Adaptive Indexing in High-Dimensional Metric Spaces

Konstantinos Lampropoulos
University of Ioannina, Greece

Fatemeh Zardbani
Aarhus University, Denmark

Nikos Mamoulis
University of Ioannina, Greece

Panagiotis Karras
Aarhus University, Denmark

ABSTRACT

Similarity search in high-dimensional metric spaces is routinely used in many applications including content-based image retrieval, bioinformatics, data mining, and recommender systems. Search can be accelerated by the use of an index. However, constructing a high-dimensional index can be quite expensive and may not pay off if the number of queries against the data is not large. In these circumstances, it is beneficial to construct an index *adaptively*, while responding to a query workload. Existing work on multi-dimensional adaptive indexing creates rectilinear space units by hyperplane-based partitioning. This approach, however, is highly ineffective in high-dimensional spaces. In this paper, we propose AV-tree: an alternative method for adaptive high-dimensional indexing that exploits previously computed distances, using query centers as vantage points. Our experimental study shows that AV-tree yields cumulative cost for the first several hundred or even thousand queries much lower than that of pre-built indices. After thousands of queries, the per-query performance of the AV-tree converges or even surpasses that of the state-of-the-art MVP-tree. Arguably, our approach is commendable in environments where the expected number of queries is not large while there is a need to start answering queries as soon as possible, such as applications where data are updated frequently and past data soon become obsolete.

PVLDB Reference Format:
Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. Adaptive Indexing in High-Dimensional Metric Spaces. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at github.com/DinosL/AVtree.

1 INTRODUCTION

Let O be a set of objects in a (high-dimensional) metric space. Given a query object q, a distance bound ϵ, and a distance metric $d()$, a *range similarity query* seeks the objects $o \in O$ for which $d(q, o) \leq \epsilon$. Similarly, given a positive integer k, a *k-nearest-neyighbor (kNN) similarity query* seeks k objects $o \in O$ having smaller $d(q, o)$ than all other objects in O. Range and kNN similarity queries are routinely used in similarity-based search and data mining tasks (e.g., clustering and NN classification) for application domains including computer vision [42], information retrieval [10], kNN search in spatial networks [1], and recommender systems [32].

Motivation We consider applications where data in a metric space are short-lived and a relatively small number of queries is expected before the data become obsolete. For example, satellite images that depict weather phenomena or other transient information are periodically received and automatically converted to feature vectors appropriate for similarity computations. Data scientists perform similarity search against the image collection to detect abnormal or alerting phenomena. Such images become obsolete when the next batch arrives, hence the number of queries applied on one batch is not expected to be large. In such environments, building an index prior to query processing for each batch of data is costly and may thus not be worthwhile. Instead, one may evaluate each query directly on the raw feature vectors by linear scan. However, doing so, we do not take advantage of previous queries in the processing of subsequent ones. *Adaptive indexing* [24] does exploit the work done for each query with the aim to reduce cost of subsequent ones. Introduced as *database cracking* [18, 26, 43, 44], while anticipated as *deferred data structuring* [48], adaptive indexing also appeared as *progressive merging* [16], leading to hybrid versions [21, 22, 27]. All aforementioned adaptive indexing methods apply on a single database column. Some preliminary efforts extend them to multidimensional spaces [23, 39, 40], yet are focused on low-dimensional spatial data and employ hyperplane-partitioning, which (i) does not scale well to high-dimensional spaces, and (ii) is inapplicable to generic metric spaces. Besides, they perform indexing only in response to range queries and do not cater to kNN queries, the most popular query type in high-dimensional spaces. In this paper, we develop adaptive indexing methods for generic high-dimensional metric spaces that eschew hyperplanes partitioning and cater to kNN queries.

Methodology Given the modern size of memories and the fact that we target applications where the data are short-lived, hence not voluminous, we assume that the data are stored in memory, like the majority of previous work in adaptive indexing [18, 24, 39]. For example, a collection O of objects in a D-dimensional vector space can be stored in a data array as a sequence of *feature vectors* $\langle o_{id}, o_1, o_2, o_3, \ldots, o_D \rangle$, where o_{id} is the identifier of object $o \in O$ and o_i is the value of the object in the i-th dimension (feature). Let (q_1, ϵ_1) be the first (range) query. While linearly scanning the data array to derive query results, we conduct object *swaps* to *crack* the array in two pieces: one piece containing all objects that are query results and another all remaining objects. At the same time, we initialize an *adaptive vantage* tree (AV-tree) with (q_1, ϵ_1) as root. As new queries arrive, we compare them to past queries using

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the AV-tree, and search only the parts of the array that may contain query results. Guided by the triangle inequality, we avoid accessing irrelevant fragments of the array, while introducing cracks and tree nodes corresponding to new queries. To prevent the tree from becoming excessively large, which would beat the purpose of the index, we abide by a threshold θ for cracked pieces; if a piece has fewer elements than θ, then it is *fixed* and not cracked again. Further, we improve upon this *standard cracking* approach; we find that cracking based on the *median distance* of all data points in a piece rather than the current query bound ϵ_i results in a much better index. In addition, in *fixed* pieces (which are not further cracked), we cache previously computed distances and exploit the sort order to achieve an *early termination* of comparisons.

Figure 1 depicts the cumulative cost of the proposed AV-tree on one of the real datasets in our experiments (MNIST) having 70K 50-dimensional points. We iteratively execute 1000 range queries, whose centers q are sampled from the data and compare our AV-tree to: (i) a *linear scan* method, which exhaustively scans all objects and computes their distances to each query; (ii) the cost for building (before the first query) and using an MVP-tree [6], which is the state-of-the-art index for high-dimensional points [7]; and (iii) AKD-tree [39], the state-of-the-art multidimensional adaptive index. Notably, the AV-tree exhibits the desired behavior of an ideal adaptive index: (i) it becomes much faster than linear scan even after a few queries; (ii) its cumulative cost converges to that of the MVP-tree after a few thousands of queries and does not become worse thereafter; and (iii) it is consistently faster than the AKD-tree.

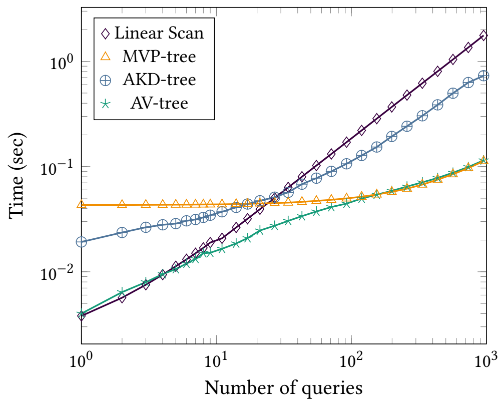


Figure 1: Cumulative cost on a workload of range queries

Besides being much more efficient than previous multidimensional adaptive indexes, our AV-tree is the first multidimensional adaptive index that supports kNN queries. We emphasize that distance-based range and kNN queries are the most general and most common operations in high-dimensional metric spaces with numerous applications [7]. The AV-tree is not only applicable in vector spaces where, for example, an L_p-norm distance (e.g., Euclidean distance) is used, but also in general metric spaces; e.g., for indexing a collection O of strings to support similarity search based on edit distance. Our experimental evaluation demonstrates the robustness of AV-tree to different metric spaces and distances.

Our contributions can be summarized as follows:

- We investigate, for the first time, the problem of building a *distance-based* adaptive index for high-dimensional metric spaces
- We define the AV-tree, an index that efficiently adapts to the query workload, forming a *unified* solution for both distance range queries and kNN queries.
- We provide several enhancements on the AV-tree.
- We conduct an extensive experimental study, showing that the AV-tree behaves as an ideal adaptive index should.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides core definitions. Our AV-tree index is presented in Section 4. Section 5 presents our experimental study and Section 6 concludes the paper and discusses future work.

2 RELATED WORK

Our work relates to metric space indexing and to adaptive indexing in databases. In this section, we review related work, focusing on the state-of-the-art.

2.1 Indexing metric spaces

Indexing high-dimensional spaces is a hard problem for two main reasons. First, due to the curse of dimensionality [5], if data points are uniformly distributed, the probability that two points are too close or too far from each other is very low, rendering similarity search mostly meaningless. Still, in many real applications, data typically form clusters, so this predicament does not apply. Second, indexes that divide the data space using rectilinear partitions (e.g., the R-tree [17], the KD-tree [4], etc.) do not perform well, as they necessarily use only a limited number of dimensions[1], hence partitions end up spanning the entire domain on most dimensions and do not separate the objects well. Besides, such indexes are only applicable in vector spaces and are mostly suited for rectilinear range queries rather than distance-based search.

Ref. [7] (see also [8]) is a recent comprehensive survey of existing indexes for exact similarity search in metric spaces. Based on this study, *pivot-based* indexes are the most effective ones. These methods select few *vantage* points (a.k.a. *pivots*, *landmarks*, *representatives*), partition the data space based on them, and use the vantage points to prune the search space, guided by the triangle inequality. Pivot-based indexes are applicable even when distances are not computed using point coordinates, but in arbitrary metric spaces (e.g., as shortest paths in graphs [2]). Besides being very efficient, pivot-based indexes provide exact and explainable results to similarity queries, which is imperative in application domains like public safety [7], bioinformatics [12], and computer forensics [33]. Hence, performing exact search in the original metric space may be preferable over solutions that transform the data to a vector space using machine-learning techniques (e.g., embedding approaches [37]) and apply search in the transformed space or LSH-based approximate indexes [14, 15, 35, 46]; the latter are mostly appropriate in spaces where objects are not well-separated, thus exact similarity search may not be meaningful or critical.

We present in detail three representative main-memory pivot-based metric indexes that we use as competitors to our approach.

[1]Partitioning a D-dimensional space at least once in each dimension defines 2^D partitions, which are much more than the data points for large D (e.g., $D = 100$).

