Journal of Information and Data Management, 2024, 15:1, doi: 10.5753/jidm.2024.3060

Usage of the Bag Distance Filtering with In-Memory Metric Trees

Sergio Luis Sardi Mergen [**Universidade Federal de Santa Maria** | ***mergen@inf.ufsm.br***]

Universidade Federal de Santa Maria, Brazil

Received: 14 December 2022 • **Published:** 17 April 2024

Abstract Metric trees are efficient indexing structures for multidimensional objects defined in terms of a metric space. One possible application is for string similarity search, using the edit distance as the metric function. A previous work proposes clustering objects under leaf nodes and using the bag distance as a filtering step before the edit distance is computed. Cost predictions estimate that the filtering compensates in practical scenarios. The work has important implications when data resides on secondary storage, where nodes have a fixed size that aligns with page disks. In this paper, we expand the discussion by using the bag distance filtering step for in-memory metric trees, where the clusters have no size constraints. We adjust existing metric trees to support leaf nodes with arbitrary cluster sizes and incorporate parameters based on size and density to decide when a leaf node should be subdivided. Experiments show that cluster size can have a substantial impact during both index construction and search. We report the gains achieved in terms of processing cost and the number of distance computations when using the most suited values for the cluster size and density parameters.

Keywords: Metric space, Metric trees, Similarity search

1 Introduction

The similarity search problem focuses on finding objects similar to a query object. It is an integral part of many correlated areas, such as pattern recognition, computational biology, and spell-checking.

If the similarity is inferred by a distance function, and the function is a metric, the metric space formed by the set of objects can be indexed based on the relative distance of the objects. According to a recent study Chen *et al*. [2017], metric trees serve as effective indexing structures when the objects can be stored within the computer's primary memory. In a pivot-based metric tree, pivot nodes lead to regions of the space that contain objects whose distances satisfy a lower or upper bound.

Traditionally, the edit distance(a.k.a the Levenshtein Distance) is used as a metric function for string similarity search Traina Jr *et al*. [2007]; Deng *et al*. [2016]; Chen *et al*. [2017]. It computes the distance between two strings as the minimum number of edit operations needed to transform one string into another. The more edit operations, the less similar they are Levenshtein [1966].

The edit distance can be used along with other functions as well. In Traina *et al*. [2002]; Bartolini *et al*. [2002], the cheaper bag distance is used as a filtering step. The general idea is to cluster strings inside leaf nodes. During a search, once a leaf node is accessed, the edit distance between the query and a clustered string only needs to be computed if the string qualifies in terms of the bag distance. One work in particular shows that the filtering usually pays off using cost predictions based on the general distribution of objects Bartolini *et al*. [2002].

The bag distance filtering step was applied over secondary storage metric trees, where nodes have a size that aligns with page disks. Consequently, leaf nodes have a fixed cluster size. We argue that the filtering is also possible for in-memory metric trees, where there are no restrictions on the size of the clusters. This opens an opportunity to verify how the usage of variable sizes affects index construction and search.

The size of the clusters is important for metric spaces based on edit distance. This metric presents a normal-like pairwise distance distribution with a very steep curve, especially when considering regions of the space where objects are closer to each other. It means that, when indexing a dense enough region, objects may become equidistant, which can potentially jeopardize the pursuit of building balanced search trees.

We propose an unbalanced index construction method that stops subdividing the space if one of two conditions is satisfied: the cluster is considered small enough or dense enough. During construction, two parameters are defined to control the desired size and density. Our evaluation demonstrates that the combination of properly unbalanced indexes with the bag-distance filtering step has the potential to reduce the elapsed time of range and top-k queries.

This paper is organized as follows: in Section 2 we present the theoretical background regarding metric space search and bag distance filtering. In Section 3, we reason that defining clusters based on size and density can reduce search processing. We also propose extensions to well-known metric trees to properly evaluate indexes whose cluster size varies according to the data being indexed, regardless of the size. Section 4 evaluates the extended trees with the clustering parameters for the problem of similarity search over a dictionary. Final remarks are given in Section 5.

2 Background

Let a metric space $M = \{d, U\}$ be defined as a data domain U and a distance function $d : U \times U \rightarrow R^+$, where d needs to be a metric. The distance $d(x, y)$ measures how far objects x and y are from each other. It is a metric if, for any x, y, z $\in U$, four properties are satisfied:

- Symmetry: $d(x, y) = d(y, x)$.
- Non-negativity: $d(x, y) >= 0$
- Identity: $d(x, y) = 0$ iff $X = Y$
- Triangular Inequality: $d(x, y) \leq d(x, z) + d(y, z)$

Objects of a metric space can be indexed according to their relative distance from each other. In Chen *et al*. [2017], indexing methods are classified as compact-partitioning based and pivot-based, according to the strategy used to divide the space and the existence of precomputed distance between pivots and objects. In this paper, we use the generic term metric-tree as a reference to all methods that index objects according to their distance to a selected set of pivots and use a tree as a data structure. There is a diversity of metric trees. Some are tailored for secondary storage, such as M-trees, while others work with nodes fully allocated in the main memory. The study of Chen *et al*. [2017] revealed that metric trees have the best performance both in terms of construction and search times, for data sets that can be fully allocated in memory. The evaluation used the following data structures: Vantage Point trees (VPT), Multiple Vantage Point Trees (MVPT), and Burkhard-Keller trees (BKT).

VPT is a binary tree that puts in the left sub-tree the objects whose distance to the pivot (the vantage point) is not greater than a radius. The remaining objects are put in the right sub-tree. Objects in a sub-tree are further partitioned into smaller sets. All indexed objects are clustered inside leaf nodes, which are partitioned when the capacity is exceeded Yianilos [1993]. The tree is intended to be balanced. The balancing is achieved by a careful selection of pivots and their corresponding radius Zhu *et al*. [2022]. Better pivots are chosen if all objects are known before index construction. Figure 1 shows an example, where the capacity of the leaf node was set to two.

MVPT is a generalization of VPT that supports n-ary trees, where a pivot has $n - 1$ radius. Objects whose distance to the pivot is not greater than radius i are put in sub-tree i. Objects whose distance is greater than the last radius are put in the right-most sub-tree. Objects are indexed according to the first qualified radius. Higher n values lead to shorter trees, and consequently, a reduced number of pivots Bozkaya and Ozsoyoglu [1997].

BKT is an n-ary tree where each path connects child nodes that share the same distance to the parent node (a pivot) Burkhard and Keller [1973]. Unlike the above-mentioned structures, it was designed for applications that use a discrete metric function, such as the edit distance. Figure 2 shows an example. Observe that objects are not clustered inside leaf nodes, and may appear as inner nodes as well. There is also no care about balancing.

Queries over a metric space are basically posed in terms of an object q associated with a search radius r (a ball drawn around q). In range queries, the radius is fixed (the maximum allowed distance). In nearest neighbor queries (top-k), the radius varies during the search according to the nearest neighbors found so far.

During a search, triangle inequality helps discard unqualified objects without computing their actual distance to q. For instance, given the query object q, a pivot p, and an object o, if $d(q, p)$ and $d(p, o)$ are known, we can compute the lower and upper bounds of $d(q, o)$ as $|d(q, p) - d(p, o)|$ and $d(q, p) + d(p, o)$, respectively. If the search radius is lower or equal to the lower bound, the object o does not qualify. If it is greater or equal to the upper bound, the object o directly qualifies.

Similarly, several techniques use triangle inequality to discard entire regions of a metric space accessible from a pivotZezula *et al*. [2006]. For instance, considering that $d(q, o)$ needs to be within 0 and the search radius r, the following holds: $d(q, p) - r <= d(p, o) <= d(q, p) + r$. In BKT, a child node is visited if its distance to the parent falls within that range. In VPT, the left sub-tree of a pivot is visited if the distance between the query and the pivot is sufficiently small. One of two conditions must be satisfied: $d(q, p) <= radius$, which puts the query object inside the left sub-tree, or $d(q, p) <= radius + r$, meaning the search radius overlaps with the left sub-tree. On the other hand, the right sub-tree of a pivot is visited if the distance between the query and the pivot is sufficiently large. One of two conditions must be satisfied: $d(q, p) > radius$, which puts the query inside the right sub-tree, or $d(q, p) > radius - r$, meaning the search radius overlaps with the right sub-tree.

