INFORMATIK
BERICHTE

384 – 07/2023

Voronoi-Partitioned Metric Space: The N-tree

Ralf Hartmut Güting, Suvam Kumar Das, Fabio Valdés, and Suprio Ray

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
D-58084 Hagen

Voronoi-Partitioned Metric Space: The N-tree

Ralf Hartmut Güting[1], Suvam Kumar Das[2], Fabio Valdés[1], and Suprio Ray[2]

[1]Fernuniversität in Hagen, Germany
[2]University of New Brunswick, Fredericton, Canada

Abstract

Similarity search is the problem of finding in a large collection of objects those that are similar to a given query object, or pairs of objects similar to each other. It is a fundamental problem in modern applications and the objects considered may be as diverse as locations in space, text documents, images, twitter messages, or trajectories of moving objects. A generic and unifying approach is metric space, which organizes the set of objects solely by a distance (similarity) function with certain natural properties.

Given a set of objects S with a metric distance function, we can create a Voronoi partitioning of S by selecting a subset $C \subset S$ as centers and assigning each element of S to its closest center in C. This can be used to organize index structures as well as distributed computation (e.g. similarity join or similarity clustering).

In this paper, we propose a novel index structure, the N-tree (neighborhood tree), which is essentially a hierarchical Voronoi partitioning. In addition, distances between all centers within a node are precomputed. This enables powerful pruning techniques for range search and kNN search, to avoid many of the expensive distance calculations. We provide a thorough experimental evaluation of parameter settings and comparison with state-of-the-art structures. In many cases, N-tree is the most efficient structure.

1 Introduction

Finding relevant objects in a large collection of objects is a fundamental database problem. For simple attribute data types from a one-dimensional domain such as numbers or strings, exact match queries or range queries are the most important query types, well supported by index structures, e.g. B-trees. For spatial data in a k-dimensional Euclidean space, range queries or nearest neighbor queries are appropriate, well supported by indexes such as R-trees, for example.

Modern applications have to deal with huge collections of data over a wide variety of more complex data types: audio or video data, text documents, photographs, twitter messages, social network profiles, recommendations, points of interest, spatio-textual data, moving object trajectories, to name only a few. The question is how to specify what are relevant objects, what are we searching for.

One option is to somehow map the given objects into k-dimensional vectors, e.g. by extracting k numeric features. After this transformation, query types and indexing techniques for spatial data are available, especially those tuned for high-dimensional spaces.

However, the most simple and natural approach to querying is to select one object from the given domain and ask for objects similar to it, that is, *similarity search.* Similarity can be defined by a distance function. Distance is inverse to similarity; hence, an object is most similar to itself, with distance 0. Such distance functions can in principle be constructed in arbitrary ways. However, if certain properties are known, they can be exploited in search.

A long and fruitful line of research has constructed indexes and search techniques [5] based solely on the fact that the distance function is a *metric.* A metric distance function d fulfills (i) $d(x, y) \geq 0$, (ii) $d(x, x) = 0$, (iii) $d(x, y) = d(y, x)$, and (iv) $d(x, z) \leq d(x, y) + d(y, z)$. The last property is known as the *triangle inequality.* This approach is known as the metric space approach to similarity search [30]. The beauty of this approach is that resulting index structures are automatically applicable to the wide diversity of data types mentioned above, as long as we can define a metric distance function for them. The crucial property for pruning is the triangle inequality.

In this paper, we propose an index structure for metric similarity search based on the concept of a Voronoi partitioning. The Voronoi diagram is a popular structure in computational geometry providing a distance-based partitioning of Euclidean space: Given a set of points S in a k-dimensional space, space is partitioned into regions consisting of the points closest to each point in S.

A *Voronoi partitioning* of a metric space is defined similarly: given a set S of objects with a metric distance function d, select a subset $C \subset S$ as *centers* and assign each element of S to its closest center in C. Let $c_{nn}(s, C)$ denote the center closest to s in C and let $P(u)$ denote the partition assigned to center u.

The purpose of partitioning is to be able to restrict in query processing attention to a subset of partitions. A crucial question is therefore how objects in different partitions can interact, that is, whether an object $s \in P(u)$ can be within distance r from an object $t \in P(v)$. Voronoi partitionings have a nice property, which we call the *range distribution property*: For an object q with $c_{nn}(q, C) = u$, only partitions $P(v)$ can have elements t with $d(q, t) \leq r$ for which holds:

$$d(q, v) \leq d(q, u) + 2r$$

The proposed N-tree has parameters k and l for inner node size and leaf size, respectively. To build an N-tree for a dataset S, k centers are selected (e.g. randomly) to form C and S is partitioned by C; if resulting partitions are larger than l, subtrees are created for them recursively. Hence so far an N-tree is simply a hierarchical Voronoi partitioning.

A simple range search algorithm for query object q with radius r would determine in a node the closest center and then use the range distribution property to determine the partitions that need to be searched recursively. This works, but can be further optimized: To determine the closest center, all distances of q to centers in C need to be evaluated. These distance computations may be expensive.

A second major ingredient to the N-tree is therefore pre-computation of all distances between centers of a node at tree construction time. An idea for range search with query object q is to find the closest center $c_{nn}(q, C)$ and use it as a kind of proxy for q: the known distances from $c_{nn}(q, C)$ to other centers should be similar to the unknown (and expensive to calculate) distances from q to these centers. Therefore we can use the known distances to prune away many of the expensive calculations.

Moreover, we can apply an iterative approach already in finding the closest center: in each iteration, evaluate the distance to some center and then prune away many of the other centers that cannot be the closest center any more, based on this distance. Experiments show that this strategy is very effective.