2.1.1 SimplePivot. SimplePivot employs Farthest First Traversal (FFT) [20] to choose vantage points. The FFT algorithm starts by selecting a random point as the first pivot. In each subsequent iteration, it selects as a pivot the object u that maximizes $\min_{p \in P} d(u, p)$, where P is the set of previously selected pivots. Thereafter, we compute the distances of each data point to all vantage points. When evaluating queries, we use the pre-computed distances to avoid unnecessary distance computations. Specifically, consider a query point q and radius ϵ. At the beginning of query evaluation, we compute and cache $d(p_i, q)$ for all p_i. For each data object $o \in S$, if there exists a pivot p_i, such that $|d(p_i, o) - d(p_i, q)| > \epsilon$, then o is certainly not a result, hence we do not need calculate $dist(q, o)$. Since $d(p_i, o)$ has been precomputed, the pruning test for each object o takes $O(m)$ time, where m is the number of pivots, whereas the computation of distance $d(q, o)$ takes $O(D)$ time for L_p distances in D-dimensional vector spaces and even more in other metric spaces (e.g., edit distance).

2.1.2 Spatial Approximation Tree. The *Spatial Approximation Tree* (SAT) [38] is a hierarchical data structure, where the children of a node are its neighbors in the Delaunay graph on the entire data set. To find the nearest neighbor (NN) of a query object q, we start from the root n; if n is closer to q than its children, then search stops reporting n as the NN. Otherwise, we navigate to the child of n nearest to q and search recursively. To evaluate a range query (q, ϵ), SAT uses the triangle inequality to prune nodes (and corresponding sub-trees) that are guaranteed to be further than ϵ from q.

2.1.3 VP-tree and MVP-tree. The *Vantage Point tree* (VP-tree) [47] partitions the data hierarchically based on a *vantage point* at each node. Starting with the root, which indexes all data, at each node v, a vantage object (pivot) p_v is selected at random from the objects indexed at this node. The data under node v are then split in two partitions as follows: Let μ be the mean distance of points under v to pivot p_v. Objects having distance to p_v less than or equal to μ are placed in the *left* sub-tree; remaining objects go to the *right* sub-tree. To evaluate a range query, we recursively traverse the VP-tree. For a query (q, ϵ), at each node v with pivot p_v having median distance from its indexed points μ, we examine the following:

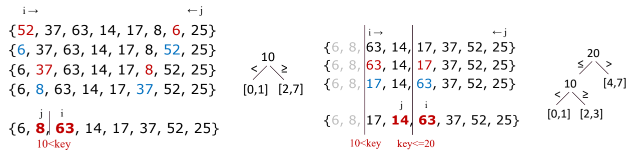
- if $dist(q, p_v) \le \epsilon$, then p_v is a result;
- if $|dist(q, p_v) - \mu| \le \epsilon$, then search the *left* sub-tree;
- if $dist(q, p_v) + \epsilon > \mu$, then search the *right* sub-tree.

For a single query, we may follow multiple paths of the VP-tree, according to the above. The MVP-tree [6] generalizes the VP-tree to a m-ary tree. Instead of splitting the objects in two partitions, it orders them by their distance to the vantage point and partitions them to m groups of equal cardinality. During search, it uses the mean distance for each of the m groups to prune subtrees that cannot include query results. According to the extensive experimental study in [7], the MVP-tree performs best compared to a wide-range of main-memory metric space indexes, including pivot-based methods [13, 29] and SAT [38]. On the other hand, notable metric-space indexes optimized for secondary memory are the M-tree [9] and the PM-tree [45]

The AV-tree we propose differs from VP-tree variants in the following ways: (i) it builds the tree *progressively*, via query evaluation, rather than in advance; (ii) it selects pivots *adaptively* from queries rather than from the data collection; and (iii) it keeps the data in a *single array* and swaps them to partition them to sub-arrays.

2.2 Adaptive indexing

Adaptive indexing constructs a data structure for a static dataset on demand by adapting to an evaluated query workload [31]. Each query divides the data space based on its results, triggering the construction of a search tree that guide the evaluation of subsequent queries. This idea has been applied to database indexing [24], progressively *cracking* an initially unsorted array to segments that obey a total order and constructing a binary search tree to prune sub-arrays that do not contain query results. Figure 2 shows the cracking of an unsorted array based on query $10 < x \le 20$ and one step in the progressive construction of the corresponding binary search tree. The array is first cracked based on $10 < x$. Indices i and j scan the array in forward and backward, respectively. For each out-of-order value found (i.e., a value > 10 at i and one ≤ 10 at j), if $i < j$, the values are swapped and the process continues; otherwise, cracking stops. After cracking based on $10 < x$, we update the binary search tree: all values less than 10 are in array positions 0 to 1 and all values of 10 or larger are in array positions 2 to 7. The second crack, based on $x \le 20$ applies on the right child of the root and cracks the corresponding subarray to two pieces: positions 2 to 3 having keys less than or equal to 20 and positions 4 to 7, having keys greater than 20. The search tree (e.g., an AVL-tree) is re-balanced after each cracking.



(a) cracking based on $10 < x$ (b) cracking based on $x \le 20$

Figure 2: Standard cracking example

In effect, cracking conducts *quicksort* incrementally, triggered by queries. This process has been extended to efficiently handle updates efficiently [25]. As an alternative to quicksort, one may perform a *mergesort* operation incrementally [16], a hybrid approach that combines merging and cracking [27], or first partition the domain in disjoint ranges and then crack the partitions [44].

Like typical database indexes do, cracking applies on a single attribute [26]. Adaptive indexing for multidimensional spaces has been explored in the context of spatial or low-dimensional databases. QUASII [40] builds a tree of D levels, where D is the data dimensionality. It first projects each query on the x dimension and performs cracking on it; then it cracks the formed x-dimension pieces on the y dimension, and so on. AKD-tree [23] extends the classic kd-tree [4] to a multidimensional adaptive index, which progressively partitions the data by hyperplanes guided by queries and constructs a kd-tree to index the enusing pieces. The latter was shown to outperform QUASII [30, 39]. However, as we will show experimentally, distance-based partitioning fares much better than hyperplane-based partitioning.

3 DEFINITIONS AND PRELIMINARIES

We examine similarity-based queries in a metric space. A metric space is defined as a pair $\{M, d\}$. M is a domain wherefrom objects are instantiated. For example, if M is be a D-dimensional vector space, each object o in it has the form $\langle id, o_1, o_2, \ldots, o_D \rangle$, where id is an identifier and $o_i \in [0, 1]$ is the i-th dimensional value of o; d is a metric distance function applied between objects in M; in vector spaces, d is typically the Euclidean distance, $d(q, o) = \sqrt{\sum_{i=1}^{D}(q_i - o_i)^2}$. In general, a metric distance function d has the following four properties:

- **Identity.** The distance of an object to itself is 0; $d(x, x) = 0$.
- **Non-negativity.** The distance between two distinct objects is positive; if $x \neq y$ then $d(x, y) > 0$.
- **Symmetry.** The distance from x to y is the same as that from y to x; $d(x, y) = d(y, x)$.
- **Triangle Inequality.** For any three objects x, y, z, $d(x, z) \leq d(x, y) + d(y, z)$.

We consider a set O of objects in the metric space M, which may become available in batches. The most common similarity-based queries are the range query and the k-nearest-neighbor query.

Definition 1. **Range Query.** *Given an object q and a distance bound ϵ, a range query returns all objects $o \in O$ that are within distance ϵ from q, i.e., $d(q, o) \leq \epsilon$.*

Definition 2. **k-Nearest-Neighbor (kNN) Query.** *Given an object q and a positive integer k, $k \leq |O|$, a k-nearest-neighbor query returns a subset $R \subseteq O$, such that $|R| = k$ and $\forall o \in R, o' \in O \setminus R : d(q, o) \leq d(q, o')$; in other words, a kNN query finds a set R of k objects in O having no larger distances to q than those outside R.*

Answering a query amounts to finding the identifiers of the objects $o \in O$ in the result set. We assume that the objects are stored in rows in memory, i.e., the entire tuple of the first object precedes the tuple of the second object, and so on; such a representation facilitates efficient distance computations.

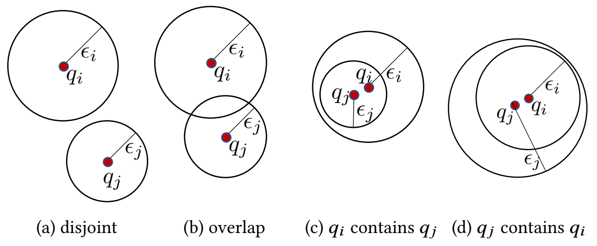


Figure 3: Four cases of overlap between q_i and q_j

We aim to exploit previously evaluated queries to expedite the evaluation of subsequent queries. To do so, we should quickly determine whether, and how, the range of a previous query (q_i, ϵ_i) overlaps with that of the current query (q_j, ϵ_j). There are four cases in this regard, depicted in Figure 3.

(a) (q_i, ϵ_i) does not overlap (q_j, ϵ_j), i.e., $d(q_i, q_j) > \epsilon_i + \epsilon_j$.
(b) (q_i, ϵ_i) overlaps (q_j, ϵ_j), but neither is a subset of the other.
(c) (q_i, ϵ_i) contains (q_j, ϵ_j), i.e., $\epsilon_i \geq d(q_i, q_j) + \epsilon_j$.
(d) (q_j, ϵ_j) contains (q_i, ϵ_i), i.e., $\epsilon_j \geq d(q_i, q_j) + \epsilon_i$.

Exploiting the triangle inequality, we can identify the relationship between two queries as follows. We first check whether $d(q_i, q_j) > \epsilon_i + \epsilon_j$. If this holds, then the query ranges do not overlap, i.e., case (a) holds. Otherwise, we have one of the remaining three cases, which we test as follows. If $d(q_i, q_j) \leq \epsilon_i - \epsilon_j$ then q_i contains q_j, i.e., case (c). If $d(q_i, q_j) \leq \epsilon_j - \epsilon_i$ then q_j contains q_i, i.e., case (d). In all other cases (i.e., $d(q_i, q_j) > |\epsilon_i - \epsilon_j|$), we have case (b).

4 THE AV-TREE

In this section, we present our proposed *adaptive vantage* tree (AV-tree) for high-dimensional metric spaces. We first show how the AV-tree is incrementally constructed while evaluating range queries and then present the corresponding algorithm for kNN queries. Notably, range queries can be interleaved with kNN queries in a mixed workload without affecting the data structure and its effectiveness. In Section 4.3, we present some enhancements to the basic version of our index, followed by a cost analysis (Sec. 4.4).

4.1 Range Query

Our algorithm builds on the framework for one-dimensional database cracking with some significant departures: First, there is no total order of the indexed data to guide the process. Second, contrary to existing multidimensional cracking approaches [39, 40], we do not partition the space by hyperplanes, but based on the distances between the query and data.