In general, the processing of a search encompasses two costs: Internal complexity and external complexity. The former refers to the cost of navigating through the pivot nodes to find candidate objects. The latter refers to the costs of checking which of the candidate objects qualify.

For typical BKT, the internal/external complexities are indistinguishable, since objects appear as non-leaf nodes also. For VPT/MVPT, the internal complexity tends to be lower than the external complexity, considering that the number of pivot nodes accessed is much lower than the number of leaf nodes.

2.1 The Bag Distance Filtering

The bag distance is a metric that compares objects represented as multi-sets. A multi-set X is defined as the set of elements $e \in E$, where E is the set of possible symbols and $X(e)$ gives the multiplicity of e in X.

To avoid confusion regarding multiplicity, we define the semantics of some operations over multi-sets.

Definition 1 *(Multi-Set Difference)*
The difference of multi-sets X and Y, denoted as $X - Y$, is defined as:

$$\forall e \in E : (X - Y)(e) = \begin{cases} X(e) - Y(e), & \textit{if X(e) > Y(e).} \\ 0, & \textit{otherwise.} \end{cases}$$

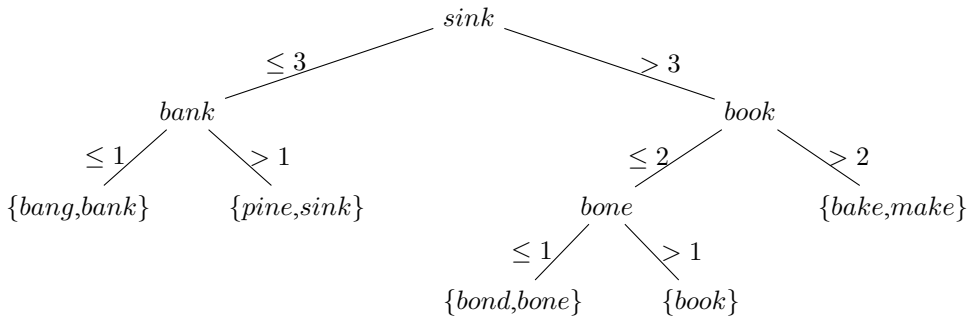


Figure 1. A Vantage-Point Tree

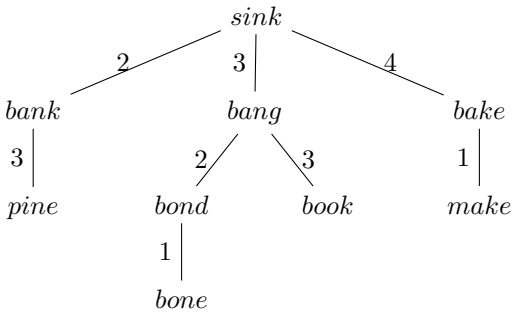


Figure 2. A Burkhard-Keller tree

Definition 2 *(Multi-Set Cardinality)*
The cardinality of a multi-set X, denoted as $|X|$, is defined as:

$$|X| = \sum_{e \in E} X(e)$$

To conclude, we present the definition of the Bag Distance:

Definition 3 *(Bag Distance)*
The Bag Distance between two multi-sets X and Y, is defined as:

$$d_B(X, Y) = max((|X - Y|, |Y - X|)$$

In essence, the bag distance is measured as the maximum number of distinct elements between two multi-sets. This is a general definition that encompasses every object that may be identified by a multi-set.

This metric can be used to compare strings, by taking the multi-sets of characters from the strings. To understand, Table 1 presents the symbols used to represent strings and multi-sets of characters. The notation clarifies how the bag distance copes with the string domain.

For example, considering A contains characters from the Latin alphabet, the strings 'bang'(s_1) and 'banana'(s_2) from the data domain S are represented as the multi-sets X_1 = {'a','b','n','g'} and X_2={'a','a','a',','b','n','n'}, respectively. The bag semantic implies that characters may appear multiple times (such as 'a' and 'n' in X_2). In this particular case, the bag distance between 'bang' and 'banana' is 3, since X_2 has three distinct characters ('a','a','n'), while X_1 has only one ('g').

In Bartolini *et al.* [2002], the authors observed that the bag distance d_B is a lower-bound of the edit distance d_E ($d_B \preceq d_E$), meaning that $\forall s_1, s_2 : d_B(s_1, s_2) \leq d_E(s_1, s_2)$. In other words, the distance computed by d_B is never higher than the distance computed by d_E.

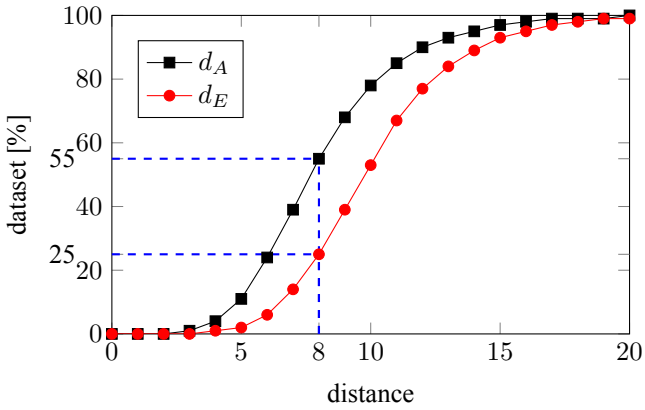


Figure 3. A hypothetical distribution considering the maximum distance among objects.

The authors used a bag distance filtering step to optimize search operations in a secondary storage metric tree (M-TREE). As some in-memory metric trees (such as VPT and MVPT), the leaf nodes of a M-TREE contain direct access to a group of objects(a cluster). The difference is that the nodes are typically larger since they are mapped into disk pages. When a leaf node is accessed, the bag distance is computed for every candidate object inside the cluster. Given the lower bound property, the edit distance only needs to be computed if the corresponding bag distance is not greater than the search radius.

It takes linear time to find the bag distance between two strings s_1 and s_2 Kahveci and Singh [2001], whereas the edit distance uses a dynamic programming solution that has a quadratic complexity ($O(s_1.s_2)$) Needleman and Wunsch [1970]. There are super-linear approaches Papamichail and Papamichail [2009]; Li *et al.* [2011], but they are not suited for measuring distances between short strings, given they have large constants of proportionality. Hence, the filtering compensates as it trades the edit distance with a less expensive function. However, the benefits depend on how selective the filtering is.

The authors provide cost predictions of the external complexity based on the distribution of the objects. A typical distribution is depicted in Figure 3. $F_{d_x}(r)$ represents the probability of the distance d_x between two random strings be at most r. For instance, given a random object q and the search radius 8, 25% of the data set is within the search ball drawn around q, using d_E as the distance function ($F_{d_E}(8) = 25\%$).

The estimation derives from the fact that the bag distance

Symbol	Meaning
s	a string object
S	the data domain for strings
X	a multi-set object containing characters
A	the alphabet of characters
c	one character from the alphabet
$d_E(s_1, s_2)$	the edit distance between strings s_1 and s_2
$d_B(s_1, s_2)$	the bag distance between the multi-sets formed by the characters taken from strings s_1 and s_2

Table 1. Notations concerning strings and multi-sets of characters.

is required for all candidate objects, whereas the edit distance is required for the subset that qualifies in terms of the bag distance. Given that, and assuming $cost_B$ and $cost_E$ are the costs for the bag distance and the edit distance, respectively, the savings S in search time of the bag distance filtering step is estimated as indicated in Equation 1. In short, the overhead increases as the bag distance becomes less selective and the cost of finding the bag distance arises. For typical applications, which use a small search radius to retrieve a few objects per query, the overhead is small, and the filtering step pays off.

$$S = 1 - F_{d_B}(r) - \frac{cost_B}{cost_E} \tag{1}$$

3 Changing the Cluster Formation

The work of Bartolini *et al*. [2002] explores the bag distance filtering step in the context of secondary storage metric trees. In this case, the IO costs prevail, so it is more important to reduce the number of random page reads than meet any other in-memory optimization. However, completely in-memory indexes also have room for improvement.