Index structures for metric space have been studied for several decades, beginning perhaps with the seminal works [13, 25]. These works introduce already two main partitioning strategies used in this field, ball partitioning and generalized hyperplane partitioning. Ball partitioning splits a set of objects by one or more distance thresholds from a given reference object (also called pivot). Generalized hyperplane partitioning uses two objects (pivots) and assigns each object of a set to the closer pivot. The conceptual positions in metric space of equal distance to the two pivots define the generalized hyperplane. The case with more than two pivots, i.e., Voronoi partitioning, has also been studied in the literature and is usually subsumed under generalized hyperplane partitioning. We prefer to use the term Voronoi partitioning to emphasize the non-binary structure and the analogy to the well known Voronoi diagrams.

An index structure using Voronoi partitioning that is most similar to our proposed N-tree is GNAT (Geometric Near-Neighbor Access Tree) [2]. In addition to hierarchical Voronoi partitioning[1], for each pair of centers (called *split points*) (u, v) the range of distances from u to elements of the partition $P(v)$ is maintained, i.e., the minimal and maximal distance $d(u, x)$ for $x \in P(v)$. This is useful for pruning in range search. However, the direct distances between centers are not maintained so that pruning techniques used in the N-tree are not available. Another more subtle difference is that GNAT partitions the set $S \setminus C$, i.e., only the elements remaining after removing the centers.

A data structure based on the idea of exploiting pre-computed distances between all elements of a dataset in metric space is AESA [26]. Here the elements are called *prototypes* and the problem addressed is to find the closest prototype for a given *test sample* (corresponding to finding the closest center for a given object). AESA also uses the idea of iteratively evaluating the distance to some prototype and then pruning all remaining candidate prototypes that cannot be closest any more.

A weakness of AESA is that the data set is flat, not hierarchical, and the number of precomputed distances and the storage requirements are quadratic. In experiments, datasets of size up to 1000 are considered. The authors try to address this weakness in follow-up papers. In LAESA (linear AESA) [15], from the set of prototypes P a relatively small set B of *base prototypes* is selected and distances are precomputed only between pairs $(b, p) \in B \times P$. Another approach [14] organizes the set of prototypes into a binary tree, combining this with ideas from [26] and [15]. However, the precomputed distances are still global for the dataset, not per node of the tree.

In contrast, the N-tree has a quadratic number of pre-computed distances only per node of the tree, which can be kept small enough, for data sets of arbitrary size.

Our contributions in this work are the following:

- We propose a new index structure for metric space, the N-tree. It combines hierarchical Voronoi partitioning with pre-computation of distances within nodes, enabling new pruning techniques.
- We provide efficient algorithms for tree construction, similarity range search, and similarity k-nearest-neighbor search.
- A careful experimental evaluation of parameter settings for the N-tree is performed such as the used center selection technique, possible distance estimates in kNN-search, and node sizes.
- We provide a thorough comparison of the N-tree with three well-known metric index structures: M-tree [6], GNAT [2], and MVPT [1]. The latter two have been found in a recent survey [5] to be among the best performing main memory indexes in terms of running time and number of distance computations. For these four structures, the behaviour for different node sizes, data sets and distance functions, and range and kNN queries is evaluated. It turns out that the N-tree yields the best results for most parameter settings in range search and in all cases of kNN search.
- The N-tree exhibits a behaviour that, to our knowledge, is not present in other metric indexes: for increasing radius in range queries after some point the number of distance evaluations and the cost *decreases*. We call this the *U-turn effect*.

The rest of the paper is structured as follows. In Section 2 we introduce the motivating range distribution property. Section 3 defines the structure of the N-tree and its construction algorithms. Section 4 addresses algorithms for range queries and their pruning rules, Section 5

[1]The author calls it *Dirichlet domain partitioning*, finding this term more appropriate. A Voronoi cell corresponds to a Dirichlet domain.

k-nearest-neighbor queries. Section 6 contains the experimental evaluation. An overview of related work is given in Section 7. Section 8 provides conclusions and hints for future work.

2 Preliminaries: Range Distribution on a Voronoi Partitioning

Suppose we are given a data set S with a distance function d. We partition S by selecting some of its elements as centers and assign each element of S to its closest center. We call this a *Voronoi partitioning*. Let $P(u)$ denote the partition of u, the subset of S assigned to center u.

Now let u be a center and $s \in P(u)$. We want to perform a range query for s with radius r on S, retrieving all elements of S within distance r from s. The question is: *which other partitions $P(v)$ may contain results?*

An answer to this question has several applications. A Voronoi partitioning can be used on the one hand to organize subtrees of a node in an index structure. On the other hand, the partitions can be assigned to different nodes of a distributed computing cluster. In the first case, for a given query point, all relevant subtrees need to be searched; in the latter case, the query point has to be sent to all relevant partitions in the cluster for distributed processing.

Assuming d is a metric distance function, an answer to this question has been given several times in the literature on metric indexing as well as in distributed similarity computation. In indexing, it is known as a pruning rule associated with generalized hyperplane partitioning, called *double pivot filtering* in [5] and attributed to [30]. In distributed computation, it has been rediscovered at least in [22] for similarity join and in [11] for density-based similarity clustering. Here, we review Theorem 6.1 and its proof from [11].

For $s \in S$, let $N_{Eps}(s) = \{t \in S \mid d(s,t) \leq Eps\}$. Here Eps corresponds to radius r.