We evaluate the first query (q_1, ϵ_1) by scanning the entire data array O, and, while computing the results, performing a *crack-in-two* operation: we place data points $o \in O$ with $d(q_1, o) \leq \epsilon$ before data points $o \in O$ with $d(q_1, o) > \epsilon$. We heed no other order constraints. At the same time, we define the root of the *adaptive vantage* tree, or AV-tree, a binary search tree which helps identify relevant data for subsequent queries and avoid redundant computations. For each subsequent query, we use the AV-tree to guide search and expand it by introducing new cracks.

Each node v in the AV-tree contains two elements: the *scope* $[v.lo, v.hi]$ of v, i.e., the range of array indices that v indexes; and the *query* $(v.q, v.\epsilon)$, that guides the search in v, if v is not a leaf node. The tree root has $lo = 0$ and $hi = |O| - 1$, where $|O|$ is number of data objects. If v is a leaf, then $v.q$ is null. Otherwise, v has two pointers v.left, v.right to its left and right children, respectively. For each object o in the scope of v.left, it is $d(q, o) \leq v.\epsilon$, while for each object o in the scope of v.right, it is $d(q, o) > v.\epsilon$.

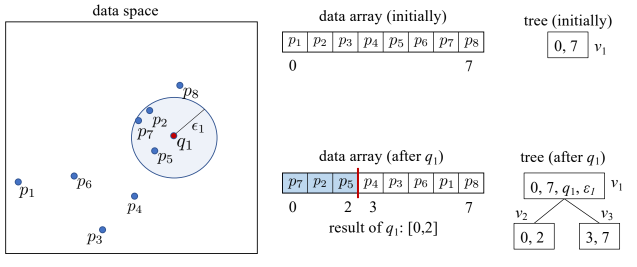
Algorithm 1 presents the search-and-crack process in detail, outlined in two procedures. The main recursive SEARCH-AND-CRACK procedure takes as input an array O with the data points, a query point q and the corresponding distance bound ϵ and the node v of the AV-tree on which it is applied. For a new query, we initialize the query result to $R = \emptyset$ and call the procedure with v being the tree root. If $v.q$ is null, then v is a leaf node, hence we crack by procedure CRACK-IN-TWO (described later), yielding two new vertices as children of v. If node v is not a leaf, then we examine the relationship between the node query range $(v.q, v.\epsilon)$ and the new query range (q, ϵ) as in Section 3. If the two ranges are disjoint, all data under the scope of v.left are not part of the query result, hence we call SEARCH-AND-CRACK for the right child $v.right$, as its scope may include results of q. On the other hand, if the new query range overlaps with $(v.q, v.\epsilon)$, we distinguish two cases. If $(v.q, v.\epsilon)$ is entirely inside (q, ϵ), then we add[2] to R all data under the scope of v.left as query results and SEARCH-AND-CRACK the right subtree of v. Otherwise, if (q, ϵ) is entirely inside $(v.q, v.\epsilon)$, we only SEARCH-AND-CRACK the left subtree of v. Lastly, if there is no containment relationship between $(v.q, v.\epsilon)$ and (q, ϵ), as in Figure 3b, then we also call SEARCH-AND-CRACK for the right subtree.

Algorithm 1 Distance-Range Search and Crack

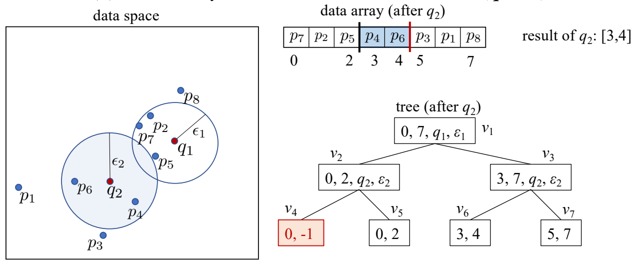
```
1:  procedure SEARCH-AND-CRACK(data array $O$, query $q$, bound $\epsilon$, node $v$, result $R$)
2:      if $v.q$ is null then                                          ▷ leaf node
3:          ($v$.left, $v$.right) ←CRACK-IN-TWO($O, v.lo, v.hi, q, \epsilon, R$)
4:          $v.q \leftarrow q$; $v.\epsilon \leftarrow \epsilon$
5:      else                                                           ▷ non-leaf node
6:          if $d(q, v.q) > \epsilon + v.\epsilon$ then                 ▷ disjoint query ranges
7:              SEARCH-AND-CRACK($O, q, \epsilon, v$.right$, R$)
8:          else                                                       ▷ overlapping query ranges
9:              if $d(q, v.q) < \epsilon - v.\epsilon$ then             ▷ $v.q$ entirely inside $q$
10:                 $R \leftarrow R \cup [v.\text{left}.lo, v.\text{left}.hi]$      ▷ update query result
11:                 SEARCH-AND-CRACK($O, q, \epsilon, v$.right$, R$)
12:             else
13:                 SEARCH-AND-CRACK($O, q, \epsilon, v$.left$, R$)
14:                 if $d(q, v.q) \geq v.\epsilon - \epsilon$ then      ▷ $q$ not entirely inside $v.q$
15:                     SEARCH-AND-CRACK($O, q, \epsilon, v$.right$, R$)
16:
17: procedure CRACK-IN-TWO(array $O$, int $lo$, int $hi$, query pt $q$, bound $\epsilon$, resullt $R$)
18:     $i \leftarrow lo$
19:     $j \leftarrow hi$
20:     while true do
21:         while $d(q, O[i]) \leq epsilon$ and $i \leq hi$ do
22:             $i \leftarrow i + 1$
23:         while $d(q, O[j]) > epsilon$ and $j \geq lo$ do
24:             $j \leftarrow j - 1$
25:         if $i \geq j$ then
26:             break
27:         swap $O[i]$ with $O[j]$                                    ▷ $O[i]$ and $O[j]$ on wrong sides
28:     $R \leftarrow R \cup [lo, j]$                                   ▷ update query result
29:     $v_L.lo \leftarrow lo$;   $v_L.hi \leftarrow j$;   $v_L.q \leftarrow$ null
30:     $v_R.lo \leftarrow j + 1$;   $v_R.hi \leftarrow hi$;   $v_R.q \leftarrow$ null
31:     return $(v_L, v_R)$
```

Procedure CRACK-IN-TWO is based on Hoare's quicksort partitioning [19]; it scans the scope of a node v array O from position lo to position hi and swaps data items to divide $[lo, hi]$ in two parts: $[lo, j]$, including data points o such that $d(q, o) \leq \epsilon$ and $[j + 1, hi]$ including remaining points. We add the former part, $[lo, j]$, to the query result R and generate two new nodes for the two new scopes, as children of calling node v. Note that one of the two scopes may be *empty*, in case the scope of v includes either (i) no query results or (ii) only query results. In the former case, $v_L.lo = lo$ and $v_L.hi = lo - 1$; in the second case, $v_R.lo = hi + 1$ and $v_L.hi = hi$. Procedure SEARCH-AND-CRACK does not perform recursive calls for a child having empty scope, as no query results can be obtained from such nodes. We call leaves that cannot be further cracked because they have an empty scope *empty* leaves.

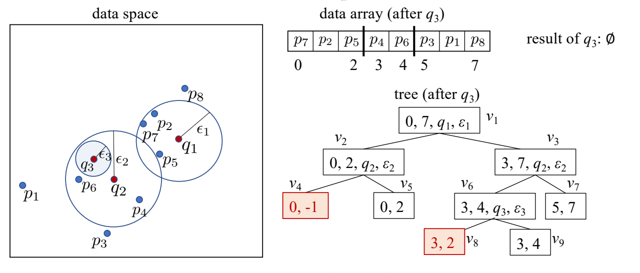
Example. Figure 4 presents a detailed example of SEARCH-AND-CRACK running. Data array O includes eight 2D points, p_1 to p_8, and initially the tree has a single node v_1 with scope $[0, 7]$. Upon the first query, (q_1, ϵ_1), CRACK-IN-TWO runs for the root node v_1 (Line 3), which is a leaf, to produce two new nodes, v_2 and v_3, as its left and right child, respectively, as Figure 4a shows. The result of R is the scope of v_2 (i.e., p_7, p_2, and p_5). For the second query, (q_2, ϵ_2), SEARCH-AND-CRACK runs for $(q, \epsilon) = (q_2, \epsilon_2)$. Figure 4b shows that the range of q_2 overlaps with the query range of the root, (q_1, ϵ_1). Hence, we enter Lines 13–15. The call in Line 13 yields a CRACK-IN-TWO of v_1's left child (i.e., node v_2). However, this crack produces no results, because all points in the scope of v_2 are outside q_2's range; thus the newly produced node v_4 as left child of v_2 has *empty range* [0,-1] (shaded in the figure). The generated right child v_5 has the same scope as its parent v_2. As we will discuss in Section 4.3.2, in our enhanced version we do not split a leaf if one of its children is empty. The recursive call at Line 15 invokes CRACK-IN-TWO for v_1's right child (i.e., v_3). Now we do have a query result [3,4] added to R (i.e., points p_4 and p_6) and two new vertices v_6 and v_7 with non-empty scopes as new children of v_3. Figure 4c shows the effect of the next query, (q_3, ϵ_3). Query range (q_3, ϵ_3) is outside the range of the root v_1, hence we only visit its right child v_3 (Line 7). We find that (q_3, ϵ_3) is fully contained in (q_2, ϵ_2), hence only visit v_3's left child v_6 (Line 13), where we find no results for q_3, yielding v_8 as an empty child of, and v_9 as identical to, v_6.



