A Compact Space Decomposition for Effective Metric Indexing [*]

Edgar Chávez[†] Gonzalo Navarro[‡]

Abstract

The *metric space model* abstracts many proximity search problems, from nearest-neighbor classifiers to textual and multimedia information retrieval. In this context, an *index* is a data structure that speeds up proximity queries. However, indexes lose their efficiency as the *intrinsic* data dimensionality increases. In this paper we present a simple index called *list of clusters* (LC), which is based on a compact partitioning of the data set. The LC is shown to require little space, to be suitable both for main and secondary memory implementations, and most importantly, to be very resistant to the intrinsic dimensionality of the data set. In this aspect our structure is unbeaten. We finish with a discussion of the role of unbalancing in metric space searching, and how it permits trading memory space for construction time.

1 Introduction

The problem of proximity searching has received much attention in recent times, due to an increasing interest in manipulating and retrieving the more and more common multimedia data. Multimedia data have to be classified, forecasted, filtered, organized, and so on. Their manipulation poses new challenges to classifiers and function approximators. The well-known k-nearest neighbor (knn) classifier is a favorite candidate for this task for being simple enough and well understood. One of the main obstacles, however, of using this classifier for massive data classification is its linear complexity to find a set of k neighbors for a given query.

The *metric space model* is gaining momentum as a paradigm to speed up proximity queries. *Metric databases* permit storing objects from a metric space and performing "proximity queries" over them efficiently, by building *metric indexes* that reduce the number of distance evaluations needed [11, 18]. By using a metric index, a knn classifier can afford massive classification tasks at reasonable time costs.

Proximity searching has applications in a vast number of fields, apart from classification tasks and multimedia data management. Some examples are image quantization and compression (where only some vectors can be represented and those that cannot must be coded as their closest representable point); text retrieval (where we look for words in a text database allowing a small number of errors), information retrieval (where we look for documents which are similar to a given query or document); computational biology (where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations); function prediction (where we want to search the most similar behavior of a function in the past so as to predict its probable future behavior); etc.

[*]This project has been partially supported by CYTED VII.19 RIBIDI Project (both authors), CONACyT grant 36911-A (first author) and Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (second author). A preliminary version of this paper appeared in [10].

[†]Escuela de Ciencias Físico-Matemáticas, Universidad Michoacana. Edificio "B", Ciudad Universitaria, Morelia, Mich. México 58000. `elchavez@fismat.umich.mx`.

[‡]Centro de Investigacin de la Web, Depto. de Ciencias de la Computación, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile. `gnavarro@dcc.uchile.cl`.

The most challenging problem in metric space searching is to deal with the so-called "high dimensional spaces" (see Section 2), where all the elements are more or less at the same distance from each other. Many metric spaces of interest in applications are high dimensional, and most indexes can do little on them.

In this paper we present the *list of clusters* (LC), a metric index based on compact partitions (see Section 3). We present analytical and experimental results to evaluate the index and understand its behavior. We show that our index is especially well suited to search high dimensional spaces, where it outperforms by far several prominent alternative metric indexes. We also show how our index could work efficiently in secondary memory and how could it be updated upon insertions and deletions in the database.

The key to the success of the LC in high dimensions is that it trades construction time for query time. Alternative structures attempt to cope with high dimensions by trading memory space for query time. Memory space is a much higher price than construction time, which is paid only once or sparsely. In practice, memory space puts a tighter limit than those derived from construction time.

We finish with a discussion of the role of unbalancing in metric space searching. In exact searching, balanced data structures are always best. We show how this ceases to be true in metric spaces as the intrinsic dimension grows, and how our LC can be seen as an extremely unbalanced tree, thus explaining its better fitting to high dimensional spaces when compared, in particular, against the many existing indexes based on balanced trees. Moreover, unbalancing gives the conceptual framework to understand how construction time can be used instead of memory space to face high dimensions.

2 Basic Concepts

2.1 Metric Spaces

Proximity queries extend exact searching in the sense that they retrieve objects from a database that are *close* to a given query object. The query object is not necessarily a database element. The concept can be formalized using the metric space model, where a distance function $d(x, y)$ is defined over pairs of elements in a set $\mathbb{X}$. The distance function has *metric* properties, that is, it satisfies $d(x, y) \geq 0$ (positiveness), $d(x, y) = d(y, x)$ (symmetry), $d(x, y) = 0$ iff $x = y$ (strict positiveness), and $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).

The database is a set $\mathbb{U} \subseteq \mathbb{X}$, and we define the query element as q, an arbitrary element of $\mathbb{X}$. A proximity query involves additional information, besides q, and can be of two basic types: *Metric Range queries*, $(q, r)_d = \{u \in \mathbb{U} : d(q, u) \leq r\}$; and *Nearest Neighbor queries*, $nn_k(q)_d = \{u_i \in \mathbb{U} : \forall v \in \mathbb{U},\ d(q, u_i) \leq d(q, v) \text{ and } |\{u_i\}| = k\}$.

Given a database of $|\mathbb{U}| = n$ objects, all those queries can be trivially answered by performing n distance evaluations. Since the distance function is usually expensive to compute, the goal is to structure the database (that is, to build an *index*) so that we perform few distance evaluations at query time. Essentially, the index maintains information on distances between database elements that permits, using the metric properties, to prove later that some database elements are far enough from the query, without actually measuring each distance.

The idea is that, given the database $\mathbb{U}$, we first build an index on it so that we can answer many queries later. The database can be static (never changing) or dynamic (incorporating or losing elements along time). In the latter case, the index must support those database updates, and it is expected that the construction and update cost is amortized by sufficient queries between updates. Many databases are essentially static and the index construction cost is less relevant. On the other hand, the memory space required by the index is relevant in all cases, as it has to be maintained all the time.