Theorem 2.1 *Let $s, t \in S$ and $T \subset S$. Let $u, v \in T$ be the elements of T with minimal distance to s and t, respectively. Then $t \in N_{Eps}(s) \Rightarrow d(v,s) \leq d(u,s) + 2 \cdot Eps$.*

Proof: Let x be a location within $N_{Eps}(s)$ with equal distance to u and v, that is, $d(u,x) = d(v,x)$. Such locations must exist, because s is closer to u and t is closer to v. Then $d(u,x) \leq d(u,s) + Eps$. Further, $d(v,s) \leq d(v,x) + Eps = d(u,x) + Eps \leq d(u,s) + Eps + Eps = d(u,s) + 2 \cdot Eps$.

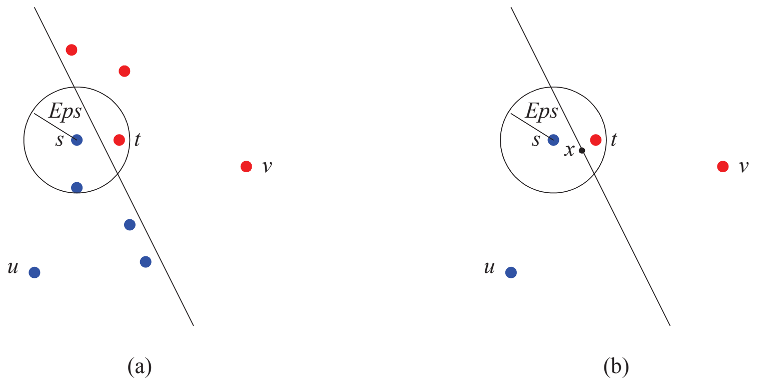


Figure 1: Range search on a Voronoi partitioning (used with permission) [11]

The setting of Theorem 2.1 is illustrated in Figure 1 (a). Here u and v are partition centers; the blue objects are closer to u, the red objects are closer to v; hence $s \in P(u)$ and $t \in P(v)$. The slanted line represents equal distance between u and v (the generalized hyperplane). The proof is illustrated in Figure 1 (b).

Theorem 2.1 says that when we perform a range query with radius r from an object s within a partition $P(u)$, we need to consider all partitions $P(v)$ such that $d(s,v) \leq d(s,u) + 2r$. We call this the *range distribution property* in this paper.

3 The N-tree

Let S be a set with a metric distance function d. An *N-tree* over S is an index structure supporting range search and kNN search on S. In this paper we study it as a main memory index, but it is also suitable as a persistent disk based index.

3.1 Structure

An N-tree (*neighborhood tree*) is a multiway tree like a B-tree or R-tree. It has two parameters: the *degree* k and the *leaf size* l, $k \leq l$.

The basic structure is quite simple. Let C be a subset of S of size k called *centers*. S is partitioned by C, assigning each element of S to its closest center in C. An internal node has a set of entries $(c_i, T(S_i))$ where c_i is a center together with a pointer to a subtree $T(S_i)$ organizing the related subset S_i of S. A leaf node just contains the subset. We start with a definition of this basic structure.

Definition 3.1 Let S be a set with a metric distance function d. A *basic N-tree over* S of degree k and leaf size l is defined as

$$T(S) \quad = \quad \begin{cases} ((c_1, T(S_1)), ..., (c_k, T(S_k))) & |S| > k \\ \quad \text{such that } S = \bigcup_{i \in \{1,...,k\}} S_i, i \neq j \Rightarrow S_i \cap S_j = \emptyset, & \\ \quad \forall i \in \{1, ..., k\} : & \\ \quad\quad c_i \in S, & \\ \quad\quad S_i = \{u \in S \mid \forall j \in \{1, ..., k\} : d(u, c_i) \leq d(u, c_j)\} & \\ S & |S| \leq l \end{cases}$$

□

The complete structure of an N-tree includes for each node some auxiliary information, namely

- all pairwise distances between centers,
- two distinguished centers called *pivots*,
- for each center, a vector of its distances to the two pivot elements,
- for each center c_i in an internal node, the *radius* of its associated subtree $T(S_i)$, defined as the largest distance from c_i to an element of S_i.

Such information is of course used for pruning in range search and kNN search. The definition for the complete structure is:

Definition 3.2 Let S be a set with a metric distance function d. An *N-tree over* S of degree k and leaf size l is defined as

$$T(S) \quad = \quad \begin{cases} (((c_1, T(S_1), r_1), ..., (c_k, T(S_k), r_k)), D, \{p_1, p_2\}, PD) & |S| > k \\ \quad \text{such that} & \\ \quad C = \{c_1, ..., c_k\}, S = \bigcup_{i \in \{1,...,k\}} S_i, i \neq j \Rightarrow S_i \cap S_j = \emptyset, & \\ \quad \forall i \in \{1, ..., k\} : & \\ \quad\quad c_i \in S, & \\ \quad\quad S_i = \{u \in S \mid \forall j \in \{1, ..., k\} : d(u, c_i) \leq d(u, c_j)\}, & \\ \quad\quad r_i = \max_{v \in S_i} d(c_i, v), & \\ (S, D, \{p_1, p_2\}, PD) & |S| \leq l \\ \quad \text{such that } C = S & \end{cases}$$

where

$$D = \{d_{ij} \mid i, j \in \{1, ..., |C|\}, d_{ij} = d(c_i, c_j)\}$$
$$p_1, p_2 \in C$$
$$PD = \{v_i \mid i \in \{1, ..., |C|\}, v_i = (d(c_i, p_1), d(c_i, p_2))\}$$

□

It is possible that a leaf has only one element; in this case D and PD are empty and the p_i are undefined. This special case is omitted in the definition. Further, a subset S with m entries, $k < m \leq l$ may be organized either as an internal node or as a leaf.