(a) data array and tree, before and after (q_1, ϵ_1)



(b) after (q_2, ϵ_2)



(c) after (q_3, ϵ_3)

Figure 4: Search-and-crack example

4.2 Nearest-Neighbor Query

Like range queries, kNN queries crack the data array and progressively construct and use AV-tree as a binary search tree that guides them to relevant data, yet now we access the AV-tree nodes in a *best-first* order that is appropriate for kNN search. Further, adaptive indexing in response to kNN queries poses a distinct challenge, as

[2]For simplicity, we denote the query result as a set of interval ranges indicating the positions of result objects in array O.

queries do not readily offer a distance bound. We define such a bound as the distance between the query object and its running k-th nearest neighbor. Algorithm 2 presents the process in detail. A query (q, k) comprises a query point q and the integer number of sought nearest neighbors k. We use two priority queues: a min-heap searchPQ that organizes unvisited nodes by least possible distance to q to guide the search in a best-first manner, initialized with the root; and a max-heap resultPQ that holds the running kNN data object results. In each iteration of the while loop (Lines 5–20), we pop the top element v from searchPQ. If v is a leaf (Line 8), we compute the distance of each data object in v to the query point q and update resultPQ accordingly, keeping track of the k nearest objects to q; resultPQ is a max-heap of data-objects ordered by their distance to the query point, hence the top element is the farthest from q, i.e., the running kth Nearest Neighbor. When looping over the data objects in a leaf (Line 9), if one is closer to q than the running kth-NN (Line 10), then we remove the current top element (Line 11), and add this object to resultPQ (Line 12). We also crack the leaf node using as bound (to be enhanced in Section 4.3) the distance to the top item in resultPQ (Line 13). If v is an internal tree node (Line 14) then we push its children to searchPQ with their priority key set as the *least possible distance* of an object under v to q, using the triangle inequality. As the left sub-tree contains points that are closer than $v.\epsilon$ to v's vantage point, q's minimum possible distance from an object therein is the maximum of 0 and distance of v's vantage point from the query minus $v.\epsilon$ (Line 15); if the latter is negative, then q is in the sphere of v. The right child, conversely, contains the objects outside the sphere centered at v's vantage point with radius $v.\epsilon$, hence q's minimum possible distance from such an object is the maximum of 0 and $v.\epsilon$ minus the distance of v's vantage point from q (Line 18); if the latter is negative, then q is outside the sphere of v. In both cases, if the minimum distance is not smaller than the running k-th smallest distance at the top of resultPQ, then we do not need to look into that child; otherwise (Lines 16 and 19), we add the node to searchPQ (Lines 17 and 20). The search terminates when searchPQ becomes empty.

Algorithm 2 kNN Search and Crack

```
1: procedure kNNSearch(data array $O$, query $q$, int $k$)
2:     searchPQ ← $PriorityQueue\langle dist, node\rangle$              ▷ guide search
3:     resultPQ ← $PriorityQueue\langle dist, pid\rangle$                ▷ results PQ
4:     searchPQ.$push([0, root])$
5:     while !searchPQ.$empty()$ and
6:             searchPQ.$top().dist$ < resultPQ.$top().dist$ do
7:         $v$ = searchPQ.$top().deheap()$
8:         if $v$ is $leaf$ then
9:             for $o$ in $v$ do                                ▷ $o$ is a data point
10:                if $d(p,q)$ < resultPQ.$top().dist$ then
11:                    resultPQ.$pop()$                          ▷ update $k$NN set
12:                    resultPQ.$push([d(p,q), pid])$
13:            Crack $v$                         ▷ on distance to current $k$-th NN
14:        else                                                  ▷ non-leaf node
15:            leftMinDist = $max\{0, d(v,q) - v.\epsilon\}$
16:            if leftMinDist < resultPQ.$top().dist$ then
17:                searchPQ.$push([\text{leftMinDist}, v.left])$
18:            rightMinDist = $max\{0, v.\epsilon - d(v,q)\}$
19:            if rightMinDist < resultPQ.$top().dist$ then
20:                searchPQ.$push([\text{rightMinDist}, v.right])$
```

4.3 Enhancements

We introduce three enhancements that significantly improve the performance of the AV-tree: First, we crack leaves by the median distance of objects therein to the query. Second, we eschew constructing empty leaves and cracking leaves that have a few objects. Third, we *cache* and sort the last computed distances of objects in leaves that cannot be cracked further, to avoid distance computations for objects that are definitely not results.

4.3.1 Cracking based on mediocre distances. A popular practice in both cracking-based and pivot-based indexing methods is to partition the data on an intrinsic *median* value rather than an extrinsic threshold. For instance, the VP-tree [47] partitions data on their *median distance* to a vantage point. Likewise, in one-dimensional cracking, using a median or *mediocre* value in a cracked piece rather than a query threshold brings efficiency benefits [48]. Inspired by such precedents, we use a *sample-based mediocre pivot* for leaf cracking. When we crack a sub-array corresponding to a leaf node v, we compute the distances from a few sample points in v to q and crack on the median thereof as $v.\epsilon$. We confirmed experimentally that, with as few as 3 samples, mediocre-based cracking is superior to the default strategy that cracks on the query range ϵ or the running kth nearest neighbor. Besides, mediocre-based cracking leads to a balanced tree, since each crack yields two partitions of almost equal size. Therefore, mediocre-based cracking improves performance and renders the index more versatile in handling different query workloads. In addition, thanks to mediocre-based cracking, we do not need to perform two passes over the data (one to update the kNN set and one to crack) with kNN-search cracking; we only sample 3 distances and then crack and update kNNs in one pass.

4.3.2 Avoiding empty leaves and applying a cracking threshold. As we saw in Section 4.1, our default algorithm may add *empty leaves* to the AV-tree. Such empty leaves add overhead in searching the tree. In our implementation, we do not create leaves with empty scopes, i.e., we do not commit a crack of a leaf v that results in an empty v_L or v_R and let v remain a leaf. In addition, the default algorithm cracks a leaf v unconditionally; however, leaf nodes that contain few objects are not worth cracking in practice, as they increase tree height without offering a significant pruning advantage in comparison to scanning objects. As in one-dimensional cracking [24], we do not crack leaf nodes holding objects no more than *cracking threshold* θ, which effectively delimits the height of the AV-tree. We thus expect average leaf size to converge to $\theta/2$ and tree height to $1 + \log_2 \frac{|O|}{\theta}$. We call such leaves that are not cracked further *fixed* leaves. When a query reaches a fixed leaf, we obtain query results therefrom by linear scan.

4.3.3 Distance Caching. As distance computations consume most of the query evaluation cost, reducing their amount would pay off in efficiency. We may do so by *caching* query-to-object distances. That is, while cracking, we can keep each object's distance to the current query. Thereby, we get the distance of each object in a leaf v to the leaf's parent node $p(v)$. Next time we visit v while processing another query q, we may use the triangle inequality on $d(p(v), q)$ and the cached distance $d(p(v), o)$ for each object $o \in v$ to check whether o can be a query result and either prune o or add it to the result set R accordingly, avoiding the associated distance computation.

Nevertheless, triangle-inequality calculations may sometimes be ineffective and thus cause an unnecessary overhead. We thus apply distance caching *conservatively*; only in *fixed* leaves, we cache the distance of each object o in leaf v to the leaf's parent node $p(v)$, and *sort* those objects by distance to $p(v)$. We utilize this sorted order to conduct only a few comparisons to cached distances with early termination and and to prune distance computations for objects for which we can directly determine whether they are query results.

Algorithm 3 Search and Cracking with Caching

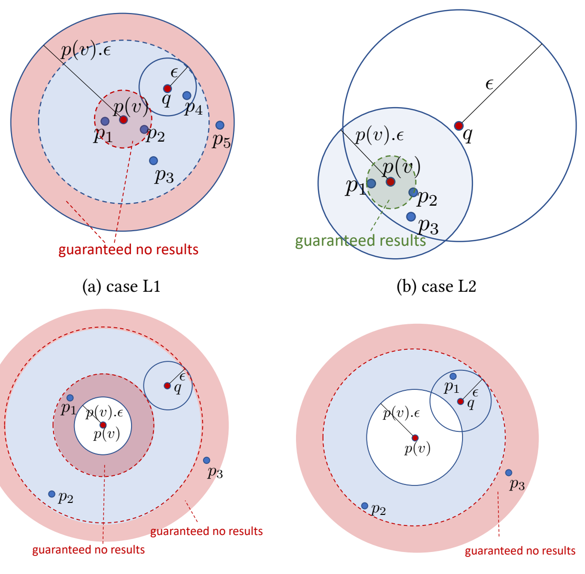
```
1: procedure SEARCH-AND-CRACK-CACHING(data array $O$, query pt $q$, bound $\epsilon$, node
   $v$, result $R$, threshold $\theta$)
2:     if $v.q$ is null then
3:         if $v.size \leq \theta$ then                                   ▷ fixed leaf node
4:             $qDist$ = distance($v$,$q$)
5:             $p(v)$ = parent of $v$
6:             if $v.isLeftChild()$ then
7:                 if $qDist > \epsilon$ then                            ▷ Case L1
8:                     $low$ = first $o \in v$, such that $d(p(v), o) \geq qDist - \epsilon$
9:                     $high$ = last $o \in v$, such that $d(p(v), o) \leq qDist + \epsilon$
10:                    for $o \in v$ from $low$ to $high$ do
11:                        if $d(q, o) \leq \epsilon$ then
12:                            $R = R \cup \{o\}$
13:                else                                                  ▷ Case L2
14:                    $low$ = first $o \in v$, s. t. $d(p(v), o) \geq \epsilon - qDist$
15:                    for $o \in v$ from start until $low$ (excl.) do
16:                        $R = R \cup \{o\}$
17:                    for $o \in v$ from $low$ until end do
18:                        if $d(q, o) \leq \epsilon$ then
19:                            $R = R \cup \{o\}$
20:            else                                         ▷ $v$ is right child of $p(v)$
21:                if $qDist \leq \epsilon + p(v).\epsilon$ then               ▷ Case R2
22:                    $low$ = first $o \in v$
23:                else                                                  ▷ Case R1
24:                    $low$ = first $o \in v$, such that $d(p(v), o) \geq qDist - \epsilon$
25:                $high$ = last $o \in v$, such that $d(p(v), o) \leq qDist + \epsilon$
26:                for $o \in v$ from $low$ to $high$ do
27:                    if $d(q, o) \leq \epsilon$ then
28:                        $R = R \cup \{o\}$
29:        else                                                   ▷ non-fixed leaf
30:            Lines 3–4 of Algorithm 1
31:            if $v.left.size \leq \theta$ then                       ▷ $v.left$ is fixed
32:                qsort($v.left$)
33:            if $v.right.size \leq \theta$ then                     ▷ $v.right$ is fixed
34:                qsort($v.right$)
35:        Lines 6–15 of Algorithm 1
```

Algorithm 3 shows how we handle fixed leaves using cached distances, modifying procedure SEARCH-AND-CRACK in Algorithm 1. We show the differing part for range queries; for kNN queries, we use the distance to the item at the top of the result heap as a threshold in comparisons in place of ϵ and update the kNN result set when accessing qualifying data objects. We now discuss in detail how we prune distance computations in each of four cases, as depicted in Figure 5: two for left-child fixed leaves v that point to data within their parent's query range $(p(v), p(v).\epsilon)$ and two for right-child fixed leaves pointing to data outside the query range.