We focus on range queries in this paper, as the others can be systematically built over these in an optimal way [17, 11, 18]. The set of points of $\mathbb{X}$ that are at distance at most r to q is called the "query ball", so $(q, r)_d$ is the intersection of the query ball and $\mathbb{U}$. We remark that the radius r (as well as the value k in nearest neighbor queries) is given as a part of the query and depends on the application. For example, it may be estimated so that a certain number of elements or fraction of the database is retrieved, or such that farther elements are known to be irrelevant.

The indexing techniques discard elements using the metric properties. There are applications, however, where some metric properties do not hold. If the distance is not strictly positive, the space is called a *pseudo-metric space*. Most techniques for metric spaces work for pseudo-metric spaces as well, by simply identifying all the objects at distance zero as a single object. In some cases we may have a *quasi-metric*, where distance is not symmetric. There exist techniques to derive a new, symmetric, distance function from an asymmetric one, such as $d'(x, y) = d(x, y) + d(y, x)$. However, specific knowledge of the domain is necessary to properly adapt a search radius to the new space. Finally, sometimes the triangle inequality holds only in relaxed form, such as $d(x, y) \leq \alpha d(x, z) + \beta d(z, y) + \delta$. After some scaling, those spaces can be searched using the same algorithms designed for metric spaces. Yet, if the triangle inequality does not hold at all, and in absence of further properties of the space, there are no known methods to avoid a linear database scan to solve a query.

On the other hand, there are some well-studied particular cases of metric space searching. The best known is the k-dimensional coordinate space $\mathbb{R}^k$, especially using the Euclidean distance. There are effective methods for this case, such as kd-trees, R-trees, X-trees and many others [15]. However, for more than roughly 20 dimensions those structures cease to work well [3]. We focus in this paper in general metric spaces, although the solutions are well suited also for k-dimensional spaces. An immediate advantage of regarding a k-dimensional space as a metric space is that, if the data embedded in the former space have lower dimension, then the real, intrinsic, dimension of the data shows up in the latter space, independently of k.

2.2 Dimensionality

It is interesting to notice that the concept of "dimensionality" can be translated to metric spaces as well: The typical feature in high dimensional spaces is that the probability distribution of distances among elements has a very concentrated histogram (with larger mean as the dimension grows), hampering the work of any proximity search algorithm [3, 5, 7]. In the extreme case we have a space where $d(x, x) = 0$ and $\forall y \neq x,\ d(x, y) = 1$, where it is impossible to avoid a single distance evaluation at search time. We say that a general metric space is high dimensional when its histogram of distances is concentrated. We use in this paper a quantitative measure of the intrinsic dimensionality proposed in [11]:

Definition: The *intrinsic dimensionality* of a metric space is defined as $\rho = \frac{\mu^2}{2\sigma^2}$, where μ and σ^2 are the mean and variance of its histogram of distances.

Under this definition, a random vector space with k coordinates has intrinsic dimension $\Theta(k)$, with a constant close to 1, so the definition extends naturally that of vector spaces. Note this measure gives only a rough idea of *how difficult* it is to search a given dataset from a metric space.

Many metric space of interest are indeed high-dimensional. Searching them is so difficult that the term "curse of dimensionality" has been coined to refer to this fact. Finding efficient solutions to high-dimensional metric space searching is currently the most important open problem in metric space searching.

3 Related Work

According to [11] there are two main approaches for metric index design (see also [18] for a different characterization):

- *Pivot-based algorithms,* which select a number of "pivots" from the database and classify all the other elements according to their distances to them. The distances between elements and pivots and between the query q and the pivots are used together with the triangle inequality to filter out elements of the database without actually measuring their distance to q. These algorithms generally improve as more pivots are added, although the space requirements of the indexes increase as well.

- *Compact partitioning algorithms,* which divide the set into spatial zones as compact as possible, and are able to discard complete zones by performing few distance evaluations (e.g., between the query q and a centroid of the zone). The partition into zones can be hierarchical, but the indexes use a fixed amount of memory and do not improve by having more space.

As shown in [11], compact partitioning algorithms deal better with high dimensional metric spaces. Despite that pivot-based algorithms can improve by using more memory, they need more and more memory to beat compact partitioning algorithms as the dimension grows. For intrinsic dimension around 20 they already need impractical amounts of extra space. Therefore, compact partitioning algorithms seem a promising alternative to index high dimensional metric spaces.

A way to see the difference between pivot-based and compact partitioning algorithms is that the former define "rings" (elements at the same distance) around pivots, and the intersections of rings define the zones in which the space is partitioned. To maintain those zones spatially compact as the dimension grows, more and more rings have to be intersected, that is, more pivots are necessary. The alternative technique uses compact partitions by construction, and takes no advantage from additional memory. It is easier to prove that a compact region is far enough from the query than to do the same for a sparse region.

3.1 Pivot-based Algorithms

Burkhard-Keller Trees (bk-trees) [6] are designed for discrete distance functions: they select a pivot element p as the root of the tree, and put at child i the elements which are at distance i to the pivot. Each subtree is recursively built with the same technique until there are b elements or less, in which case the elements are simply stored in a "bucket" at the tree leaf. A range query q with tolerance radius r is searched by measuring $d(p, q)$, reporting p if appropriate, and entering only into subtrees numbered $d(p, q) - r$ to $d(p, q) + r$. The rest need not be considered because of the the triangle inequality. The buckets reached are exhaustively compared against q.

Fixed Queries Trees (fq-trees) [2] are an evolution where the same pivot is used for all the nodes of the same level of the tree. In this case the pivot does not need to belong to the subtree. Many comparisons are saved in the backtracking process because only one different pivot per level exists. However, the tree is taller. A variant called Fixed Height fq-tree (fhq-tree) is also proposed where all the leaves are at the same depth h, regardless of the bucket size.