Despite the unprecedented amount of available data, where secondary storage indexes have become indispensable, new approaches to in-memory indexing remain relevant. This is evidenced by recent studies focused on enhancing data retrieval relying solely on data structures that fit in main memory Sprenger *et al*. [2019]; Jensen *et al*. [2021].

We argue that in-memory indexes can benefit from the filtering step to improve search, and the improvement is strongly related to the cluster size and the distribution of the objects inside a cluster. In what follows we explore this issue in greater detail. We also present extensions to well-known metric trees that enable the clusters formation to be explored more effectively.

3.1 Partitioning the Metric Space

To understand why cluster formation is important, at least for in-memory trees, we rely on an example based on a VPT (Figure 4). The goal of VPT is to divide the space into non-overlapping regions. The space gets subdivided if the number of objects reached lies above a predefined capacity. In the example, assume the number is eight. Two pivots were defined: $p1$ and $p2$. A pivot separates the objects into two regions: an inner region consisting of objects whose distance to the pivot is smaller than a specified radius, and an outer region that encompasses the remaining objects, which are farther away. The inner region of a pivot is defined by the dashed circle surrounding the pivot. The dotted circle indicates the objects that are not part of any inner region.

The Vantage-Point tree recursively connects pivot nodes according to the inner and outer regions they map. Finally, the regions accessible by leaf nodes form the clusters. A similar structure applies to BKT, except that a pivot leads to a number of regions defined by concentric hyper-spheres drawn around the pivot. In either case, as the space gets subdivided, smaller and denser regions are created. Also, more pivots are added, which increases the internal complexity of a search.

Ideally, the space should be divided into smaller and denser regions up to a point where search performance is maximized, by reducing the overall cost of the internal and external complexities. The key part is knowing when to stop partitioning.

To explore this issue, consider two alternative arrangements:

- **a)** a region composed by n objects forms a single cluster. During a search, if the cluster is reached, all objects need to be verified.
- **b)** instead of forming a single cluster, a pivot is selected and the n objects are split into x sub-regions with the same number of objects each. During a search, if the cluster is reached, the distance between the pivot and the query is computed. Based on the distance found, all objects from the y qualifying sub-regions need to be verified.

The number x refers to the degree of the pivot node (the number of regions accessed by it). In a VPT, the degree is two. MVPT supports a higher degree, as well as BKT. Here, we assume a generic scenario where x can be greater than two.

Given a query, arrangement **a** takes n bag distance operations to find the candidate objects. Conversely, arrangement **b** (with partitioning) requires one edit distance (for the pivot) and $n \times \frac{y}{x}$ bag distances, to find the candidate objects. Equation 2 estimates when the arrangement **b** compensates.

$$\begin{aligned} cost_E + cost_B \times n \times \frac{y}{x} &< cost_B \times n \\ \frac{cost_E}{cost_B} &< n \times (1 - \frac{y}{x}) \\ n &> \frac{\frac{cost_E}{cost_B}}{(1 - \frac{y}{x})} \end{aligned} \tag{2}$$

There is a direct correspondence between the size of a cluster and the ratio between the edit and the bag distance operations. The higher the $cost_E$ with respect to $cost_D$, the larger

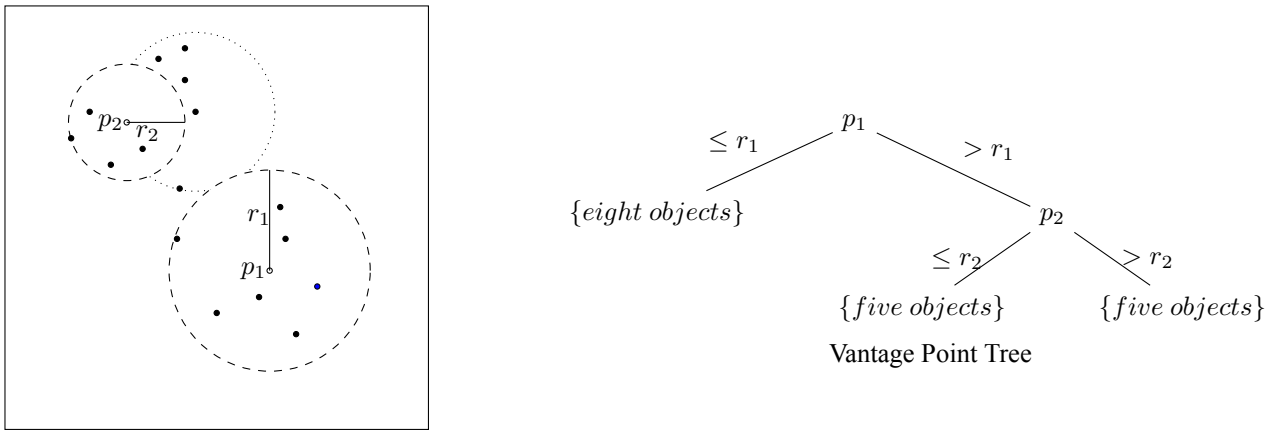


Figure 4. The distribution of objects according to a metric space using a Vantage Point Tree.

the cluster should be. Intuitively, as partitions get smaller, the internal complexity becomes more significant when compared to the external complexity. For a small enough cluster, it is better to stop partitioning, as it is cheaper to compute the bag distance to the inner objects than to compute the edit distance to an additional pivot node that gives access to an even smaller cluster.

Provided that the size of a cluster can affect a search, we propose using a clustering rule based on size, as follows:

Partition Rule 1 *During indexing, if a cluster has less than a pre-defined number of objects cs (cluster size), the partitioning stops.*

This is already a natural partitioning rule of cluster-based metric trees, such as VPT: if the number of clustered objects overcomes a capacity (eight, in the example of 4), a new partitioning occurs. In this paper, we expand the discussion by proposing a new partition rule, as discussed next.

According to Equation 2, the most suited size of a cluster is inversely proportional to the number of regions that do not qualify during a search($1 - \frac{y}{x}$). The higher the number of qualifying regions ($\frac{y}{x}$), the larger the cluster should be.

In other words, if a pivot fails to efficiently guide the search towards a smaller area within the space, it is preferable to maintain the objects clustered rather than attempting to subdivide them. The pruning ability of a pivot is very query dependent. A pivot is less capable of reducing the search space if the query has a large search radius. We can infer that, if the clustered objects are sparse (distant from each other), the pivot tends to do a better job.

Provided that the proximity of the objects is valuable information to this end, we propose using an additional clustering rule based on distance, as follows:

Partition Rule 2 *During indexing, the partitioning stops if the clustered objects are closer to the pivot than a pre-defined threshold dp(distance to pivot).*

Figure 5 demonstrates the application of the partition rules. The metric space A (on the left) is the same one presented in Figure 4, where the objects are partitioned based solely on Partition Rule 1 (clusters with more than 8 objects are subdivided).

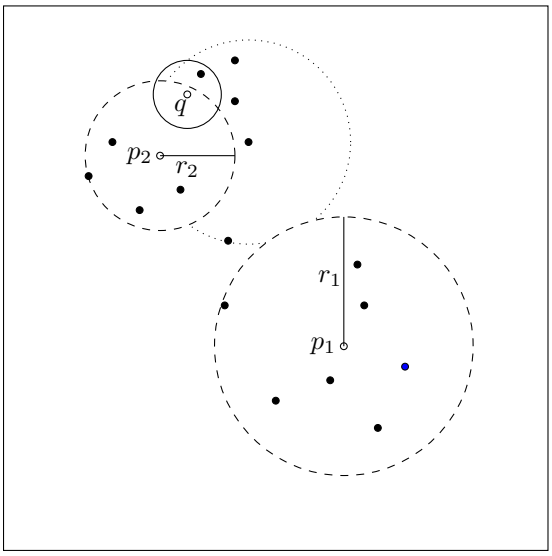
In contrast, the metric space B (on the right) requires the validation of both partition rules to subdivide the space. In this case, assume that the region outside $p1$ is not subdivided because it breaks Partition Rule 2: the furthest object in that region is close enough to the pivot $p1$ (the distance is lower than dp). As a result, we end up with fewer partitions, by merging partitions where objects are closer.