In **Case L1**, $d(q, p(v)) > \epsilon$. Then, by the triangle inequality, objects o with $d(p(v), o) < d(q, p(v)) - \epsilon$ or $d(p(v), o) > d(q, p(v)) + \epsilon$ cannot be query results. We then find, by binary search among sorted cached distances, the position of the first object $o \in v$ with $d(p(v), o) \geq d(q, p(v)) - \epsilon$ and that of the last object $o \in v$ with $d(p(v), o) \leq d(q, p(v)) + \epsilon$, scan objects in-between, and include in the result those having distance to q at most ϵ (Lines 10–14). In Figure 5a, assume $p(v)$ is the parent of fixed leaf v, with objects p_1 to p_5 sorted by distance to $p(v)$. Objects other than p_3 and p_4 cannot be query results; we compute distances to q only for those two, yielding p_4 as a result.

In **Case L2**, $d(q, p(v)) \leq \epsilon$. Then, objects o with $d(p(v), o) \leq \epsilon - d(q, p(v))$ are surely query results. Hence, we find, by binary search, the position of the first object $o \in v$ with $d(p(v), o) > \epsilon - d(q, p(v))$ (e.g., p_3 in Figure 5b), add all objects heretofore to the query result (e.g., p_1 and p_2 in Figure 5b), and conduct distance computations only for objects thereafter (Lines 17–24).

In **Case R1**, the ranges of query q and $p(v)$ are disjoint, i.e., $d(q, p(v)) > p(v).\epsilon + \epsilon$. Then, objects o outside the query range of $p(v)$ with $d(p(v), o) < d(q, p(v)) - \epsilon$ or $d(p(v), o) > d(q, p(v)) + \epsilon$ (e.g., p_1 and p_3 in Figure 5c) cannot be query results; thus, as in Case L1, we find, by binary search, the range of candidate query results and compute distances only for those. **Case R2** applies when $d(q, p(v)) \leq p(v).\epsilon + \epsilon$ and is similar to Case R1, except that now there are no objects o with $d(p(v), o) < d(q, p(v)) - \epsilon$ outside the query range $(p(v), p(v).\epsilon)$, hence a single binary search suffices.



(a) case L1 (b) case L2 (c) case R1 (d) case R2

Figure 5: Use of cached distances at AV-tree leaves

In addition, when possible, we avoid binary search to compute the range of positions in which to scan objects and compute distances. For example, in Case L1, if $p(v).\epsilon \leq d(q, p(v))$, then there are no objects within the query range of $p(v)$ with $d(p(v), o) > d(q, p(v)) + \epsilon$, hence $high$ is the last position of v. Similarly, in Case L2, if $d(q, p(v)) + p(v).\epsilon \leq \epsilon$, then all objects within the query range of $p(v)$ are query results, hence we need not do any comparisons. We also exploit the fact that, as objects in fixed leaves are sorted, the first and the last object in v provide lower and upper bounds to $d(p(v), o)$, respectively. Thus, in Cases R1 and R2, if the last cached distance $dLast$ is $dLast \leq d(q, p(v)) + \epsilon$, then we set $high$ to the last object position in v and eschew binary search.

4.4 Cost Analysis

Here, we analyze the cost of the AV-tree index with all enhancements. Assuming that AV-tree leaves of size no larger than θ are not cracked, we expect the tree to reach its maximum size after a large number of queries, whereupon no more cracks are performed. In this state, each leaf has $\theta/2$ objects on average, so the expected number of leaves is $\frac{2n}{\theta}$, where n is the number of objects in O. Since the AV-tree is a binary tree, the expected number of nodes is $\frac{4n}{\theta} - 1$, hence the index space complexity is $O(n/\theta)$. The worst-case cost of query processing is $O(n)$, accessing all leaves and data objects and computing their distances to q. So is the cost of the first query over an uncracked array. However, after a large number of queries, we expect the cost per query to drop and to converge to that of using a fully built VP-tree [47], since the AV-tree is expected to be balanced thanks to mediocre cracks (see Section 4.3.1) dividing leaves into two pieces of roughly equal numbers of objects. Lastly, the space overhead of distance caching (Section 4.3.3) is negligible as we only store one scalar (i.e., a float) per object. Thus the space requirements of caching are $O(n)$, whereas those for storing the D-dimensional data array are $O(Dn)$.

5 EXPERIMENTAL EVALUATION

We evaluate AV-tree against the following competitors:

- **Linear scan** computes distances $d(q, o)$ for all $o \in O$ and does not perform any data array re-organization or indexing.
- **SimplePivot** is described in Section 2. After experimental tuning, we opted to set the number of pivots m to 5, which yields the best performance; this parameter value selection is consistent with the experimental setup in [7].
- **MVP-tree** [6] is the best performing high-dimensional index for memory-resident data, according to [7]. We used the same implementation[3] as in [7]. MVP-tree has two parameters, bucket size (equivalent to the threshold θ in AV-tree) and arity (i.e., number of children per node). Through experimentation, we determined that MVP-Tree performs the best with bucket size 64 and arity 5.
- **AKD-tree** [39] is the state-of-the-art adaptive index for multi-dimensional points; we used the authors' implementation[4]. To prevent excessive tree growth, we select 128 as the size threshold after experimentally assessing various values. The original implementation handles rectangular queries, i.e., the L_{max} distance metric. To adapt it to the L_2 distance metric, we first perform an L_{max}-query for the surrounding tangent box using the given q and ϵ, and then filter false positives by a L_2-based linear scan.
- **SAT** is an implementation[3] of the Spatial Approximation Tree [38] (see Section 2.1.2), which has no construction parameters.

Table 1: Datasets used in experiments

Dataset	Cardinality	dimensionality	distance	size (MB)
MNIST	70k	5, 20, **50**, 100	L_1, $\boldsymbol{L_2}$	3-55
Words	650k	2-33	edit distance	8
Synthetic	50k, **100k**, 200k, 500k	100	L_1, $\boldsymbol{L_2}$	20-450

5.1 Experimental Settings

Datasets. We use two publicly available real datasets and synthetically generated high-dimensional vectors. Table 1 summarizes statistics about the data with the default values of parameters and distance metrics shown in **boldface**. We provide more details below.

- **MNIST**[5] is a database of 70K handwritten digits [11]. Each digit is stored as a grayscale image with a size of 28x28 pixels. MNIST has been used in numerous similarity search studies (e.g., [3, 28, 34]). We use the UMAP [36] dimensionality reduction method to create various vector representations of the data with D in 5–100.
- **Words** is a database of 650K proper nouns, acronyms, and compound words, taken from the Moby project[6], with lengths varying from 2 to 33 characters. On Words, the query goal is to find words that are similar to a given query string by edit distance.
- **Synthetic** are generated clustered datasets of 10 non-overlapping, equally sized clusters comprising 20K–500K points in 100 dimensions, generated as isotropic Gaussian blobs by the make_-blobs function of the sklearn [41] Python library with a standard deviation of 0.5.

Queries. To evaluate the performance of all methods, we ran workloads of range and kNN queries. In line with previous work [24, 39], our query workload consists of 1000 randomly sampled query points from the target dataset[7] For range queries we tuned ϵ to ensure that queries return the desired number of results. We do experiments with selectivity of 20–1000 (default 100). For kNN queries, we seet k to 20 by default and let it range in $\{5, 20, 50, 100\}$.

Cost measures. In accordance with previous work on adaptive indexing [18, 24, 25, 27, 39], we evaluate all methods by their (i) cost per query and (ii) cumulative cost, as the query workload progresses; we average results over 5 runs. As SimplePivot, MVP-tree, and SAT are built in advance, we add their construction cost to the cumulative cost prior to the first query. Linear scan, AV-tree, and AKD-tree do not bear preprocessing costs.

5.2 Enhancements and parameter tuning

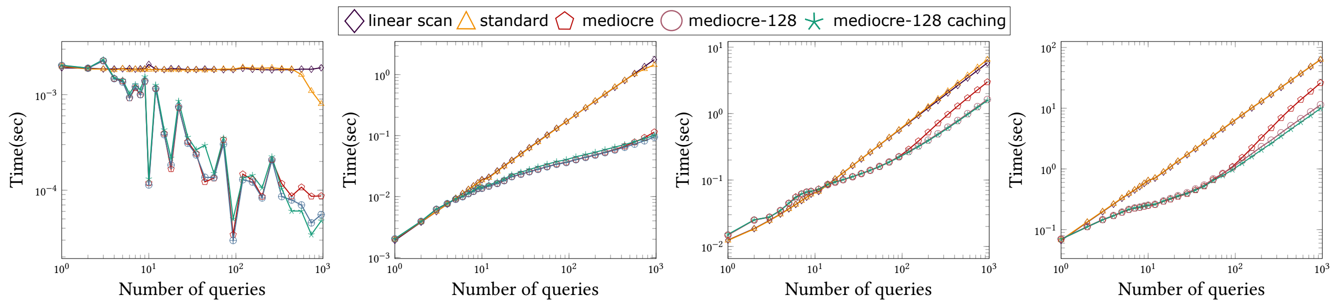
5.2.1 Effect of AV-tree enhancements. First, we evaluate different AV-tree versions with 1000 queries on the 50D MNIST dataset, including the performance of linear scan for reference. We compare the basic version of AV-tree, which uses standard cracking without any enhancements (labeled 'standard') to (i) its variant using mediocre cracking (Section 4.3.1, 'mediocre'); (ii) a variant using mediocre cracking and threshold $\theta = 128$ (Section 4.3.2, 'mediocre-128'); and (iii) a variant that applies all enhancements including caching (Section 4.3.3, 'mediocre-128 caching').

Figures 6a and 6b show the per-query and cumulative costs, respectively, of all AV-tree variants on MNIST, while Figures 6c and 6d show their cumulative costs on Sythetic and Words. Notably, both mediocre cracking and thresholding boost performance, with the effect of thresholding being smaller on MNIST. Mediocre cracking creates a balanced tree, as each crack splits a leaf in two partitions of roughly equal size, while thresholding avoids building an excessively tall tree, which would be detrimental to performance, as its traversal does not pay off compared to the achieved savings. On the other hand, caching pays off in cases where distance computation is expensive, e.g., on the Words data, where we use edit distance.