Note that there is no balancing condition; the subsets S_i of a node may have different sizes and so the representing subtrees may have different depths.

3.2 Construction

An N-tree is constructed for a set S by selecting k elements from S as centers and then partitioning S by these centers. For each partition, an N-tree is constructed recursively if it has more than l elements.

Algorithm 1: *build*(S, k, l)

```
Input: $S$ - a set of objects with a distance function $d$;
       $k$ - an integer, the degree of the N-tree;
       $l$ - an integer, the maximal leaf size
Output: a node representing $S$
1 if $|S| \leq l$ then
2     compute the auxiliary information $D, \{p1, p2\}, PD$;
3     return leaf($S, D, \{p1, p2\}, PD$)
4 else
5     $C$ := $determineCenters(S, k)$;
6     $\{(c_1, S_1, r_1), ..., (c_k, S_k, r_k)\}$ := $partition(S, C)$;
7     compute the auxiliary information $D, \{p1, p2\}, PD$;
8     return node($((c_1, build(S_1, k, l), r_1), ..., (c_k, build(S_k, k, l), r_k)), D, \{p1, p2\}, PD$)
```

The two pivot elements are selected randomly. Perhaps, the simplest algorithm for *determineCenters* is random selection, which is expected to yield good results. In Section 6.2 we compare it experimentally with three other algorithms for selecting centers.

The algorithm for *partition* needs to determine for each element u of S the closest center in C. A straightforward implementation computes all distances $d(u, c_i)$. A better implementation uses the algorithm *closestCenter* developed in Section 4.2 in the context of range search, which prunes a lot of the expensive distance calculations.

3.3 Updates

It is possible to design simple update algorithms for inserting an element into an N-tree or for deleting an element from it. Insertion of element x follows a path from the root, always selecting the subtree with the closest center from x and inserting x into a leaf. When a leaf overflows, an internal node is constructed for it. On the path down, the radii of subtrees can be updated easily.

Deletion works similarly. However, one cannot easily update radii of subtrees on the path; so they would be left possibly a bit too large, without losing correctness. Precise algorithms are omitted for brevity.

4 Range Queries

In this section we develop algorithms for range queries; Section 5 addresses kNN queries. The main idea is to use the precomputed distances available in nodes to avoid many expensive distance computations.

We briefly review the definitions of range queries and kNN queries.

Definition 4.1 Let S be a set, q a query object, d a distance function applicable to S and q, $r \in \mathbb{R}$ a *search radius*, and $k \in \mathbb{N}$. A *range query* is

$$range(S, q, r) = \{s \in S \mid d(q, s) \leq r\}$$

A *kNN query* (k-nearest-neighbors query) is

$$kNN(S, q, k) = U \subseteq S \text{ such that } |U| = k \wedge \forall u \in U, \forall s \in S \setminus U : d(q, u) \leq d(q, s)$$

□

4.1 Overview

The problem of range search on the root node of an N-tree is illustrated in Figure 2. The partitioning of set S into partitions $((c_1, S_1), \ldots (c_k, S_k))$, where each element of S is assigned to its closest center, corresponds to a Voronoi partitioning of the metric space. Here we illustrate it by a Voronoi diagram in the 2D Euclidean space. There is one center c_x in C that is closest to the query object q; we say q *falls into* partition S_x. In Figure 2 this is partition a. There are other partitions that may contain objects within distance r from q; these are partitions b through i in Figure 2.

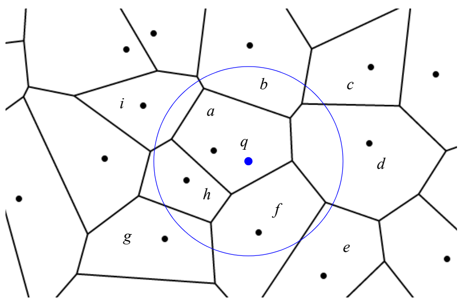


Figure 2: Range query with query point inside partitioning

Hence the search on the root node needs to identify the partition into which q falls as well as the other partitions that may contain objects within its query radius; the respective subtrees need to be searched recursively.

The latter are given by the range distribution property: Theorem 2.1 says that an object p assigned to another partition with center c_y can only have distance $d(q, p) \leq r$ if $d(q, c_y) \leq d(q, c_x) + 2r$.

As it is important for pruning to know the distance $d(q, c_x)$, the algorithm for range search on the root node proceeds as follows:

1. Find the center c_x closest to q;
2. Determine other centers c_y fulfilling the condition $d(q, c_y) \leq d(q, c_x) + 2r$, using precomputed distances as much as possible.
3. Recursively search all qualifying partitions.

At levels of the tree below the root node, however, the situation is not always the same as in the root node. For partition S_x that q falls into, it is the same, so we can search this child of the root node in the same way. However, for the other partitions q lies "outside" as illustrated in Figure 3.

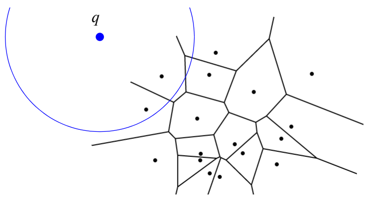


Figure 3: Range query with query point outside partitioning

It turns out that in this case, the pruning criterion of Theorem 2.1 is not effective. It is therefore not a good strategy to find the closest center to q first, also because the pruning of distance computations in finding the closest center is not effective. In the following subsections, we therefore address the following subproblems:

- Finding the closest center to a query point.
- Range search for a query point inside a partitioning.
- Range search for a query point outside a partitioning.
- Overall algorithm for range search.