Vantage Point Trees (vp-trees) [25, 27] are designed for continuous distance functions. The root has two equal-size subtrees that divide the elements in closer to and farther from the root. This can be extended to m-ary trees (mvp-trees) [5, 4].

Finally, algorithms like AESA [26], LAESA [21, 20] and its variants [23, 8] and Fixed Queries Arrays (fq-arrays [9]) are based in a common idea: k pivots are selected and each object is mapped to k coordinates which are its distances to the pivots. Later, the query q is also mapped and if it differs from an object in more than r along some coordinate then the element is filtered out by the triangle inequality. That is, if for some pivot p_i and some element v of the set it holds $|d(q, p_i) - d(v, p_i)| > r$, then we know that $d(q, v) > r$ without need to evaluate $d(v, q)$. The elements that cannot be filtered out using this rule are directly compared.

An interesting feature of most of these algorithms is that they can reduce the number of distance evaluations by increasing the number of pivots. Define $D_k(x, y) = \max_{1 \le j \le k} |d(x, p_j) - d(y, p_j)|$. Using the pivots $p_1, ..., p_k$ is equivalent to discarding elements u such that $D_k(q, u) > r$. As more pivots are added we need to perform more distance evaluations (exactly k) to compute $D_k(q, *)$, but on the other hand $D_k(q, *)$ increases its value and hence it has a higher chance of filtering out more elements. It follows that there exists an optimum k, This optimum, however, cannot be normally reached because it is too high in terms of space requirements: kn distances have to be precomputed and stored in order to use k pivots. Hence, in general these methods use as many pivots as they can, and they are normally well below their optimum.

3.2 Compact Partitioning Algorithms

Generalized Hyperplane Trees (gh-trees) [25] use two "centers" for each tree node and divide the space according to which of the two centers is closer to each object. At search time the query enters into the subtrees whose zone of influence has a nonempty intersection with the query ball.

Bisector Trees [19, 24] are similar but the zones are not defined according to which is the closest center but using the concept of "covering radius". The *covering radius* of a zone is the minimum radius of a sphere that is necessary to contain all the points in the zone, and the elements are inserted in the subtrees trying to minimize covering radii. This is generalized to Voronoi Trees (v-trees) in [13] to reduce more the covering radii.

Gh-trees are generalized to an m-ary partition in the Geometric Near-neighbor Access Tree (gna-tree) [5], which makes a Voronoi-like partition of the space [1] among the m centers at each node of the tree. However, the gna-tree uses also the covering radius criterion to prune the search even more.

The M-tree [12] also takes m elements and divides the space among its zones of influence, but it uses only the covering radius information to classify and search the elements. The M-tree is able of dynamic insertion and deletion of points and is optimized for secondary memory.

The D-Index [14] data structure is also designed for secondary memory. It partitions the data using the concept of "separable partitions", where a minimum distance is guaranteed among objects of different partitions. They combine their idea with pivots to get the best of each approach.

Spatial Approximation Trees (sa-trees) [22] are based on approaching the query spatially: the search starts at the root of the tree and moves to neighbors that are closer to the query. The ideal data structure to obtain this is a Voronoi graph, which in the paper is proven impossible to build on a general metric space. Therefore the sa-tree is a simplification which forces some backtracking in the tree. Dynamic and secondary memory improvements have been recently added.

4 A New Compact Partitioning Technique

We propose now a simple but effective technique to index a metric space. We start by choosing a "center" $c \in \mathbb{U}$ and a radius r_c whose value is discussed later (do not confuse with the search radius r, whose possible values are unknown at indexing time). We define the *center ball* of (c, r_c) (or just c if no ambiguity is possible) as the subset of elements of $\mathbb{X}$ which are at distance at most r_c from c. Now we define

$$I_{\mathbb{U},c,r_c} \;=\; \{u \in \mathbb{U} - \{c\},\ d(c,u) \leq r_c\}$$

as the bucket of "internal" elements, which lie inside center ball of c, and

$$E_{\mathbb{U},c,r_c} \;=\; \{u \in \mathbb{U},\ d(c,u) > r_c\}$$

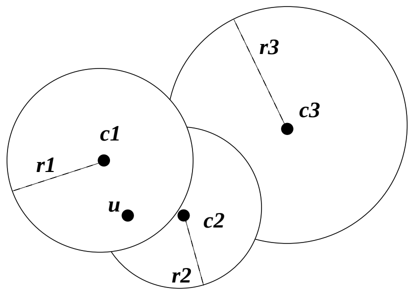
as the rest of the elements (the "external" ones). Now the process is repeated recursively inside E. The construction procedure returns a list of triples (c_i, r_i, I_i) (center,radius,bucket) and it is shown in Figure 1.

The data structure that is built looks rather symmetric, but it is not. The first center chosen has preference over subsequent centers in case of overlapping balls, as illustrated on the right of Figure 1. All the elements that lie inside the ball of the first center (c_1 in the figure) are stored in its I bucket, despite that they may also lie inside the I buckets of subsequent centers (c_2 and c_3 in the figure). The figure also shows how the data structure can be seen as a list.

The search algorithm is depicted in Figure 2. The idea is that if the first center chosen is c and its radius is r_c, then the search for a query (q, r) starts by measuring $d(c, q)$ and adding c to the set of results if appropriate. Then, we search exhaustively the bucket I only if the query ball has some intersection with the center ball of c. After considering the first ball, we go on with E. However, given the asymmetry of the data structure, we can also prune the search in the other way: If the query ball is totally contained in the center ball of c, then we do not consider E, as by construction we know that all the elements that are inside the query ball have been inserted into I.