Table 2 shows the cumulative costs, distance computations and number of AV-tree nodes after 1000 range queries on the 50-dimensional MNIST. The fully optimized AV-tree surpasses all other

[3]https://github.com/kaarinita/metricSpaces
[4]https://github.com/pdet/MultidimensionalAdaptiveIndexing
[5]http://yann.lecun.com/exdb/mnist/

[6]https://en.wikipedia.org/wiki/Moby_Project
[7]In most real applications in metric spaces, queries are not ad hoc, but follow the data distribution. For example, in image similarity search, query images are typically taken from the queried collection; in kNN-based classification, samples to be classified are part of the same collection as the training data.



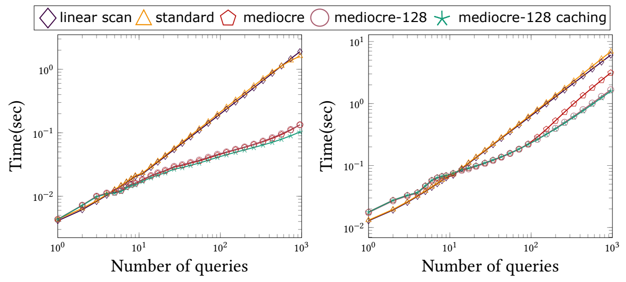
(a) Time per query (MNIST50) **(b) Cumulative time (MNIST50)** **(c) Cumulative time (Synthetic)** **(d) Cumulative time (Words)**

Figure 6: AV-tree versions, 100 selectivity range workload.

versions in all respects. While on these data Mediocre without threshold performs similarly to Mediocre-128, it incurs a significant space overhead by building an AV-tree even bigger than Standard. The same also holds for all other datasets; we omit the corresponding tables in the interest of space.

Table 2: AV-tree versions, MNIST50, post 1k range queries

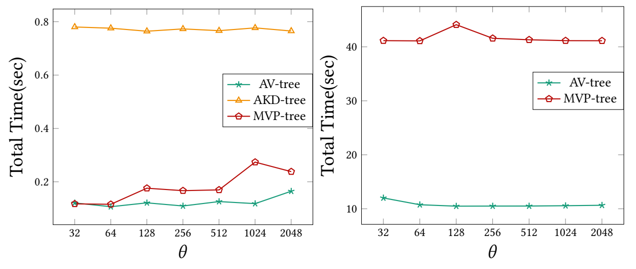
	cum. time	cum. distance comp.	#nodes
Linear Scan	1.8548	70000	-
Standard	1.4831	38818.553	11983
Mediocre	0.1215	1602.105	85809
Mediocre-128	0.1019	1735.236	1843
Mediocre-128 caching	0.1002	1555.677	1843



(a) MNIST50 **(b) Synthetic**

Figure 7: L_1 distance, 100-selectivity range workload

Figure 7 shows the cumulative cost of AV-tree variants on MNIST and Synthetic using L_1 distance instead of L_2 (Euclidean). Note that the performance difference when using L_1 is insignificant. Henceforward, we adopt all enhancements in the AV-tree and use L_2 as a distance measure on MNIST and Synthetic.



(a) MNIST50 **(b) Words**

Figure 8: Parameter Tuning, 100 selectivity range workload

5.2.2 *Parameter setting.*

To set parameter values for all indexes, we tuned them on the MNIST50 and Words data. The AV-tree uses a single threshold parameter, similar to that of the AKD-tree and the MVP-tree, while the MVP-tree requires one more parameter, tree *arity*, which we set to 5, as both our own evaluation and [7] suggest. Figure 8 plots the total cost for an 1K-query workload vs. different threshold values. As the plot shows, the optimal threshold values for AV-tree, AKD-tree, and MVP-tree, are 128, 128, and 64 respectively.

5.3 Comparative study

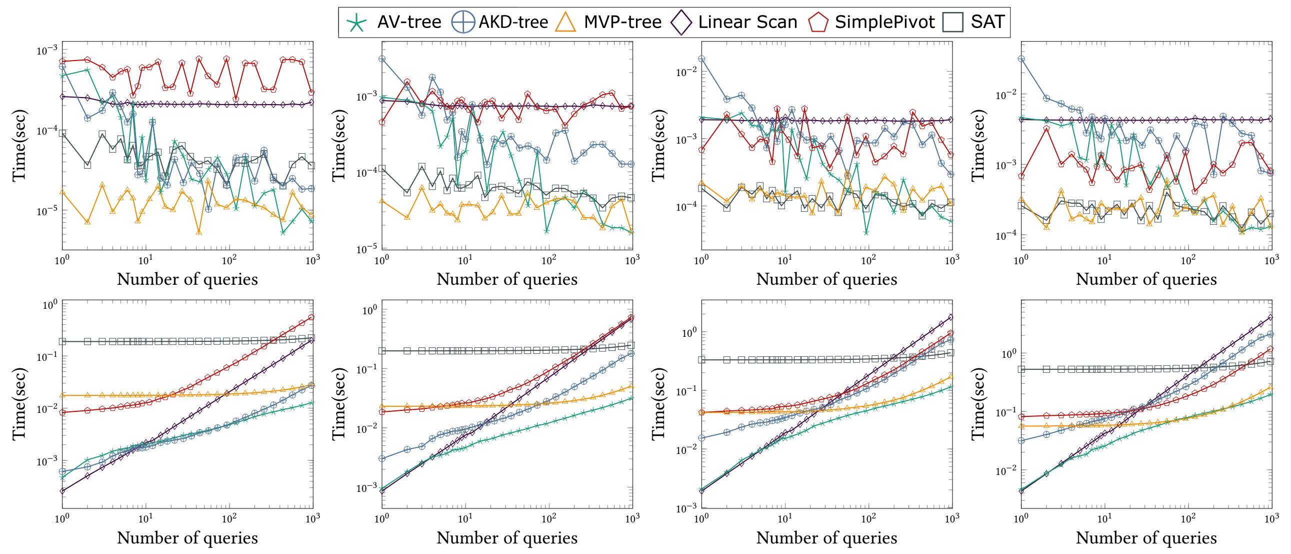
5.3.1 *MNIST.*

Next, we try the fully enhanced AV-tree vs. the competitors listed in Section 5 with range and kNN queries on MNIST.

Dimensionality Figure 9 shows the per-query and cumulative cost, as the query workload (selectivity $s = 100$) progresses, on MNIST datasets of varying dimensionality. AV-tree exhibits the ideal behavior of an adaptive index: its per-query cost gradually drops and reaches that of the MVP-tree. Its cumulative cost outpaces all competitors and eventually matches the MVP-tree. This progression is slower on lower dimensionality; on higher dimensionality, the two lines meet after around 100 queries. The AKD-tree performs competitively to the AV-tree only on very low dimensionality (D=5), where hyperplane-based partitioning works satisfactorily. Until it reaches the size threshold, the AKD-tree creates $2D$ new levels per crack, leading to an exorbitantly tall tree that is expensive to traverse, hence its disadvantage on higher dimensionality. SimplePivot is inferior to MVP-tree and SAT, especially when D is small. These results are consistent with the findings in [7]. MVP-tree has lower per-query cost than SAT in data of medium dimensionality, but the two costs are similar in high-dimensional spaces. Still, SAT incurs a very high start-up (i.e., construction) cost compared to MVP-tree.

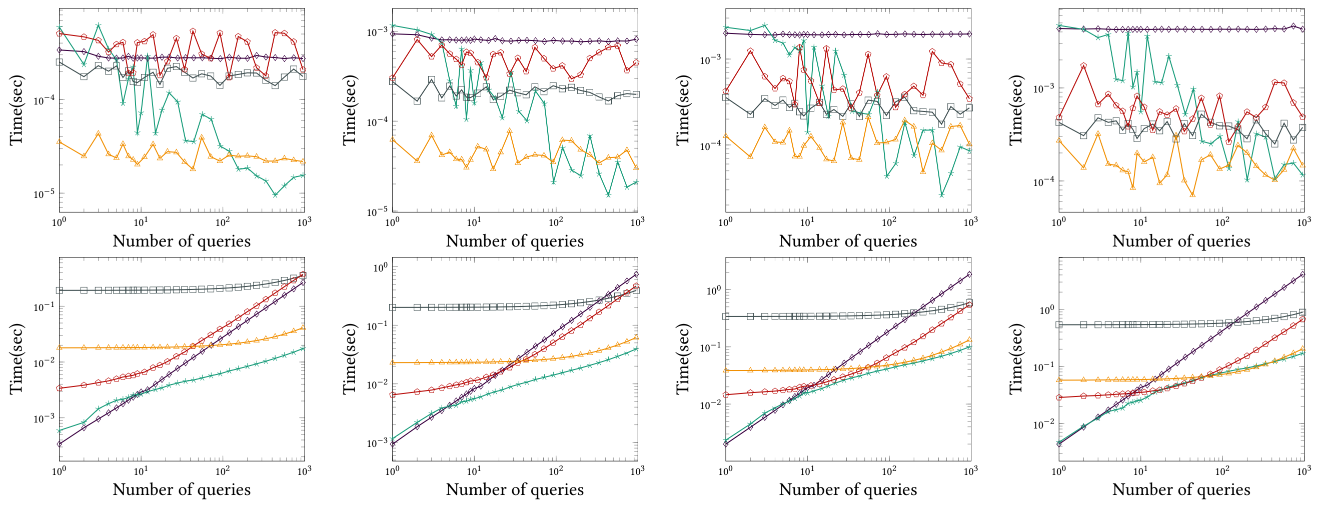
Figure 10 repeats the experiment with kNN queries, setting k to 20. We excluded the AKD-tree from the comparison, as it does not support kNN queries. Our findings reaffirm those for range queries, as the data are reorganized (i.e., cracked) similarly in both cases, leading to a good data structure, while the use of the two priority queues in the AV-tree prevents redundant search.