4.2 Finding the Closest Center

The goal in designing an algorithm to find the closest center for a query point q in a set of centers C is to avoid as many of the expensive distance computations between the query point and a center as possible, using the precomputed distances between centers. The strategy is to consider all centers as candidates and then, in each step, to evaluate one distance $d(q, c_i)$ and to prune all centers c_j based on their known distance to this center, $d_{ij} = d(c_i, c_j)$, that cannot be the closest center any more.

4.2.1 Pruning Rules

Two pruning rules can be determined. We call the first *simple pruning.* It is illustrated in Figure 4. The distance $d(q, c_i)$ has just been evaluated; the known distances between centers are denoted as d_{ij}. We have (triangle inequality for metric distance functions):

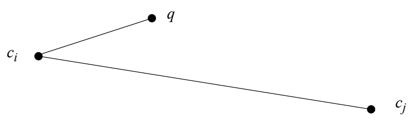


Figure 4: Simple pruning

$$|d_{ij} - d(q, c_i)| \leq d(q, c_j) \leq d_{ij} + d(q, c_i)$$

Now assume $d_{ij} > 2d(q, c_i)$.

$$2d(q, c_i) < d_{ij} \Rightarrow d(q, c_i) < d_{ij} - d(q, c_i) \leq d(q, c_j)$$

Hence c_i is closer to q than c_j and c_j can be pruned from the set of candidates. This is illustrated in Figure 5. All objects with distance larger than $2d(q, c_i)$ from c_i can be pruned.

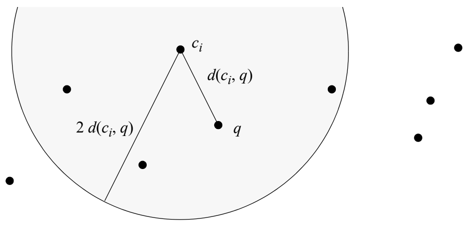


Figure 5: Simple pruning

A second pruning rule is called *mindist pruning*. It is illustrated in Figure 6.

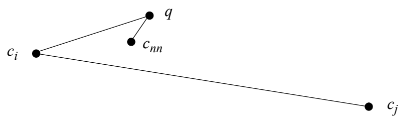


Figure 6: Mindist pruning

This rule uses not only the distance $d(q, c_i)$ just evaluated but also keeps track of the closest center c_{nn} discovered so far and the related minimal distance $d_{min} = d(q, c_{nn})$. It is easy to see that in this case we can prune a center c_j not only when d_{ij} is too large, but also when it is too small. Obviously, any center c_j can be pruned for which holds $d_{ij} < d(q, c_i) - d_{min}$ or $d_{ij} > d(q, c_i) + d_{min}$. Note that the second condition is the one for simple pruning when we substitute $d(q, c_i)$ for d_{min}. Hence this rule subsumes the first one.

4.2.2 The Order of Candidates

For pruning to be effective, it would be good to select first candidates c_i for distance evaluation that are close to q, because then the radius for pruning is small and many points can be pruned early.

Consider evaluating distances from two selected elements of C, r_1 and r_2, to (i) q and (ii) to the nearest neighbor of q, c_{nn}. These distances should be similar. We can represent these distances as 2d vectors $(d(q, r_1), d(q, r_2))$ and $(d(c_{nn}, r_1), d(c_{nn}, r_2))$. We can visualize these vectors as shown in Figure 7.

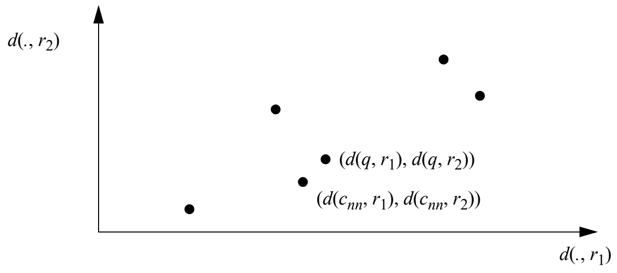


Figure 7: Mapping distances into a Euclidean space

Hence it should be a good strategy to process candidates for distance evaluation with q in the order of increasing distance of their vector to the vector of q. Reference points for mapping arbitrary distances into a Euclidean space are called *pivots* in the literature. Instead of using two pivots, we might also use three or a higher number n, using Euclidean distance in an n-dimensional space. Experiments have shown that three pivots do not yield better results than two, so we use two pivots in our structure.

4.2.3 Algorithm *closestCenter*

The algorithm *closestCenter* can be formulated as shown in Algorithm 2.

```
Algorithm 2: $closestCenter(C, q, D, \{p_1, p_2\}, PD)$
Input: $C$ - the set of centers;
       $q$ - a point;
       $D$ - the set of pairwise distances $d_{ij}$ between centers in $C$;
       $\{p_1, p_2\}$ - the two pivot elements of $C$;
       $PD$ - the pivot distance vectors of $C$.
Output: $c_{nn} \in C$ - the center with minimal distance to $q$;
       $d_{min}$ - the distance between $q$ and $c_{nn}$
1  let $C = \{c_1, \ldots, c_m\}$;
2  let $PD = \{v_1, \ldots, v_m\}$;
3  $v_q := (d(q, p_1), d(q, p_2))$ (*);
4  $C' := \langle c_{i_1}, \ldots, c_{i_m} \rangle$such that $\forall j, l \in \{1, \ldots, m\} : j < l \Rightarrow dEuc(v_q, v_{i_j}) \leq dEuc(v_q, v_{i_l})$;
5  $(c_{nn}, d_{min}) := (\perp, \infty)$;
6  while $not(isempty(C'))$ do
7      $c_i := first(C'); C' := rest(C')$;
8      $u := d(q, c_i)$ (*); $DQ_i := u$;
9      if $u < d_{min}$ then
10         $c_{nn} := c_i; d_{min} := u$
11     $C' := \{c_j \in C \mid u - d_{min} < d_{ij} < u + d_{min}\}$
12 return $(c_{nn}, d_{min})$
```

Lines where expensive distance evaluations occur are marked as (*).