This is an essential feature absent in other compact partitioning algorithms, where the search needs to enter into all the partitions which are intersected by the query ball. With our data structure, the consideration of relevant partitions can be preempted as soon as the query ball is totally contained in a partition. Figure 2 (right) illustrates.

```
Build ($\mathbb{U}$)
    if $\mathbb{U} = \emptyset$ then return an empty list
    Select $c \in \mathbb{U}$
    Select a radius $r_c$
    $I \longleftarrow \{u \in \mathbb{U} - \{c\},\ d(c,u) \leq r_c\}$
    $E \longleftarrow \mathbb{U} - I$
    return ($c$,$r_c$,$I$):Build($E$)
```



$$\begin{tikzcd}
(c1,r1) \arrow[r, "E"] \arrow[d] & (c2,r2) \arrow[r, "E"] \arrow[d] & (c3,r3) \arrow[r, "E"] \arrow[d] & {} \\
I & I & I &
\end{tikzcd}$$

Figure 1: The construction algorithm. The operator ":" is the list constructor. It is not hard to remove the tail recursion to make it iterative. On the right, the influence zones of three centers taken in this order: c_1, c_2, c_3. We also show a list arrangement for the data structure.

```
Search ($L$,$q$,$r$)
    if $L$ is empty then return
    Let $L = (c, r_c, I) : E$
    Compute $d(c,q)$
    if $d(c,q) \leq r$ then add $c$ to the list of results
    if $d(c,q) \leq r_c + r$ then search $I$ exhaustively
    if $d(c,q) > r_c - r$ then Search ($E$,$q$,$r$)
```

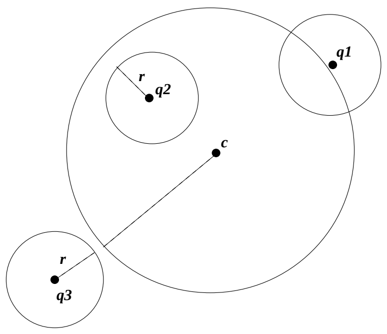


Figure 2: The search algorithm. It is not hard to remove the tail recursion to make it iterative. On the right we illustrate three cases of query ball versus center ball. For q_1 we need to consider the current bucket and the rest of centers. For q_2 we can prune the search inside the rest of the partitions. For q_3 we can avoid considering the current bucket.

5 Building and Updating the Data Structure

5.1 Center and Radius Selection

We have not discussed until now how centers c and radii r_c are chosen when the structure is built. This affects only performance, not correctness. We show several alternatives here and test them in Section 6.

Center selection. We can apply different heuristics to select the i-th center.

($p1$) At random.

($p2$) The element closest to c_{i-1} in the remaining set.

($p3$) The element farthest from c_{i-1} in the remaining set.

($p4$) The element minimizing the sum of distances to previous centers.

($p5$) The element maximizing the sum of distances to previous centers.

The first alternative is the simplest but not necessarily the best one. The second one aims at building a bucket ordering that moves slowly across the metric space. The third one aims at minimizing the overlap between partitions. ($p4$) and ($p5$) are more global versions of ($p2$) and ($p3$), respectively. Moreover, ($p2$) and ($p4$) aim at finding a next center close to the current one, as in sa-trees, while ($p3$) and ($p5$) try that the volumes of different partitions do not overlap, as gna-trees.

Radius selection. Two simple alternatives are:

Partitions of Fixed Radius: The simplest alternative seems to be selecting a fixed radius r^* for all the balls in the list. This implies that, as we advance in the list, they get emptier.

Partitions of Fixed Size: Another choice is to try to have a fixed number m^* of elements inside each center ball, and to define the radii accordingly. This also fixes the length of the list to $\lceil n/(m^*+1)\rceil$. As we advance in the list, the balls will be spatially larger.

5.2 Construction

Our data structure can be built by brute force in $O(n^2/p^*)$ time for fixed radius partitions and $O(n^2/m^*)$ time for fixed size partitions, where p^* is the expected bucket size. Although in principle this cost is independent of the dimension, it gets higher in practice as the dimension grows because m^* or p^* have to be reduced to ensure low query times on higher dimensions.

This cost, however, can be reduced by noting that I is defined as the result of a range query (c_i, r^*) for fixed radius partitions and of a nearest neighbor $nn_{m^*}(c_i)$ query for fixed size partitions. Therefore, another (cheaper to build) data structure built on the metric space could be used as an auxiliary data structure to build ours. This matches especially well with the center selection heuristics ($p1$) and ($p2$), while the others may need extra work. It is also worthwhile to note that this auxiliary data structure should be able of efficient deletion of the elements that are inserted into each new partition, in order to answer queries on the remaining set.

5.3 Updating

Let us consider the process of inserting a new element into the fixed radius data structure. The insertion can be done by traversing the list of partitions until the element falls inside some center ball, or otherwise creating a new partition for it at the end of the list.

Deletion can be trivially done except if a center is deleted, in which case a first choice is to keep it anyway as a fake element. A safer choice is to remove the whole bucket from the list and reinsert all the elements (note that the insertion of those elements can be done just in the tail of the list, as we already know that they do not lie inside any previous center ball).

If r^* has been correctly computed in the beginning, it should not change as we insert more elements. However, a massive insertion of elements may affect the optimality of the r^* value chosen. In those cases a periodic rebuild of the whole data structure may be benefical for the performance.

These update operations are a bit more complex if we have a fixed bucket size. When inserting an element, as soon as we find its appropriate ball i, the bucket will overflow. Hence we take the element of the bucket which is farthest from the center c_i, remove it from the bucket (modifying r_i accordingly), and continue the insertion process in the tail of the list with the new element. Hence we are guaranteed to traverse the whole list of centers for every insertion. Deletion presents a more difficult problem, since the bucket underflows and we have to find the next nearest neighbor of c_i in the rest of the elements. This can be done using the same data structure, but it is costly anyway. Two choices are lazy deletion (i.e., leave the hole hoping that a new insertion will fit the place) and setting a range of values for m^* instead of a fixed value. Deletion of a center can be handled as for the fixed radius data structure.