We reason that, during a search, all objects from dense regions tend to be accessed. In this case, the existence of more partitions imposes an overhead of having to compute the edit distance between the query and the additional pivots.

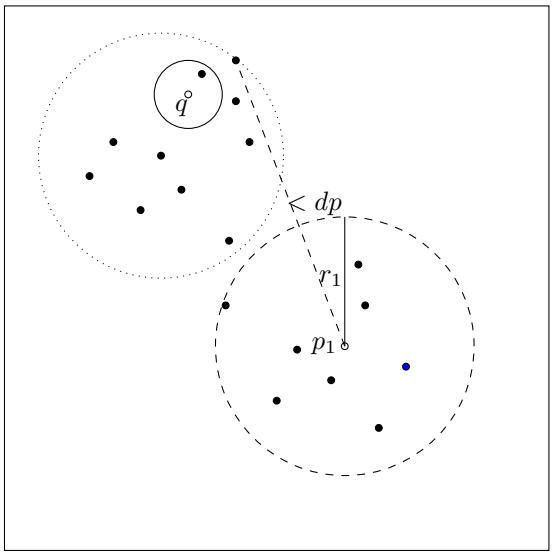
To exemplify, consider that the query q is posed (as described in Figure 5). The circle surrounding q determines the search radius (the search ball). Given metric space B, when the search reaches the outer region of $p1$, there is no need to compute the edit distance between the query and a pivot, so we can proceed directly to the filtering step, where the Bag distance is computed.

On the other hand, given metric space A, the edit distance between $p2$ and q has to be computed, so we can infer the relative position of the search ball with respect to the two regions mapped by $p2$. If the search ball overlaps with the inner region, the objects that belong to the $p2$ cluster need to be accessed. If the search ball overlaps with the outer region, then the objects farther away from $p2$ need to be accessed. The worst-case scenario happens when the search ball overlaps the two regions (as the example illustrates) since this requires the search to span across both regions. Therefore, in both metric spaces, all objects in the outer region of $p1$ need to be accessed. However, in metric space A, there is an overhead related to the edit distance computation.

Of course, this example, along with Equation 2, is a simplification that makes a strong assumption about how the objects are distributed and queried. In real scenarios, objects may be unevenly divided into sub-regions. Nevertheless, our reasoning shows that the cluster's size and density have the potential to impact query efficiency, and should be chosen carefully, as we demonstrate in the Experimental Results section.



Metric Space A - Using Partition Rule 1



Metric Space B - Using Partition Rules 1 and 2

Figure 5. Cluster formation and querying considering two settings: using only the cluster size to stop partitioning (Metric Space A) and using the cluster size and distance to pivot to stop partitioning (Metric Space B)

3.2 BKT Extension

Among the well-known metric trees, the BKT has some unique features: it was designed for discrete functions with short ranges and it is unbalanced by nature. To a large extent, those factors explain the common architectural designs behind this data structure. Firstly, the trees have an unbounded arity since allocating a sub-tree to each possible distance value is affordable. Additionally, there is a lack of concern with clustering objects into a single node. During insertion, typical implementations simply follow a path from the root to a leaf node, where the object is inserted as a new sub-tree. As a consequence, objects may appear as inner nodes.

To provide cluster support, two properties were added to the BKT nodes:

- **cluster:** the set of objects clustered inside a leaf node.
- **distance to pivot:** the distance between each of the clustered objects and the node's pivot.

The pseudo-code in Algorithm 3.1 shows how insertion handles the existence of clusters. Once a leaf node is found, the new object o becomes part of the cluster. If the conditions in line 3 are satisfied, the cluster is cleared and its objects are added as sub-trees. If the size of the cluster lies below cs, the partition stops (Partition Rule 1). The partition also stops if the distance between the clustered objects and the pivot is lesser than dp (Partition Rule 2). Observe that, with BKT, all clustered objects share the same distance to their respective pivot. The $bkt\_add\_node(o)$ function refers to the original method that ends up creating a sub-tree to allocate the new object o.

```
EXTENDED_ADD_NODE(o)
 1: if current_node is a leaf node then
 2:     add o to the cluster
 3:     if cluster size exceeds cs and distance to pivot > dp
        then
 4:         for all objects o_i in current_node.cluster do
 5:             remove o_i from current_node.cluster
 6:             bkt_add_node(o_i)
 7:         end for
 8:         current_node.clustered ← false
 9:     end if
10: else
11:     bkt_add_node(o)
12: end if
```

Algorithm 3.1: Adding a node to a BKT

3.3 VPT Extension

Unlike BKT, clustering objects as single leaf nodes is part of the VPT conception. A cluster is partitioned whenever its size exceeds a predefined capacity. This behavior naturally satisfies (Partition Rule 1), which states that the partition should stop if the cluster size lies below cs.

However, there is a caveat that generally passes unnoticed. It is not possible to split a set when all objects are equidistant. If that occurs, the partition routine enters an infinity loop failing to segregate the data even further. The problem can occur when the capacity is shorter than the equilateral dimension (the maximum number of equidistant elements) Blumenthal [1954]. In a d-dimension Euclidean space, the equilateral dimension is $d + 1$. In a space defined by the edit distance function, it is directly proportional to both d and the alphabet size.

Take for instance four-lettered words. In terms of edit distance, the equilateral dimension is not five. Instead, it is bounded by the number of completely different words that can be formed using four characters from the alphabet. The number is definitely larger than five. Of course, those words are separated during indexing. Still, some of them may fall into the same cluster. If the cluster capacity is smaller than the number of equidistant words, the partition will fail. VPT and MVPT usually chose a cluster size that bypasses the problem. Another approach is to remove the pivot from the leaf level to prevent going into an infinite loop, which would form a highly unbalanced sub-tree.

We propose a workaround that accepts clusters of any size. The extension is useful for evaluating how a VPT behaves when clusters are very small. The extension adds a rule that stops the partitioning whenever the previous split put all objects into the same sub-tree. The pseudo-code is presented in Algorithm 3.2. The function that recursively splits a sub-tree is only called if both sub-trees are not empty (line 14). The code can be easily incorporated into MVPT, by testing a variant number of sub-trees instead of only two.

The code also shows how to locate the object that is further away from the pivot vp(lines 8 and 11 find the object from the left and right sub-trees, respectively). This found distance is used to decide when the partitioning should stop (line 1), according to (Partition Rule 2).

```
BUILD_VPT(maxDist)
 1: if cluster size exceeds cs and maxDist > dp then
 2:    vp ← chooseVP(current_node.cluster)
 3:    radius ← findMedianDistance(current_node.cluster,
       vp)
 4:    for all objects o in current_node.cluster do
 5:       remove o from current_node.cluster
 6:       if d_E(o, vp) ≤ radius then
 7:          add o to the cluster of the left_sub-tree
 8:          maxLDist ← max(maxLDist, d_E(o, vp)
 9:       else
10:          add o to the cluster of the right_sub-tree
11:          maxRDist ← max(maxRDist, d_E(o, vp)
12:       end if
13:    end for
14:    if both subtrees are not empty then
15:       left_subtree.build_VPT(maxLDist)
16:       right_subtree.build_VPT(maxRDist)
17:    end if
18: end if
```

Algorithm 3.2: Adding a node to a VPT

4 Experimental Results

Our primary goal is to check how performance is affected when varying the size and the density of the clusters. The parameters cs (cluster size) and dp(distance to pivot) are used to control the cluster size and the density, respectively. Recall that, during indexing, the partition stops if either the cluster size is less than cs or the distance between all elements and their corresponding pivot is lower than dp. A larger dp leads to fewer partitions, since sparser clusters are accepted. When $dp = 0$, only the cluster size is used as a partitioning factor.