In Line 4, $\langle x, y, z, \ldots \rangle$ denotes the ordered sequence, or list, of elements $x, y, z, \ldots$. In this algorithm, *dEuc* denotes the 2d-distance of vectors in the Euclidean space. Hence the sequence of candidates is returned ordered by increasing distance of their 2d vectors from v_q, the 2d vector of the query point.

In Line 8, after evaluating a distance $d(q, c_i)$ we store it in an array DQ_i. This avoids reevaluating the same distance in the algorithm *rangeSearch1* presented below.

4.3 Query Point Inside Partitioning

After determining the center c_{nn} closest to q and its distance $d_{min} = d(q, c_{nn})$, we can address the range search problem illustrated in Figure 2. In inner nodes, we need to determine partitions that need to be searched recursively; in leaves we need to report centers within the query range r. The idea is to use the known distances d_{min} and d_{ij} to prune centers without performing expensive distance computations, as far as possible.

4.3.1 Nearest Neighbor Pruning

A first pruning rule based on Theorem 2.1 says that a center c_j must be considered (i.e., the search must traverse the related subtree) if $d(q, c_j) \leq d(q, c_{nn}) + 2r$.

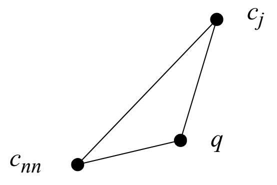


Figure 8: Range query

Due to the triangle inequality we have:

$$d(c_{nn}, c_j) \leq d(q, c_{nn}) + d(q, c_j)$$

Therefore

$$\begin{aligned} d(q, c_j) \leq d(q, c_{nn}) + 2r \Rightarrow d(c_{nn}, c_j) &\leq d(q, c_{nn}) + d(q, c_{nn}) + 2r \\ &= 2d(q, c_{nn}) + 2r \end{aligned}$$

Assuming that c_{nn} is the center with index i, we can rewrite this as

$$d(q, c_j) \leq d(q, c_{nn}) + 2r \Rightarrow d_{ij} \leq 2 \cdot d_{min} + 2r$$

Hence we can retrieve all centers c_j fulfilling $d_{ij} \leq 2 \cdot d_{min} + 2r$ and check whether they also fulfill $d(q, c_j) \leq d_{min} + 2r$. Other centers can be ignored. This requires an expensive distance calculation $d(q, c_j)$.

Let $T = \{c_j \in C \mid d_{ij} \leq 2 \cdot d_{min} + 2r\}$. Can we infer for some elements of T that $d(q, c_j) \leq d_{min} + 2r$? We know (triangle)

$$d(q, c_j) \leq d_{min} + d_{ij}$$

Therefore

$$d_{ij} \leq 2r \Rightarrow d(q, c_j) \leq d_{min} + 2r$$

In summary, we can retrieve all elements of C within distance $2 \cdot d_{min} + 2r$ from c_{nn}. The elements fulfilling $d(q, p) \leq d_{min} + 2r$ must be among them. For those elements c_j retrieved for which $d_{ij} \leq 2r$ holds, we do not need to evaluate the distance to q; they are guaranteed to fulfill $d(q, c_j) \leq d_{min} + 2r$. For the remaining elements, we need to check this condition.

4.3.2 MaxDist Pruning

A second pruning rule uses the maximal distance of any element in the partition S_j from center c_j stored as the radius r_j. Considering a center c_j, we can distinguish the following cases:

1. r is too small to reach c_j (for a leaf) or any element in the partition S_j (for an inner node). We can prune c_j or $T(S_j)$.

2. r is so large that it definitely includes c_j or any element in S_j. We can report c_j or S_j without distance evaluations.

3. We need to evaluate the distance $d(q, c_j)$ (for a leaf) or search the subtree $T(S_j)$ (for an inner node).

These cases can be determined as follows. Note that $d(q, c_i) = d_{min}$, the distance to the nearest neighbor of q.

1. $r < d_{ij} - d_{min} - r_j$ (Figure 9 (a))

2. $r \geq d_{ij} + d_{min} + r_j$ (Figure 9 (b))

3. Otherwise, distance or subtree needs to be evaluated.

In case of a leaf, we can simply set $r_j = 0$.