5.4 Secondary Memory

Our data structure has the advantage of a rather predictable access pattern. The partition centers are compared always in the same order. Sometimes we need to retrieve a whole bucket, sometimes not. Finally, we can stop the search at any moment in the list of centers.

A simple linear arrangement of the centers yields an efficient disk layout for this search algorithm, with minimal seek time. The buckets should be similarly arranged in a separate list. Fixed size buckets make this extremely simple, while fixed radius partitions need an expansion mechanism to accommodate their varying size. There are well known mechanisms of that type, and the histogram can be used as a tool to upper bound overflow probabilities.

6 Experimental Results

We present now experiments that compare different choices of our data structure, as well as alternative structures.

Our metric space is the unitary real cube in k dimensions ($[0,1)^k$) under the Euclidean distance. We generate a fixed number n of random points and search random queries q with a radius r such that 0.01% to 0.1% of the set of points is retrieved. We show the results as a function of the dimension k of the space. Despite that this is a restricted case of vector space, we can in this case effectively control the dimension, which is difficult to do in real-world examples. We make the experiments with $n = 100,000$ elements.

Our first experiment tries to determine the best choice among $(p1) - (p5)$. Figure 3 shows the results using two different choices for m^* (12 and 100) and r^* (1/4 and 1/8 of the maximum distance).

For fixed bucket size $(p3)$ and $(p5)$ are better choices, which favors heuristics that try to minimize the intersection among partitions [27, 5]. The difference among $(p3)$ and $(p5)$ is not statistically significant when using a large bucket size. With a smaller bucket size (12) the $(p5)$ heuristic is clearly better and therefore we use $(p5)$ from now on, as it is a more elaborated version of $(p3)$ that should work in more complex scenarios (such as clustered data).

For balls of fixed radius the results are quite different. For a large radius r^* the difference between the five heuristics is not statistically significant. For a smaller radius r^* the best heuristic is $(p4)$. Observe also that the results for the fixed radius alternative are quite promising, though the tuning of the algorithm is far more complicated than for the fixed bucket strategy. Figure 3 (bottom) shows that a relatively small change in the fixed radius yields a very large difference in performance.

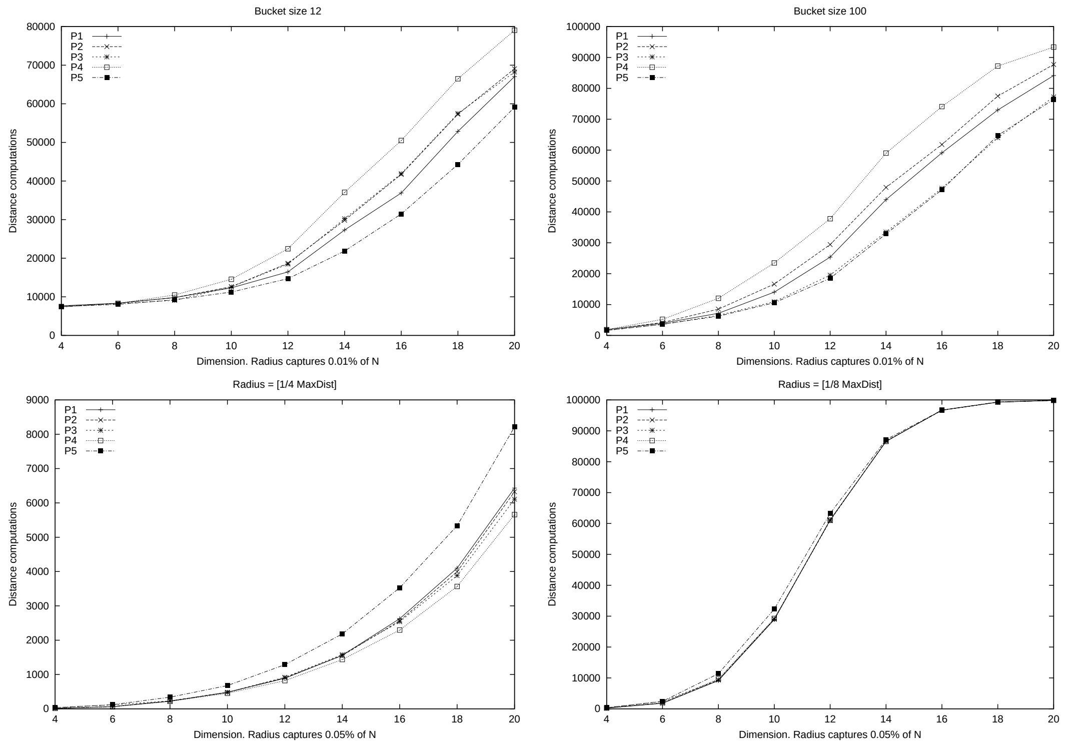


Figure 3: Number of distance evaluations for center selection techniques $(p1)$ to $(p5)$, as the dimension grows. On the top row, fixed bucket sizes $m^* = 12$ and $m^* = 100$, capturing 0.01% of the database. On the bottom row, fixed radii $r^* = 1/4$ and $r^* = 1/8$ of the maximum distance, capturing 0.05% of the database.

Let us from now on focus in partitions of fixed size, and consider the optimal m^* parameter.

Figure 4 shows that there exist optimal bucket sizes. These depend on the search radius and the intrinsic dimension of the dataset. In the left of the figure, we have used a smaller search radius, showing how the optimum shifts to the right as the search radius increases.