The two extended metric trees were used: BKT and VPT. VPT relies on a pivot selection strategy that best separates objects in a metric space. The technique chooses, from a list of five random candidates, the one whose distance to the other objects is maximized.

The data set was taken from the Moby project[1]. It contains a list with 593.248 proper nouns, acronyms, and compound words. This dictionary was used to demonstrate the efficiency of metric trees for in-memory indexing Chen *et al*. [2015, 2017]. The query set is composed of a sample of 100 objects taken from the data set. Similar results were found using dictionaries of seven different languages taken from the Metric Space Library projectFigueroa *et al*. [2007].

Two query strategies were used:

- Range-d Query: The purpose is to find the indexed objects that are within a maximum distance of d from the query object.
- Top-k Query: The purpose is to find the k indexed objects that are closer to the query object.

The implementation of the top-k strategy is an extension of the range-d strategy, where the maximum distance varies according to the current closest object. Since the search radius starts with a high value, and only reduces as closer objects are found, the top-k strategy tends to access more regions of the space, when compared to the range-d strategy which has a fixed search radius.

The evaluation was held on an Intel Core i5-3470@3.20GHz, with 8GB of RAM. Run-time results were measured as the average of 30 executions, ignoring the top and the bottom 10%. All code is in Java.

4.1 Construction Performance

Table 2 shows statistics regarding the metric trees when varying the defined cluster size cs from 2^1 to 2^{10}. For now, only the cluster size is used as a partitioning factor. The impact of the dp parameter is discussed later.

VPT and BKT are binary and n-ary trees, respectively, as the table shows. The average arity of BKT increases as clusters get larger. The reason is simple. In larger clusters, the inner elements are more scattered around their corresponding pivot, which increases the fan-out required to index every distance found. A higher arity implies that the height of the tree is lower. It does not necessarily mean that search operations are cheaper since a search can span across multiple branches, as we discuss later.

The average cluster size naturally increases according to the defined cluster size. The cs value is an upper bound of the actual cluster size, as clusters larger than cs are necessarily partitioned. Additionally, as the average size increases, the number of clusters decreases. Observe that BKT has smaller clusters than VPT, which has a direct correspondence to the arity.

Figures 6 and 7 show the number of edit distance computations and the time required to construct the metric trees, respectively. In general, the performance of both search trees increases as the clusters grow. There is an intuitive reason. Once the objects in a specific region fit into a cluster, the partition stops. Larger clusters lead to shorter trees and, consequently, fewer edit distance computations.

The results show that BKT is better at index construction. There is no concern regarding the production of a balanced outcome, which considerably reduces the number of edit distance computations when deciding which branch a node should follow. It produces imbalanced trees as a downside,

[1] https://en.wikipedia.org/wiki/Moby_Project

	avg arity		max height		avg clusters size		number of clusters	
defined cluster size	BKT	VPT	BKT	VPT	BKT	VPT	BKT	VPT
2	3	2	23	35	1	1	75,219	451,808
4	4	2	21	33	2	2	105,142	208,773
8	5	2	18	32	3	5	95,054	97,058
16	6	2	17	30	5	10	67,436	46,760
32	7	2	16	28	9	20	41,938	23,001
64	7	2	13	27	17	40	24,632	11,410
128	8	2	10	24	31	79	13,945	5,727
256	8	2	9	21	57	157	7,756	2,878
512	9	2	8	18	106	315	4,210	1,435
1,024	9	2	6	15	196	635	2,289	711

Table 2. Anatomy of BKT and VPT

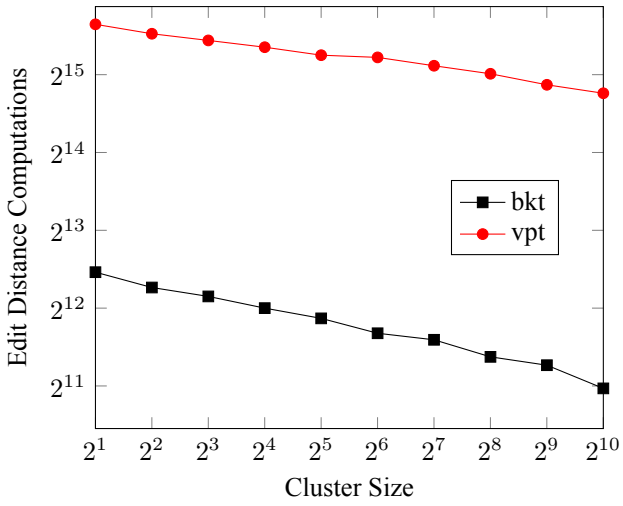


Figure 6. Edit distance computations of BKT and VPT during Index Construction

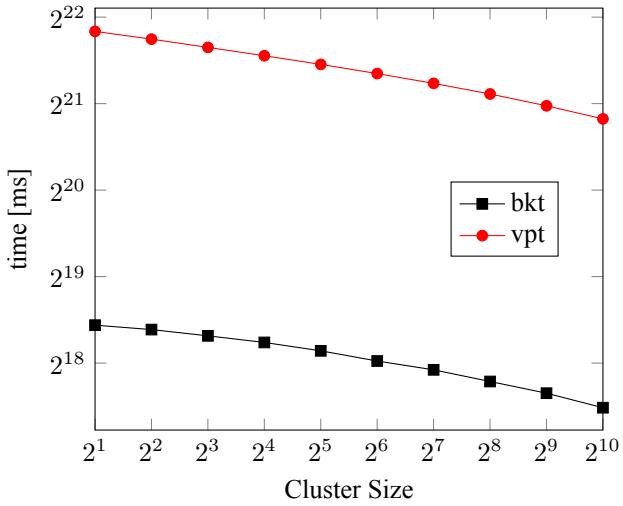


Figure 7. Run-time of BKT and VPT during Index Construction

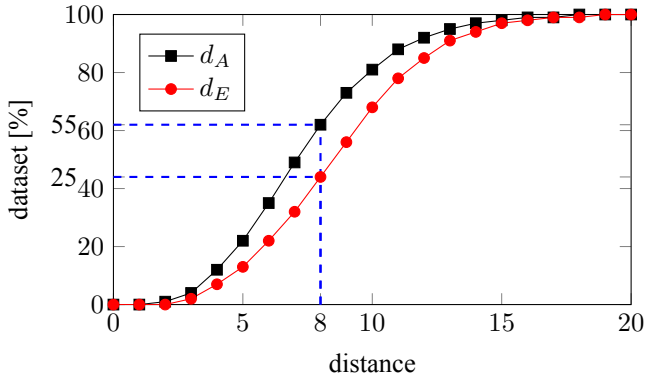


Figure 8. Distribution considering the maximum distance among objects using the Moby dataset

unlike VPT. However, as we'll see below, the search performance cannot be measured solely based on internal complexity.

4.2 Pairwise Data Distribution

As described in Section 2.1, the importance of the filtering step depends upon both the selectivity of the bag distance and its proportional cost with respect to the edit distance. In this section, we demonstrate that the filtering step indeed compensates, by analyzing the pairwise data distribution among objects when using both distances.

Figure 8 shows the results found. The curve is similar to the ones shown in Section 2.1(Figure 3). Observe that the bag distance loses selectivity as the distance grows. This indicates that it becomes less appealing when the search radius is high.

We have also estimated the proportional cost of the bag distance and the edit distance. For that, we computed the bag and edit distances among objects from the query set and the whole data set and compared the elapsed times. As a result, the bag distance cost is approximately 0,1% of the edit distance cost. It means that, even when working with a higher search radius, the filtering step is still very likely to provide a compensating trade-off.

