest neighbor and range queries and is discussed in detail in Section 4 of Chapter 4.

5.7.5 PAC Nearest Neighbor Search

A technique called *Probably Approximately Correct* (PAC) nearest neighbor search in metric spaces is proposed in [Ciaccia and Patella, 2000b]. The approach searches for a $(1+\epsilon)$ -approximate-nearest-neighbor with a user-specified confidence interval. The proposed algorithm stops execution prematurely when the probability that the current approximate nearest neighbor is not the $(1+\epsilon)$ -approximate-nearest-neighbor falls below a user-defined threshold δ . Details of the approach are given in Section 5 of Chapter 4.

Similarity Search

The Metric Space Approach

Zezula, P.; Amato, G.; Dohnal, V.; Batko, M.

2006, XVII, 220 p., Hardcover

ISBN: 978-0-387-29146-8