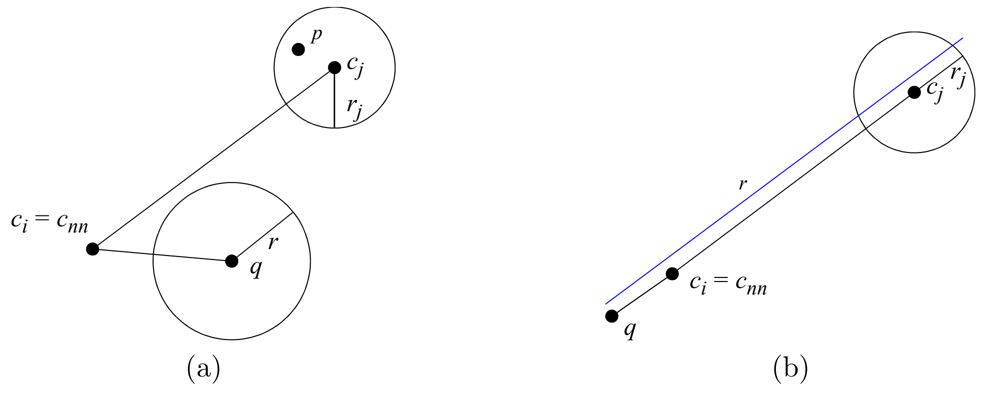


Figure 9: MaxDist Pruning (a) small r (b) large r

4.3.3 Algorithm *rangeSearch1*

The algorithm for range search for the "inside" case combines the two pruning rules. It is shown as Algorithm 3. The following notations are used:

$all(c_j)$ = return all elements of the subtree for center c_j.

$distance(q, c_j)$ =
if DQ_j is defined **then return** DQ_j **else return** $d(q, c_j)(*)$ **endif**

The array DQ_i is defined in Algorithm 2; it stores distances already evaluated in that algorithm.

The algorithm returns separately results already found, the closest center, and other centers whose partitions need to be searched recursively. The partition for c_{nn} will be processed again by this algorithm (for the "inside" case) whereas the other partitions will be processed by the algorithm of Section 4.4 (for the "outside" case).

4.4 Query Point Outside Partitioning

We now address range searching from "outside" a partitioning as illustrated in Figure 3. In this case, it is not effective to determine the closest center initially. Instead, similar as in the algorithm for finding the closest center, we consider the elements of the set of centers sequentially and after each distance evaluation determine elements that can be pruned or reported.

4.4.1 MaxDist Pruning

Again we can use the maximal distance stored as radius r_j for subtree $T(S_j)$. The cases to be considered are the same as in Section 4.3.2, namely

1. r is so small that we can prune c_j or $T(S_j)$.

2. r is so large that we can report c_j or S_j without distance evaluations.

3. Evaluation is needed.

```
Algorithm 3: rangeSearch1(p, q, r)
Input: p – a node of the N-tree;
       q – a query point;
       r – the search radius
Output: c_i – the closest center to q;
        Res – elements within distance r from q, to be returned;
        Search – centers for subtrees to be searched
1  $(c_i, d_{min}) := closestCenter(p.C, q, p.D, p.\{p_1, p_2\}, p.PD)$;
2  $Res := \emptyset$;
3  $Search := \emptyset$;
4  if p is a leaf then
5      foreach $c_j \in p.C$ do
6          if $d_{ij} + d_{min} \leq r$ then $Res := Res \cup \{c_j\}$ else
7              if $d_{ij} - d_{min} \leq r$ then
8                  if $distance(q, c_j) \leq r$ then $Res := Res \cup \{c_j\}$
9  else
10     foreach $c_j \in p.C \setminus \{c_i\}$ do
11         if $d_{ij} + d_{min} + r_j \leq r$ then $Res := Res \cup all(c_j)$ else
12             if $(d_{ij} - d_{min} - r_j \leq r) \wedge (d_{ij} \leq 2d_{min} + 2r)$ then
13                 if $d_{ij} \leq 2r$ then $Search := Search \cup \{c_j\}$ else
14                     if $distance(q, c_j) \leq d_{min} + 2r$ then $Search := Search \cup \{c_j\}$
15 return $(c_i, Res, Search)$
```

We have the following cases. We assume $d(q, c_i)$ has just been evaluated and we consider center c_j.

- Case 1: $d(q, c_i) \geq d_{ij}$
  1. $r < d(q, c_i) - d_{ij} - r_j$. Subtree c_j can be pruned. See Figure 10 (a).
  2. $r \geq d(q, c_i) + d_{ij} + r_j$. Subtree c_j can be reported. See Figure 10 (b).
- Case 2: $d(q, c_i) < d_{ij}$
  1. $r < d_{ij} - d(q, c_i) - r_j$. Subtree c_j can be pruned. See Figure 11 (a).
  2. $r \geq d_{ij} + d(q, c_i) + r_j$. Subtree c_j can be reported. See Figure 11 (b).

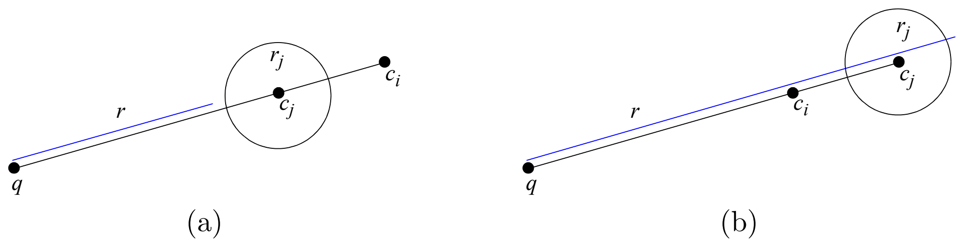


Figure 10: MaxDist Pruning, Case 1: $d(q, c_i) \geq d_{ij}$. (a) small r (b) large r

These results are illustrated in Figure 12. Depending on the radius, we can prune or report the subtree for c_j without distance evaluation.