4.3 Search Using a Fixed Search Radius

Similarity search demands a proper search radius. A high radius ends up bringing irrelevant strings. On the other hand, a small radius may be too narrow. In the context of string

search over dictionaries, normally a maximum radius of two suffices for finding similar objects for most applications Lee *et al*. [2007]. In what follows, we show performance when the search radius varies from one to three.

Figure 9 shows the results achieved when answering range-1 queries and using only the cluster size to determine whether a region should be subdivided during indexing. The right part reports the number of distance computations required. As the cluster size grows, the number of bag distance computations naturally increases. There is also a reduction in the number of edit distance computations. The reasons are twofold: i) there are fewer pivots, which reduces the internal complexity and ii) as the cluster size increases, the strings in a cluster become farther apart from each other. In those cases, bag distance filtering is more effective in pruning.

The left part of Figure 9 shows the time spent searching. It is interesting to observe how the run-time first drops gradually and then increases. The curves are a direct result of the inverse proportionality between the number of bag distances and edit distances computations. The best cluster size is 32, for both BKT and VPT. This is the value whose summed cost of the internal and external complexities is minimum. Additionally, VPT is approximately 25% more efficient than BKT.

We now compare the best results found with and without the application of the filtering step (Table 3), when performing range queries whose range varies from one to three.

When no filtering is used, BKT achieved its best result with unclustered objects(cs = 1). The reason is simple. When objects are grouped, the edit distance would have to be computed against all clustered objects. If objects are sub-partitioned, there is always a chance to prune sub-trees that lead to non-qualifying regions. On the other hand, the unfiltered VPT works best when objects are grouped into small clusters. As opposed to BKT, our VPT implementation stores all objects in leaf nodes. Consequently, some objects will appear both as pivots as internal and external nodes. During a search, there may occur cases when the edit distance has to be computed twice for the same duplicate object. By reducing the internal complexity, we reduce the number of duplicate computations. Observe that, when the filtering is used, it is better to work with a higher cluster size, to reduce the external complexity by performing fewer edit distance computations for the clustered objects.

It is important to notice that the filtering step always led to a reduced cost. Also, the improvement is more meaningful as the search radius increases, as indicated by the savings described in the last column. At first, this finding appears dissonant with Equation 1, which states that the savings decay as the Bag Distance becomes less selective. However, range-3 queries are still selective enough. As these queries reach a higher portion of the metric space, and considering most of the objects reached are irrelevant, the filter is important to reduce the number of edit distance computations to those irrelevant objects.

Using the dp Partitioning Rule: The previous results were based on indexes built without the usage of the dp partitioning rule. In what follows we analyze how this factor affects the run-time. Recall that the dp parameter is used to stop partitioning if the cluster is considered dense enough. A high value means that a sparser cluster is acceptable and the partition can stop, regardless of the cs parameter. The value zero means that only the cluster size is used as a partitioning factor. Also, when cs is one, only dp is used as a partitioning factor. Higher values for dp and/or cs lead to larger clusters.

Figure 10 analyzes the results found for range-1 queries. For each cs value, the graph shows the dp value that achieved the best performance. Observe that, when cs is small, a higher dp is more suited, as a means to prevent the partitioning of clusters that are already small. On the other hand, when cs is too high, it is better to perform the partitioning, regardless of the density ($dp = 0$). The best setting for both BKT and VPT occurs when only the cs rule is applied ($dp = 0$ and $cs = 2^5$). Conversely, the settings where only the dp rule is applied do not yield good performance ($dp = \{2$ and $3\}$ and $cs = 2^0$).

Table 4 reports the results achieved when using the best cluster size (2^5) and a variable dp. Observe that higher dp values are indeed associated with larger clusters and, consequently, fewer clusters. Also, the search cost is higher when working with larger clusters. The results indicate that, for searches with a small radius (like range-1), it is not beneficial to stop partitioning based on the density within a cluster.

4.4 Search Using Top-k

Using a fixed search radius does not necessarily yield results. If the purpose is to return the closest objects to a query, it is better to use top-k queries, where k determines the exact amount of objects to be returned.

Figure 11 shows the results achieved when answering top-1 queries and using only the cluster size to determine whether a region should be subdivided. The first thing to notice is that top-1 queries are much costlier than range-1 queries. The best top-1 run-time is approximately 16 times more expansive than the best range-1 run-time. This happens because top-1 queries access a larger area of the metric space to find the closest objects, which increases the number of edit distance operations that need to be performed.

In such a scenario, where a wider region of the space is accessed, it is better to divide the space into larger clusters, so that the number of pivots accessed (and consequently, the internal complexity) is reduced. Indeed, the best results occur with a cluster size of 32 and do not change meaningfully when working with even higher values.

It is also interesting to observe that BKT is better than VPT, as opposed to the results found when working with range-1 queries. This indicates that BKT subdivides the space in a way that favors queries with a high search radius.

We now compare the best results found with and without the application of the filtering step (Table 5), when performing top-k queries with k=1, k=5, and k=10.

The results resemble the ones found when working with range queries: i) when no filter is applied, it is better to work with a reduced cluster size; ii) the filtering step always leads to a reduced run-time and iii) the savings of the filter does not deteriorate when working with less-selective queries (except for using VPT to answer top-10 queries).

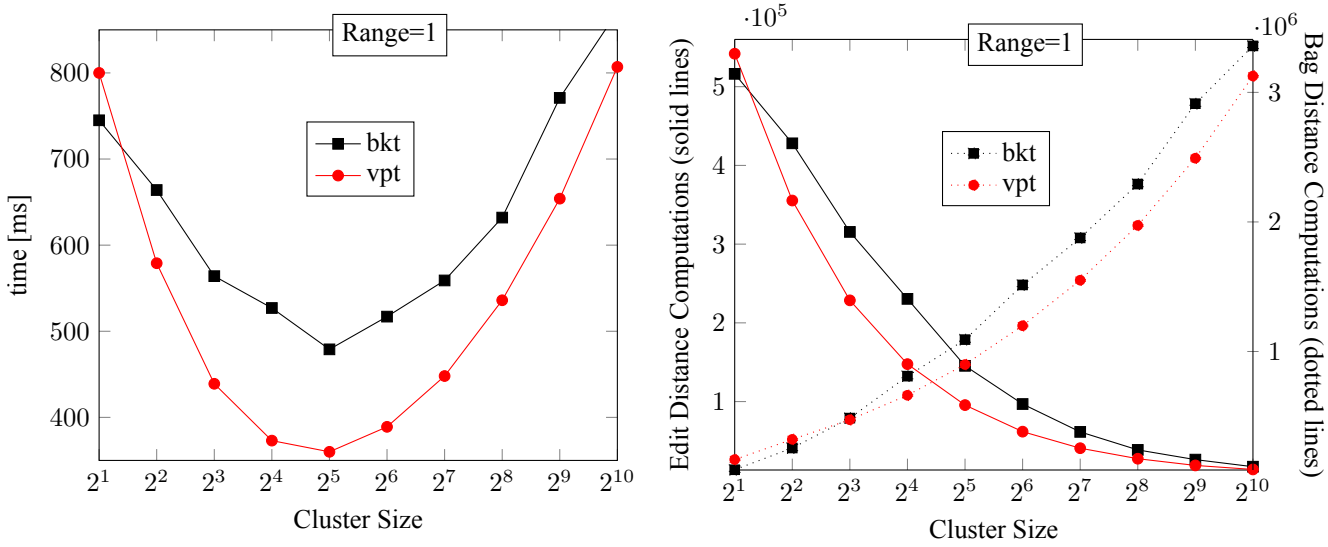


Figure 9. Performance of BKT and VPT answering range queries with $d = 1$ with variable cluster sizes

strategy	filtering	query	cs	bag distances	edit distances	time[ms]	savings
BKT	with	range-1	32	1,091,679	145,414	479	39%
BKT	without	range-1	1	0	559,022	783	
VPT	with	range-1	32	899,332	95,555	360	62%
VPT	without	range-1	4	0	675,227	945	
BKT	with	range-2	128	9,878,579	398,423	3,047	62%
BKT	without	range-2	1	0	5,659,316	7,923	
VPT	with	range-2	64	9,880,634	416,718	3,073	73%
VPT	without	range-2	8	0	8,091,596	11,328	
BKT	with	range-3	256	19,668,196	1,025,333	6,392	67%
BKT	without	range-3	1	0	14,046,396	19,665	
VPT	with	range-3	128	21,912,599	1,000,368	6,922	75%
VPT	without	range-3	16	0	19,684,548	27,558	