Selectivity Figure 11 juxtaposes all methods on workloads of varying selectivity s; their relative performance is largely unaffected by selectivity, with the discernible exception of the AKD-tree, which is sensitive to large s due to the expensive L_2-filtering; as s grows, the items to be scanned increase. Cost is largely unaffected by k in kNN queries, as Figure 12 shows. The AV-tree is robust to selectivity, as



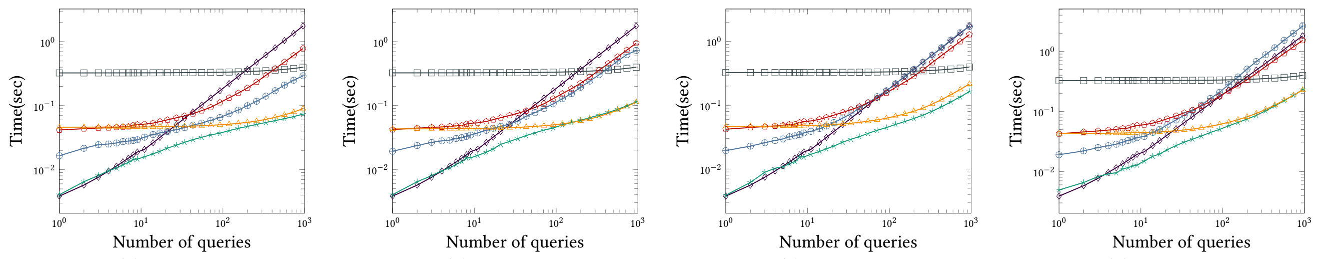
(a) 5D **(b) 20D** **(c) 50D** **(d) 100D**

Figure 9: Effect of dimensionality, MNIST data & 100-selectivity range workload, per query (top) and cumulative time (bottom).



(a) 5D **(b) 20D** **(c) 50D** **(d) 100D**

Figure 10: Effect of dimensionality, MNIST data & 20NN workload, per query (top) and cumulative time (bottom).

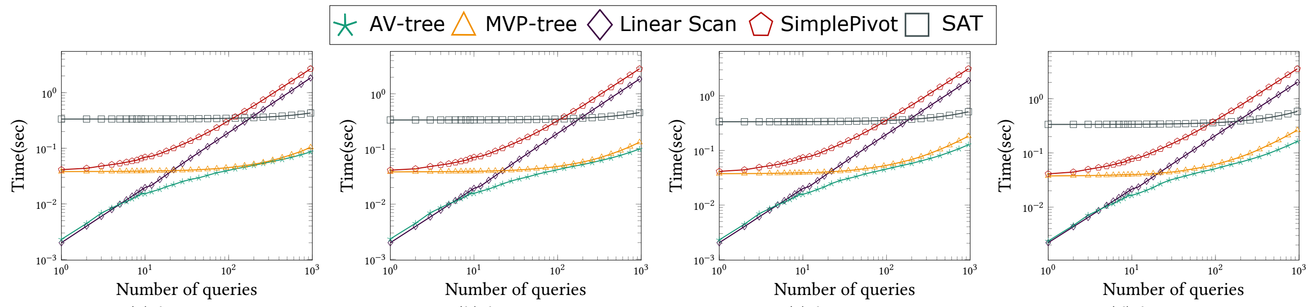


(a) s=20 **(b) s=100** **(c) s=500** **(d) s=1000**

Figure 11: Effect of selectivity, MNIST50 data & range workload, cumulative time.

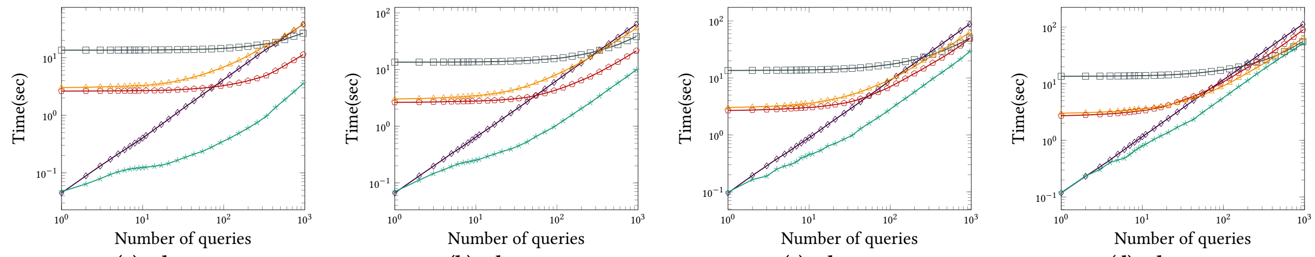
cracking is insensitive to the number of query results and thanks to the data structures it uses to manage *k*NN query results.

Cost breakdown Figure 15 breaks down the total runtime of the default range workloads for the AV-tree and AKD-tree on the default MNIST50 and Synthetic datasets. Total time comprises the costs for: (i) searching the index for relevant partitions (Index Search); (ii) index restructuring, i.e., creating new nodes and swapping (Adaptation); and (iii) scanning data objects in fixed leaves that are not being cracked further (Scan) — in the AKD-tree, Scan includes the time for L_2 filtering. All costs are higher for the AKD-tree: (i) index-search cost due to the ineffectiveness of hyperplane-based



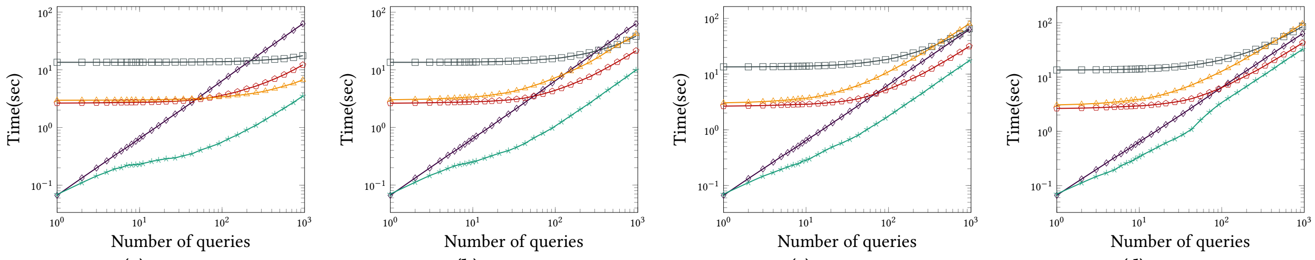
(a) k=10 **(b) k=20** **(c) k=50** **(d) k=100**

Figure 12: Effect of selectivity, MNIST50 data, kNN workload, cumulative time.



(a) wl=4 **(b) wl=6** **(c) wl=8** **(d) wl=10**

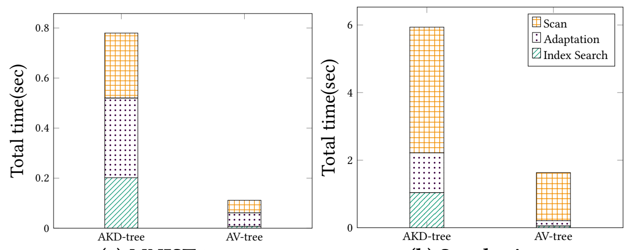
Figure 13: Effect of query length, Words data, edit-distance $\epsilon = 2$, cumulative time.



(a) ϵ=1 **(b) ϵ=2** **(c) ϵ=3** **(d) ϵ=4**

Figure 14: Effect of query selectivity, Words data, 6-letter-word queries, range workload, cumulative time.

partitions and the larger index size, (ii) adaptation cost due to generating more (hyperplane-based) partitions than the AV-tree, (iii) scan cost due to refining spherical range queries.



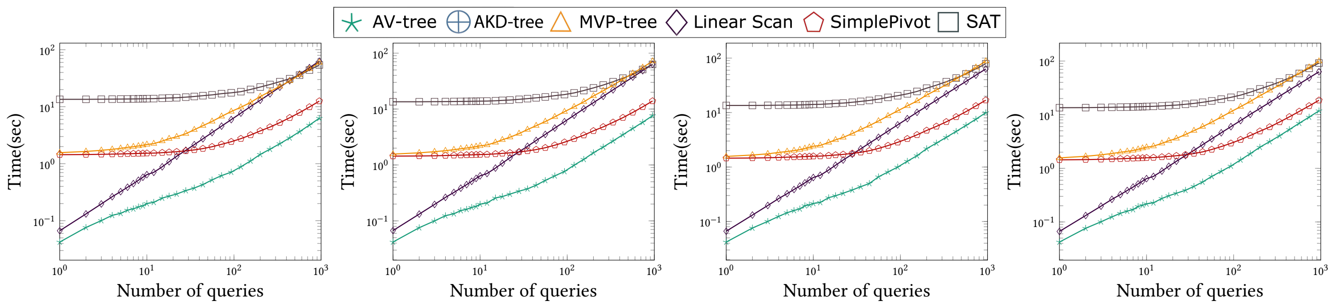
(a) MNIST50 **(b) Synthetic**

Figure 15: Cost Breakdown

5.3.2 *Words.* We next compare all methods for range and kNN query workloads on the Words data. We omit the AKD-tree, as it does not support non-vector data and non-L_p distance measures. Recall that the data includes words of various lengths. First, we create range query workloads by picking 1000 random words of fixed length (4 to 10), set $\epsilon = 2$, and measure the cumulative cost of all methods. As Figure 13 shows, AV-tree outperforms all competitors, and fares better on smaller query word lengths. With longer words, the curse of dimensionality comes into play and all index-based methods acquire costs similar to linear scan; yet even then, the AV-tree outpaces the pre-built SimplePivot and matches the MVP-tree. Remarkably, SimplePivot dominates the MVP-tree on queries of smaller length.