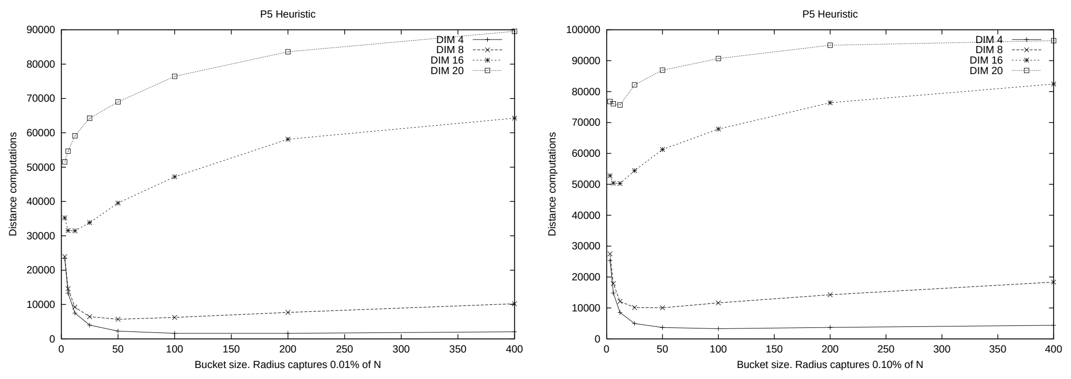


Figure 4: Selection of the optimal bucket sizes, for varying dimension. On the left we used a small search radius (0.01% of the dataset), and a larger one on the right (0.1% of the dataset).

We now compare our data structure against some existing techniques. Observe in Figure 5 that three pivot-based algorithms (fhq-trees, fq-arrays and LAESA) have needed at least 64 times more memory than the other compact partitioning algorithms (gna-trees and sa-trees) in order to beat them in medium dimension. This experimental evidence favors the use of compact partitioning algorithms instead of pivot-based ones in high dimensions. In particular, we had not enough memory in our machine to give them enough pivots so that they beat our LC in 20 dimensions, and this would be even more difficult for them in higher dimensions or larger search radii. Our index, instead, does not need more memory to cope with higher dimensions. Moreover, its complexity grows much slower as the dimension grows. In particular, the combination we have chosen is by far the best in 20 dimensions, even if we allow using 64 times more memory to competing pivot based algorithms.

Finally, in Figure 6 we contrast the construction cost versus the space usage. We note that LC uses a constant amount of space per bucket (all the rest is solved by a suitable arrangement of elements), while the best representative of pivots uses an increasing number of pivots in each dimension to keep up with the LC. The efficiency of LC in space usage (right) is paid in the construction cost (left).

7 Discussion

7.1 Balancing versus Unbalancing

Note that it is possible to see our list of clusters as a particular case of vp-trees, by considering I and E as the left and right subtrees of the root c. There is, however, a fundamental difference. While vp-trees and many related data structures try to build balanced trees, our structure is extremely unbalanced, as I is much smaller than E. Moreover, our I bucket does not have any internal structure.

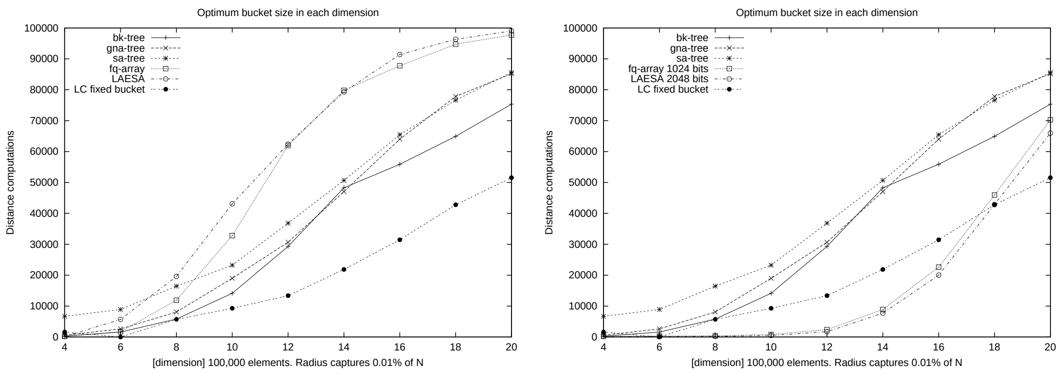


Figure 5: Comparison with existing approaches. On the left all the indexes use (about) the same memory. On the right the pivot-based algorithms are allowed to use more memory.

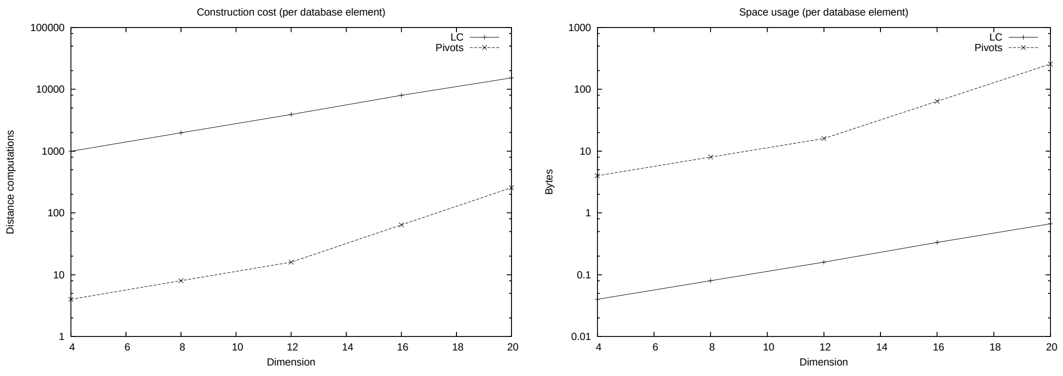


Figure 6: Pivots and LC compared for construction cost (left) and space usage (right). LC uses the optimal bucket size for each dimension. The number of pivots used in each dimension are those necessary to reach the same LC query cost.

One of the first lessons learned in any elementary book of algorithms is that balanced data structures (trees in general) provide the best performance. Indeed, as a tree becomes more unbalanced it becomes more similar to a linked list, and the search cost raises from $O(\log n)$ to $O(n)$. Different techniques to have balanced data structures are proposed, such as 2-3 trees, AVL trees and red-black trees for the worst case; splay trees for the amortized worst case; and randomized trees and skip lists for the average case [16].