Table 3. The run-time implications of using the Bag Distance Filtering for searches with varying ranges.

	avg clusters size		edit distances		bag distances		time to search[ms]	
sparsity value(dp)	BKT	VPT	BKT	VPT	BKT	VPT	BKT	VPT
0	9	20	145,414	95,555	1,091,679	899,332	479	360
1	9	20	145,411	95,547	1,091,694	899,368	479	360
2	9	20	141,749	92,463	1,126,023	93,7199	482	366
3	10	22	131,102	82,422	1,338,622	1,205,645	521	419
4	11	25	113,895	66,653	1,915,367	1,826,782	642	554
5	12	30	96,781	54,272	3,159,870	2,886,307	932	803
6	15	39	72,509	40,713	4,852,233	6,603,336	1,324	1,721
7	19	52	48,064	28,485	6,661,166	8,094,252	1,746	2,080
8	25	74	28,126	19,377	8,641,524	17,732,148	2,217	4,496
9	35	112	18,318	13,412	10,031,150	19,855,847	2,553	5,022
10	50	187	12,737	9,316	10,867,343	21,204,969	2,756	5,357

Table 4. Performance of BKT and VPT answering range-1 queries with 2^5 as the cluster size and a variable sparsity parameter (dp).

strategy	filtering	query	cs	bag distances	edit distances	time[ms]	savings
BKT	with	top-1	512	22,501,551	428,206	6,270	65%
BKT	without	top-1	1	0	12904916	18067	
VPT	with	top-1	128	25,980,943	1,018,298	7973	77%
VPT	without	top-1	32	0	25,182,564	35,256	
BKT	with	top-5	512	33,779,608	988,092	9,896	69%
BKT	without	top-5	1	0	22,845,512	31,984	
VPT	with	top-5	256	32,061,814	1,898,893	10,738	75%
VPT	without	top-5	32	0	30234898	42,329	
BKT	with	top-10	1024	36,536,033	1,200,562	10888	69%
BKT	without	top-10	1	0	25,432,347	35,605	
VPT	with	top-10	256	33,798,410	2,647,826	12,224	23%
VPT	without	top-10	32	0	32,086,578	44,921	

Table 5. The Run-time Implications of the Bag Distance Filtering for Searches with a Varying Top-k.

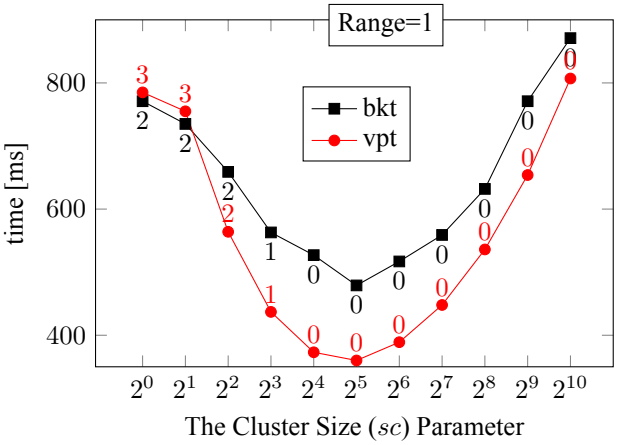


Figure 10. Performance of BKT and VPT answering range-1 queries with a variable cluster size. The label over the marks indicates the best *dp* value.

Using the *dp* partitioning rule: Figure 12 analyzes how the usage of the *dp* parameter affects the run-time. For each cluster size, the graph shows the *dp* value that achieves the best performance. Recall that this parameter acts as a way to prevent over-partitioning the space.

The figure suggests that top-1 queries benefit from larger clusters when compared to the results found when using range-1 queries. In other words, queries with a high search radius (like top-1 queries) are more efficient in settings where the partitions are not too small, and a higher *dp* value helps achieve a more suited arrangement, especially for BKT.

It is also worth noting that the best VPT results do not benefit from the clusters density ($dp = 0$). On the other hand, BKT relies on the *dp* factor to achieve higher efficiency for all defined cluster sizes. This happens because BKT clusters are smaller than the VPT clusters (as presented in Section 4.1). Hence, the *dp* parameter is more important as a partitioning stopper to BKT than it is to VPT. However, it works best when used along with a proper *cs* value.

Figure 13 reports the results achieved when using the best cluster size for BKT(2^9), VPT(2^7) and a variable distance to pivot. Corroborating our findings, the VPT does not benefit from the *dp* factor. On the other hand, the density is important to stop partitioning the space when using BKT. The results show that there is an optional *dp* value (4). The run-time is approximately 10% lower than the BKT solution that does not use the *dp* value and approximately 30% lower than the best VPT solution.

To summarize the importance of the *dp* parameter for BKT when answering top-k queries, Table 6 compares the best settings with the density value ($dp > 0$) and without the density value ($dp = 0$). Observe that the usage of the density parameter to partition the space represent savings in the elapsed time when compared to settings that do partition based on *dp*.

When analyzing the strategies that rely on *dp*, it is also important to notice that, as k grows, it is better to use a higher *dp* value, which leads to larger clusters. Curiously, when compared to top-1, the top-5, and top-10 queries use smaller clusters. This demonstrates that the density is also an important feature when defining the clusters formation. The higher external complexity of working with a lower cluster size is counterbalanced by the lower internal complexity of working with denser clusters. This translates into a more efficient setting, at least for top-k queries.

5 Concluding Remarks

This paper investigates how the density and size of clusters impact the performance of string similarity search when bag distance filtering is used over in-memory metric trees.

To enable a consistent evaluation, two metric trees were extended: BKT and VPT. Our extended VPT brought stability to the index construction when the cluster size is small. This is an important fix since it enabled a more consistent comparison that includes very small clusters. On the other hand, our extended BKT allows clusters to be formed, contrasting with the original solution that supports a single object per node. This proves to be beneficial both in index construction and search.

Two types of queries were tested: fixed-range queries and top-k queries. The latter tends to be more expensive, since the search radius necessarily starts with a high value, leading the search to access more objects.

Our evaluation confirmed the expectations that clusters have a direct impact on performance. Also, the best cluster formation is highly dependent on the search radius. The performance of the fixed search radius deteriorates when *cs* is greater than 32. On the other hand, top-1 queries are more resilient concerning larger cluster sizes. Since more objects

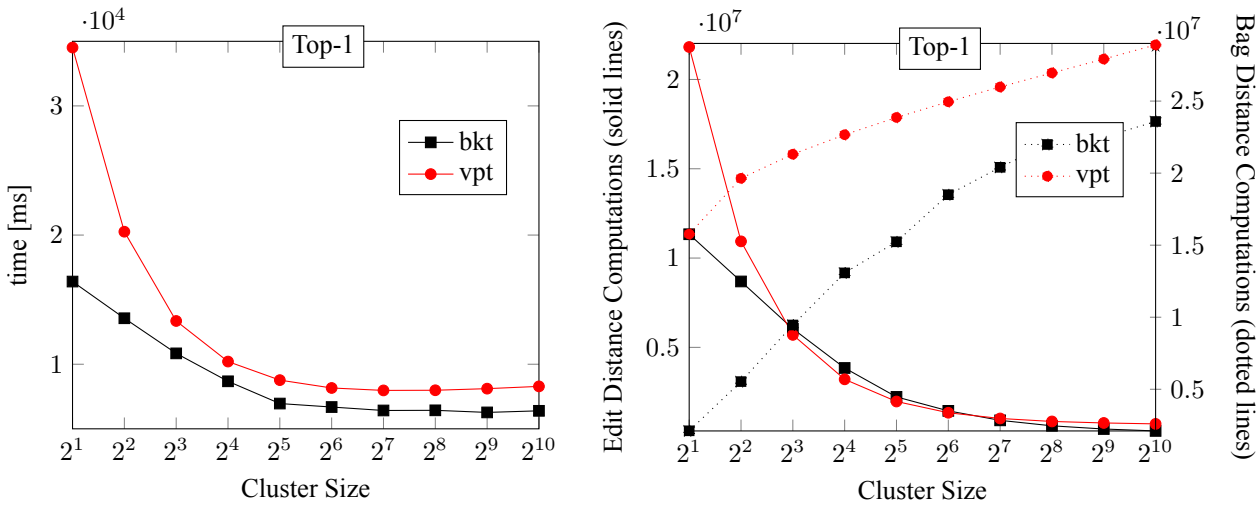


Figure 11. Performance of BKT and VPT answering top-1 queries with variable cluster sizes.