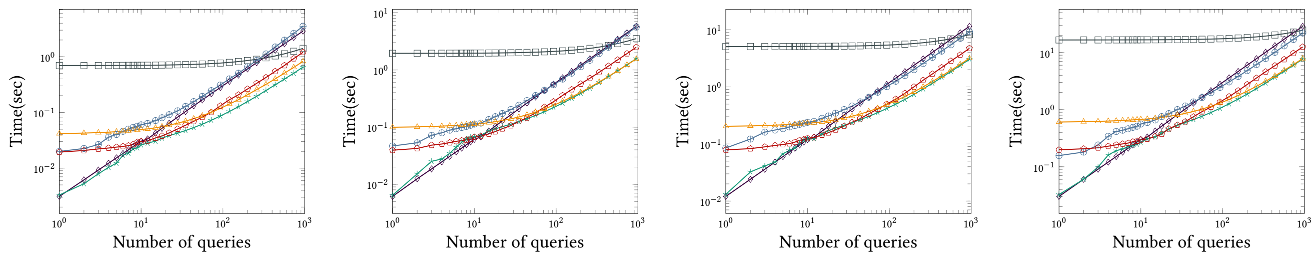
Figure 14, we juxtapose all methods the same workload of length-6 queries, tuning the values of ϵ, i.e., varying selectivity. With more selective queries (lower ϵ), index-based methods outperform linear scan, and the AV-tree gains an advantage. However, indexes are less effective with less selective queries ($\epsilon = 4$), thus the AV-tree advantage diminishes. Lastly, Figure 16 shows the performance of AV-tree on kNN queries vs. the value of k. The results resemble those for range queries of varying selectivity. Overall, AV-tree presents an ideal behavior on the Words dataset, as its cumulative cost is consistently below that of all other methods, with the difference being more striking in the first few hundreds of queries.

5.3.3 *Synthetic data.* We now compare the performance of all methods against synthetically generated datasets of different scale, generated as described in Section 5.1. Figure 17 shows cumulative costs on range query workloads. Noticeably, the superior performance of the AV-tree is insensitive to data scale; its cost is close to that of linear scan in the first few queries and matches that of



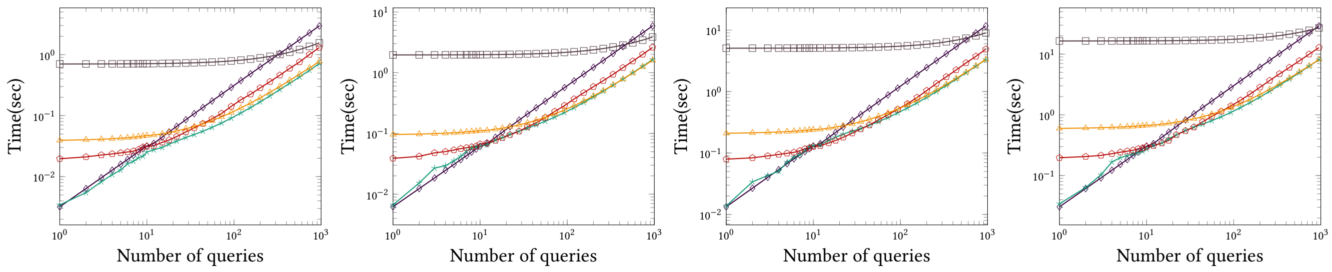
(a) k=10 (b) k=20 (c) k=50 (d) k=100

Figure 16: **Effect of k, Words data, 6-letter-word kNN queries, cumulative time.**



(a) 50K (b) 100K (c) 200K (d) 500K

Figure 17: **Effect of data size, Synthetic 100D data, 100-selectivity range workload, cumulative time.**



(a) 50k (b) 100k (c) 200k (d) 500k

Figure 18: **Effect of data size, Synthetic 100D data & 20NN workload, cumulative time.**

MVP-tree after a few hundreds of queries, while linear scan remains too slow. AV-tree is equally robust to data scale on kNN query workloads, as Figure 18 shows.

5.4 Index Size

Lastly, we compare the *eventual* index sizes of AV-tree, AKD-tree, and MVP-tree, after the default workload of 1K range queries of selectivity 100. As Table 3 shows, the AV-tree is a ligthweight index, as it has size controlled by a threshold and caches at most one distance per object. Compared to the size of the corresponding datasets in Table 1, the AV-tree occupies little space; this is yet another advantage of our method. Remarkably, if we eschew distance caching in AV-tree (2nd column), the index becomes even smaller than the MVP-tree, at the price of a small overhead in the search performance.

Table 3: Index size (MB) after 1K range queries.

	AV-tree	AV-Tree (no cache)	AKD-tree	MVP-tree
MNIST	0.3989	0.1189	2.7989	0.2864
Words	3.2654	0.6654	-	2.7
Synthetic	0.5682	0.1682	2.6732	0.2909

6 CONCLUSIONS

We introduced the *adaptive vantage* tree (AV-tree), the *first*, to our knowledge, adaptive index tailored for high-dimensional metric spaces. In manner reminiscent of previously proposed adaptive indices for single columns [18, 24] and for a few attributes [39, 40], the AV-tree gracefully adapts to a query workload to progressively build a complete high-quality index. Nevertheless, unlike previous adaptive indexing methods, the AV-tree partitions the space around query centers into units defined by hyperspheres using *mediocre* distance bounds that naturally adapt to the data distribution, rather than into rectilinear units. Our experimental study on two real datasets of different natures, with diverse distance metrics, demonstrates that the AV-tree achieves low cumulative query cost compared to (i) iteratively applying a linear scan; (ii) using a pre-built MVP-tree, the state-of-the-art index for *metric spaces*; and (iii) employing the AKD-tree, the state-of-the-art *adaptive* index for multidimensional data. In the future, we intend to investigate the performance of a multiway AV-tree (MAV-tree), which will divide the space around each query into multiple layers based on several distance bounds, in contrast to the current binary space partitioning by mediocre distances.

REFERENCES

[1] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. 2016. k-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-Memory Implementation. *Proc. VLDB Endow.* 9, 6 (2016), 492–503.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. 349–360.

[3] David Alvarez-Melis and Nicolo Fusi. 2020. Geometric Dataset Distances via Optimal Transport. In *NeurIPS 2020*.

[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[5] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When Is “Nearest Neighbor” Meaningful?. In *ICDT*. 217–235.

[6] Tolga Bozkaya and Meral Ozsoyoglu. 1999. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Trans. Database Syst.* 24, 3 (1999), 361–404.

[7] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing Metric Spaces for Exact Similarity Search. *ACM Comput. Surv.* 55, 6 (2023), 128:1–128:39.

[8] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based Metric Indexing. *Proc. VLDB Endow.* 10, 10 (2017), 1058–1069.

[9] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*. 426–435.

[10] Richard Connor. 2016. A Tale of Four Metrics. In *Similarity Search and Applications - 9th International Conference, SISAP (Lecture Notes in Computer Science)*, Vol. 9939. 210–217.

[11] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[12] Nicki Skafte Detlefsen, Søren Hauberg, and Wouter Boomsma. 2022. Learning meaningful representations of protein sequences. *Nature Communications* 13, 1914 (2022).

[13] Ada Wai-Chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. 2000. Dynamic vp-Tree Indexing for n-Nearest Neighbor Search Given Pair-Wise Distances. *VLDB J.* 9, 2 (2000), 154–173.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[15] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Morgan Kaufmann, 518–529.

[16] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, Vol. 426. ACM, 371–381.

[17] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[18] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 5, 6 (2012), 502–513.

[19] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321.

[20] Dorit Hochbaum and David Shmoys. 1985. A Best Possible Heuristic for the k-Center Problem. *Mathematics of Operations Research - MOR* 10 (1985), 180–184.

[21] Pedro Holanda and Stefan Manegold. 2021. Progressive Mergesort: Merging Batches of Appends into Progressive Indexes. In *EDBT*. 481–486.

[22] Pedro Holanda, Stefan Manegold, Hannes Mühleisen, and Mark Raasveldt. 2019. Progressive Indexes: Indexing for Interactive Data Analysis. *Proc. VLDB Endow.* 12, 13 (2019), 2366–2378.

[23] Pedro Holanda, Matheus Nerone, Eduardo C. de Almeida, and Stefan Manegold. 2018. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper). In *DATA*. 393–399.

[24] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.

[25] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a Cracked Database. In *SIGMOD*. 413–424.

[26] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*. 297–308.

[27] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 4, 9 (2011), 586–597.

[28] Omid Jafari, Parth Nagarkar, and Jonathan MontaÃśo. 2020. Improving Locality Sensitive Hashing by Efficiently Finding Projected Nearest Neighbors.

[29] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.

[30] Anders Hammershøj Jensen, Frederik Lauridsen, Fatemeh Zardbani, Stratos Idreos, and Panagiotis Karras. 2021. Revisiting Multidimensional Adaptive Indexing [Experiment & Analysis]. In *EDBT*. 469–474.

[31] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. 1988. Deferred Data Structuring. *SIAM J. Comput.* 17, 5 (1988), 883–902.

[32] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and Exact Inner Product Retrieval in Recommender Systems. In *SIGMOD*. 835–850.

[33] Shuo Li, Kang Li, Jun Yang, Yiwen Liu, Wenqiang Han, and Yaping Luo. 2023. Research on the local regional similarity of automatic fingerprint identification system fingerprints based on close non-matches in a ten million people database – Taking the central region of whorl as an example. *Journal of Forensic Sciences* 68, 2 (2023), 488–499.

[34] Wenye Li, Jingwei Mao, Yin Zhang, and Shuguang Cui. 2018. Fast Similarity Search via Optimal Sparse Lifting. In *NeurIPS*. 176–184.

[35] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[36] Leland McInnes, John Healy, and James Melville. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *CoRR* abs/1802.03426 (2018). arXiv:1802.03426 http://arxiv.org/abs/1802.03426

[37] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR, Workshop Track Proceedings*.

[38] Gonzalo Navarro. 2002. Searching in metric spaces by spatial approximation. *VLDB J.* 11, 1 (2002), 28–46.

[39] Matheus Agio Nerone, Pedro Holanda, Eduardo C. de Almeida, and Stefan Manegold. 2021. Multidimensional Adaptive & Progressive Indexes. In *ICDE*. 624–635.

[40] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *EDBT*. 325–336.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] Yong Rui, Thomas S. Huang, and Shih-Fu Chang. 1999. Image Retrieval: Current Techniques, Promising Directions, and Open Issues. *J. Vis. Commun. Image Represent.* 10, 1 (1999), 39–62.

[43] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.* 7, 2 (2013), 97–108.

[44] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2016. An Experimental Evaluation and Analysis of Database Cracking. *The VLDB Journal* 25, 1 (2016), 27–52.

[45] Tomás Skopal, Jaroslav Pokorný, and Václav Snásel. 2004. PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases. In *ADBIS*.

[46] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2022. DB-LSH: Locality-Sensitive Hashing with Query-based Dynamic Bucketing. In *ICDE*. 2250–2262.

[47] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*. 311–321.

[48] Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. 2020. Revisiting the Theory and Practice of Database Cracking. In *EDBT*. 415–418.