However, all the concept of balancing is based on the implicit assumption of *exact searching*: We have a search query and want to find its exact replica in the tree. Hence, we enter only one branch of the tree, and therefore a balanced tree minimizes the cost. More sophisticated queries such as range searching are still based on the assumption that there exists a total linear order on the keys. Hence, these queries are reduced to a couple of exact searches to find the extremes of the range of interest.

None of these assumptions is valid in proximity searching. The only tool to organize a data structure on metric spaces is the distances among elements. Many proposals still manage to design tree data structures, where a total linear order is imposed by sorting the elements according to their distances to the root. Probably influenced by a strong algorithmic background, most authors try as well to obtain a balanced data structure by splitting the range of distances so that the subtrees have the same size[1].

The real problem with this approach appears when one considers the type of search carried out on these balanced trees. As explained, the search is not exact, but it has a tolerance radius r which is fixed at query time and is insensitive to the slices assigned by the tree. Low dimensional metric spaces have a histogram of distances which is more uniform than those of high dimensional spaces. In low dimensional spaces, therefore, the query is compared against the root and a range of the histogram is selected (see Figure 7). This range contains a reasonably small fraction of the distribution and therefore the problem is reduced well along the iterations. Moreover, since the histogram is not concentrated, a partition where the subtrees have the same number of elements yields slices of approximately the same width, and therefore the search enters into a reasonable number of subtrees.

Consider now a high dimensional space (right hand of Figure 7). All the histogram is concentrated in a small range, where the query also lies with high probability when compared to the root of the tree. Hence a large proportion of the elements will now be selected by the query range. This is the basic reason that makes searching in high dimensional spaces so difficult.

However, balancing the trees adds an extra inefficiency to this. As the histogram becomes more concentrated, the slices to partition the elements in equal sized groups become thinner (Figure 8). Since the search radius stays the same, it will intersect more slices and the search will need to enter more subtrees. This shows why the search model for proximity queries makes balanced trees a poor choice for high dimensional metric spaces.

A tree where the slices have fixed width avoids this last problem. Since the width is independent on the dimension of the space, the search will not enter more subtrees of a node as the dimension grows (Figure 8). However, a new consequence shows up when fixed slices are used: The subtrees corresponding to the slices containing the core of the distribution will have much more elements

[1]This approach is sometimes of value if we are also interested in disk access costs, as an unbalanced tree might access more disk pages.

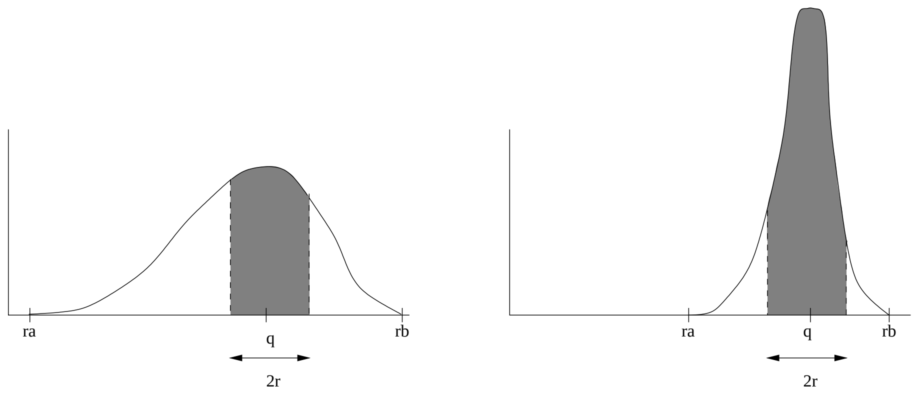


Figure 7: A low-dimensional (left) and high-dimensional (right) histogram of distances, showing that on high dimensions virtually all the elements become candidates for the exhaustive evaluation.

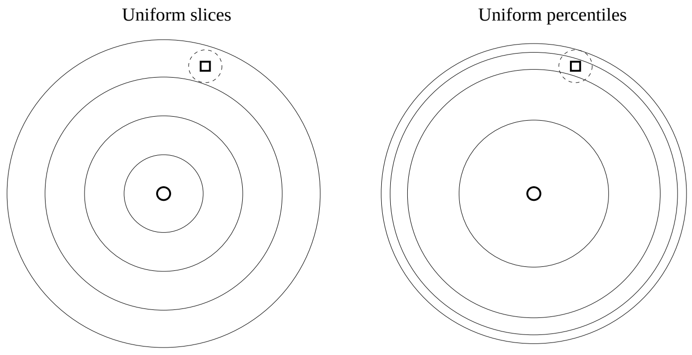


Figure 8: The same query intersects more rings when the slices have the same number of elements (outer rings are denser).

than the rest, and therefore the tree will be more and more unbalanced as the dimension grows.

As the tree becomes more unbalanced and hence taller as the dimension grows, the leaves of the tree will know their (approximate) distance to more pivots: from $O(\log n)$ in a balanced tree to $O(n)$ in a very unbalanced tree. Also a random query will be compared against more pivots as it traverses the tree. This effect is very similar to having a large number k of pivots in plain pivot-based algorithms. Unlike those, however, this unbalanced tree takes always linear space. The main problem is its construction cost, which moves from $O(n \log n)$ to $O(n^2)$ as the tree loses balance.

To summarize, unbalancing permits, in essence, to have a large number of pivots without incrementing the space cost (the price is paid in construction cost). Hence we can reach the optimum number of pivots, which grows with the dimension. Needless to say, this shows that the division into pivoting and compact partitioning schemes can be blurred and one can consider them as quantitative variants of a single general concept.