strategy	dp	cs	average cluster size	bag distances	edit distances	time[ms]	savings
top-1	0	512	106	22501551	428206	6270	8,50%
top-1	4	512	114	20609031	388072	5737	
top-5	0	512	106	33779608	988092	9896	9,98%
top-5	5	256	68	29621981	1031166	8908	
top-10	0	1024	196	36536033	1200562	10888	9,63%
top-10	5	256	68	31160065	1419318	9839	

Table 6. The Implications of the *dp* Partitioning Rule for Searches with Varying Top-k using BKT

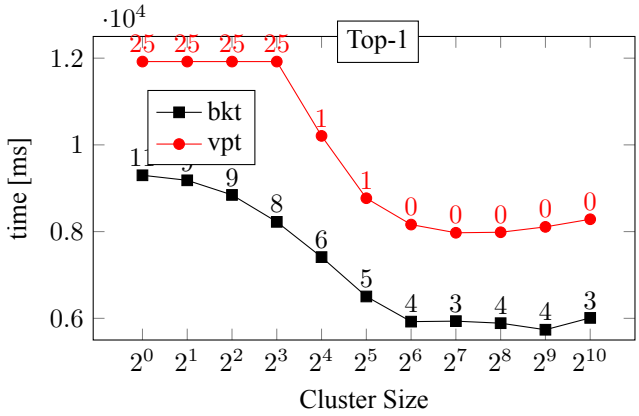


Figure 12. Performance of BKT and VPT answering top-1 queries with variable cluster sizes. The label over the mark indicates the *dp* value that achieved the best results.

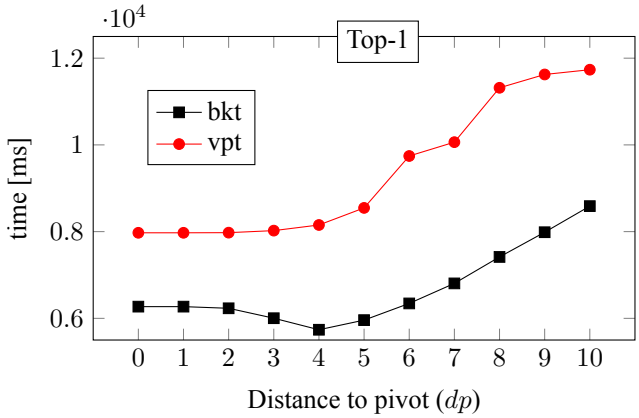


Figure 13. Performance of BKT and VPT answering top-1 queries with a variable distance to pivot (*dp*).

end up being accessed, the larger clusters reduce the internal complexity cost.

We have seen that the density (measured by the distance of the objects to their respective pivot) becomes valuable information for top-k queries. The usage of a proper *dp* value helps to stop clusters to be over-partitioned. The benefit appears for BKT since VPT trees tend to form larger clusters by nature. The results show that BKT is approximately 10% faster when *dp* is used. This is a significant result, considering that top-k queries tend to be more expansive than fixed-range queries. We would like to highlight the existence of optimized strategies designed for executing top-k queries, which function effectively under certain assumptions. One such assumption is the ability to return results as they are discovered, which is particularly valuable for partial sorts Hjaltason and Samet [1999]. In future work, we plan to investigate and delve deeper into these specific scenarios.

To conclude, the experiments suggest that the filtering is more suited to in-memory metric trees than to secondary storage since the size can be set to a proper value, unlike non-volatile mechanisms that must conform to a fixed size. This conclusion was drawn from experiments conducted in the string domain. However, our findings can prove beneficial for other domains too. This possibility arises when there are available cost-effective functions that serve as lower bounds for the distance metric used to construct the metric space.

We also note that the bag distance is by itself a metric Deza and Deza [2009]. Hence, instead of using the bag distance just for filtering, we can create a metric space defined in terms of the bag distance, and leave the edit distance just for the validation phase. The possible impact of the metric space transformation on string similarity search is a question

that deserves further investigation.

Competing interests

The author declares that he has no competing interests.

Availability of data and materials

The datasets analyzed during the current study are available at `https://github.com/mergen-sergio/similarity-search/blob/main/datasets/jidm_dataset_2023`.

References

Bartolini, I., Ciaccia, P., and Patella, M. (2002). String matching with metric trees using an approximate distance. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, SPIRE 2002, pages 271–283, London, UK, UK. Springer-Verlag.

Blumenthal, L. M. (1954). *Theory and applications of distance geometry*. Clarendon Press - Oxford.

Bozkaya, T. and Ozsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD Record*, volume 26, pages 357–368. ACM.

Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

Chen, L., Gao, Y., Li, X., Jensen, C. S., and Chen, G. (2015). Efficient metric indexing for similarity search. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 591–602. IEEE.

Chen, L., Gao, Y., Zheng, B., Jensen, C. S., Yang, H., and Yang, K. (2017). Pivot-based metric indexing. *Proceedings of the VLDB Endowment*, 10(10):1058–1069.

Deng, D., Li, G., Wen, H., Jagadish, H., and Feng, J. (2016). Meta: an efficient matching-based method for error-tolerant autocompletion. *Proceedings of the VLDB Endowment*, 9(10):828–839.

Deza, M. M. and Deza, E. (2009). Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer.

Figueroa, K., Navarro, G., and Chávez, E. (2007). Metric spaces library. Available at `http://www.sisap.org/Metric_Space_Library.html`.

Hjaltason, G. R. and Samet, H. (1999). Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318. DOI: 10.1145/320248.320255.

Jensen, A. H., Lauridsen, F. A., Zardbani, F., Idreos, S., and Karras, P. (2021). Revisiting multidimensional adaptive indexing. *EDBT, pages 469ś474*.

Kahveci, T. and Singh, A. K. (2001). An efficient index structure for string databases. In *VLDB*, volume 1, pages 351–360.

Lee, H., Ng, R. T., and Shim, K. (2007). Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proceedings of the 33rd international conference on Very large data bases*, pages 195–206. VLDB Endowment.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

Li, H., Ni, B., Wong, M.-H., and Leung, K.-S. (2011). A fast cuda implementation of agrep algorithm for approximate nucleotide sequence matching. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 74–77. IEEE.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453.

Papamichail, D. and Papamichail, G. (2009). Improved algorithms for approximate string matching. *BMC bioinformatics*, 10(1):S10.

Sprenger, S., Schäfer, P., and Leser, U. (2019). Bb-tree: A main-memory index structure for multidimensional range queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1566–1569. DOI: 10.1109/ICDE.2019.00143.

Traina, C., Traina, A., Faloutsos, C., and Seeger, B. (2002). Fast indexing and visualization of metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260.

Traina Jr, C., Traina, A. J., Vieira, M. R., Faloutsos, C., *et al.* (2007). The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *The VLDB Journal—The International Journal on Very Large Data Bases*, 16(4):483–505.

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. (2006). *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media.

Zhu, Y., Chen, L., Gao, Y., and Jensen, C. S. (2022). Pivot selection algorithms in metric spaces: a survey and experimental study. *The VLDB Journal*, pages 1–25.