7.2 Optimum Unbalance

Let us still see our data structure as an unbalanced binary tree. We focus now on how to split the range of distances so as to optimize the average search time. Our goal is to show analytically why it might be benefical to unbalance the tree in high dimensions.

Let us assume that we choose a tree root p at random and cut the interval of distances to the tree root at value s. Let us call $F(x)$ the cumulative distribution of the histogram of distances, that is, $F(x) = Prob(d(a,b) \leq x)$. The fraction of elements that are stored inside the left subtree is $P_0 = F(s)$, while $P_1 = 1 - F(s)$ go to the right subtree.

At search time, the probability of entering the left subtree with a query of radius r is that of $[d(q,p) - r, d(q,p) + r]$ intersecting the interval $[0,s]$, that is, $d(q,p) - r \leq s$, or $d(q,p) \leq s + r$. Let us call Q_0 this probability, so $Q_0 = F(s+r) \geq P_0$. Similarly, $Q_1 = 1 - F(s-r) \geq P_1$ for the right subtree.

Let us assume that the subtrees conserve the same probabilistic structure (this is a simplification, but serves to illustrate our point). In this case the average cost of a query with radius r in a tree of n elements is $T(1) = 1$ and

$$T(n) \quad = \quad 1 + Q_0 T(P_0 n) + Q_1 T(P_1 n)$$

whose rationale is as follows. The first comparison is for the root of the tree. Then, each subtree is entered with probability Q_i and inside it we have recursively the same problem on $P_i n$ elements. Despite that entering into different subtrees are not independent events, the average is still the same as if they were.

It is easy to prove by substitution that $T(n) = (Sn^\alpha - 1)/(S-1) = O(n^\alpha)$, where $S = Q_0 + Q_1 > 1$ and α is the solution to the trascendental equation

$$Q_0 P_0^\alpha + Q_1 P_1^\alpha \quad = \quad 1$$

where we see that $0 \leq \alpha \leq 1$, as this summation goes from $Q_0 + Q_1 \geq 1$ to $Q_0 P_0 + Q_1 P_1 \leq 1$ as α moves from zero to one. Note that if $r = 0$ then the solution is different, $O(\log n)$, as it is exact searching. Proximity searching has an entirely different analytical structure.

Our optimization problem is then to choose s values so as to minimize the search complexity, that is, to satisfy $\sum Q_i P_i^\alpha = 1$ for the least possible α. Note that s determines the P_i values and, together with r, the Q_i values. So the optimum choice will depend on the r of interest.

Consider now the case of low dimension. The histogram is relatively flat, and therefore the differences $Q_i - P_i$ are relatively independent of s (they do depend on r). In this case the optimum solution is at $s = 1/2$, that is, a balanced structure. On the other hand, when the histogram is concentrated around its mean, $Q_i - P_i$ is much larger around the mean (close to $s = 1/2$). As the Q_i values grow, α goes to 1 and the complexity increases. In order to reduce this effect, it is preferable to shift s to an area of the histogram where the mass of the histogram in the area $[s-r, s+r]$ is small. This occurs at the tails of the distribution, hence advising an unbalanced partition. A larger radius only increases this effect, as the area $[s-r, s+r]$ is enlarged.

8 Conclusions

We have presented a new compact partitioning index for proximity searching, which is experimentally shown to be much more efficient than others in high dimensions. It is also simple to program and use, needs little space, and is amenable of a secondary memory implementation.

Unlike existing approaches, which face high dimensions by increasing the memory space of the index, ours increases instead construction time. This is a much more affordable cost in practice and it permits handling higher dimensions efficiently. Given the good properties of the data structure, it is worth to explore parallel and distributed algorithms for the index construction.

We have shown how our structure can be seen as the result of unbalancing some classical tree approaches for metric space searching, which work well only on low dimensions. Moreover, we have shown how unbalancing is a key feature to deal with high dimensional spaces, and how it is the key to use more construction time instead of more memory space.

Future work involves improving the construction procedure, possibly by using auxiliary data structures to build the I buckets. We also plan to pursue in the problem of obtaining a dynamic data structure that supports insertion and removal of elements.

The *List of Clusters* give a sui-generis hierarchical view of the data. It could be interesting to further investigate the relationship between classical data cluster detection algorithms and the clusters obtained with our algorithm.

Finally, it would be interesting to devise I/O efficient variants that are able to compete with M-trees and D-indexes in secondary memory. We have sketched possible alternatives but a deeper study is necessary.

We wish to thank the anonymous referees who helped us to improve the presentation.

References

[1] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.

[2] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. CPM'94*, LNCS 807, pages 198–212, 1994.

[3] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. SIGMOD'97*, pages 357–368, 1997. Sigmod Record 26(2).

[5] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584, 1995.

[6] W. Burkhard and R. Keller. Some approaches to best-match file searching. *CACM*, 16(4):230–236, 1973.

[7] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In *Proc. WSP'97*, pages 21–36. Carleton University Press, 1997.

[8] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

[9] E. Chávez, J.L. Marroquin, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

[10] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 75–86. IEEE CS Press, 2000.

[11] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[12] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[13] F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[14] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21:9–33, 2003.

[15] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[16] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1991.

[17] G. R. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, Computer Science Department, 2000.

[18] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems (TODS)*, 28:517–580, December 2003.

[19] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5), 1983.

[20] L. Micó, J. Oncina, and R.C. Carrasco. A fast branch and bound nearest neighbor classifier in metric spaces. *Patt. Recog. Lett.*, 17:731–739, 1996.

[21] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Patt. Recog. Lett.*, 15:9–17, 1994.

[22] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[23] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. PAMI*, 19(9):989–1003, 1997.

[24] H. Nolteimer, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203. Springer-Verlag, 1992.

[25] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *IPL*, 40:175–179, 1991.

[26] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Patt. Recog. Lett.*, 4:145–157, 1986.

[27] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. SODA'93*, pages 311–321, 1993.