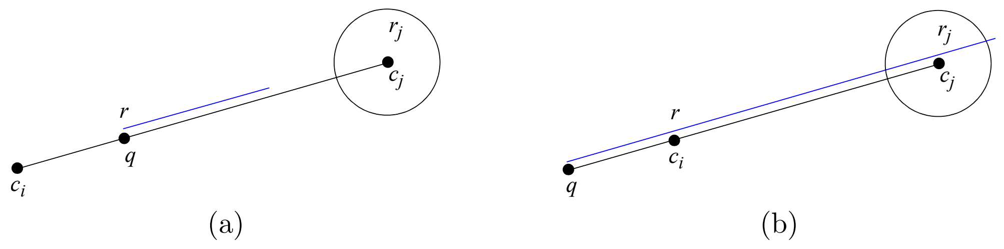
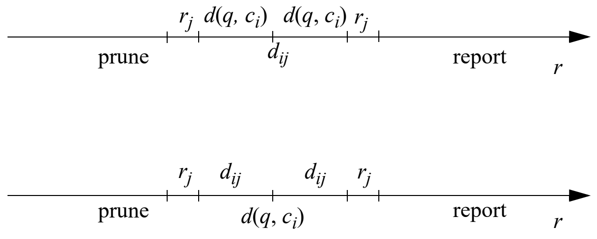


Figure 11: MaxDist Pruning, Case 2: $d(q, c_i) < d_{ij}$. (a) small r (b) large r



r_j $d(q, c_i)$ $d(q, c_i)$ r_j
prune
d_{ij}
report
r
r_j d_{ij} d_{ij} r_j
prune
$d(q, c_i)$
report
r

Figure 12: Pruning or reporting depending on radius

The results can be summarized as follows:

$$r < |d_{ij} - d(q, c_i)| - r_j \quad \Rightarrow \quad \text{subtree } c_j \text{ can be pruned.} \tag{1}$$

$$r \geq d_{ij} + d(q, c_i) + r_j \quad \Rightarrow \quad \text{subtree } c_j \text{ can be reported.} \tag{2}$$

For centers in leaves we have the same analysis setting $r_j = 0$.

So the algorithm to process a node will sequentially consider its centers. For a given center c_i it evaluates the distance $d(q, c_i)$ and prunes or reports all subtrees (elements) qualified by the conditions. The remaining centers are processed in the following steps.

4.4.2 Nearest Neighbor Pruning

When MaxDist pruning is finished, the distances from q to all remaining centers have been computed. Hence at this time also the center c_{nn} with minimal distance d_{min} is known. Therefore the nearest neighbor pruning condition can be applied.

Consider the non-pruned centers (call this set C') in inner nodes. If for $c_j \in C'$ the condition

$$d(q, c_j) \leq d_{min} + 2r$$

does not hold, then we can prune partition S_j because no element of this partition can be within distance r from q. If it were close enough, it would have been assigned to partition S_i for $c_i = c_{nn}$ instead.

4.4.3 Algorithm *rangeSearch2*

The algorithm can be formulated as shown in Algorithm 4. It uses a subalgorithm *prune* shown as Algorithm 5. The notation $node(c_j)$ refers to the root node of the subtree associated with center c_j.

The statement marked with (*) is the only one where an expensive distance evaluation is performed.

Algorithm 4: *rangeSearch2*(p, q, r)

```
Input: $p$ – a node of the N-tree;
       $q$ – a query point;
       $r$ – the search radius;
Output: the set of elements within distance $r$ from $q$
1  let $C = \langle c_1, \ldots, c_k \rangle$ be the centers of node $p$;
2  $Res := \emptyset$;
3  $C' := \emptyset$;
4  $d_{min} := \infty$;
5  while $not(isempty(C))$ do
6      $c_i := first(C); C := rest(C)$;
7      $u := d(q, c_i)$ (*);
8      if $u < d_{min}$ then $d_{min} := u$ if $p$ is a leaf then
9          $Res := Res \cup prune(C, c_i, u, q, r, leaf)$;
10         if $r > u$ then $Res := Res \cup \{c_i\}$
11     else
12         $Res := Res \cup prune(C, c_i, u, q, r, inner)$;
13         if $r > u + r_i$ then
14             $Res := Res \cup all(c_i)$;
15         else if $r > u - r_i$ then
16             $C' := C' \cup \{(c_i, u)\}$;
17 $C' := \{(c, u) \in C' | u \leq d_{min} + 2r\}$;
18 $Res := Res \cup \bigcup_{(c,u) \in C'} rangeSearch2(node(c), q, r)$;
19 return $Res$
```

4.5 Range Search Algorithm

The overall algorithm *rangeSearch* is called on the root node of the N-tree. It starts by applying Algorithm *rangeSearch1* on the root node for the "inside" case. On lower levels it uses either *rangesearch1* or *rangeSearch2* depending on whether the query point is inside or outside the partitioning. The overall algorithm is shown as Algorithm 6.

Algorithm 5: *prune*$(C, c_i, u, r, nodetype)$

```
Input: $C$ - a set of candidate centers;
    $c_i$ - a selected center with distance $u$ from query point $q$;
    $u = d(q, c_i)$;
    $r$ - the query radius;
    $nodetype \in \{leaf, inner\}$;
Side Effect: some subtrees or elements are removed from $C$ and their elements reported
    when appropriate
Output: a set of elements within distance $r$ from $q$
1  $Res := \emptyset$;
2  if $nodetype = leaf$ then
3      for $c_j \in C$ do
4          if $r > u + d_{ij}$ then
5              $Res := Res \cup \{c_j\}$; $C := C \setminus \{c_j\}$;
6          else if $r < |u - d_{ij}|$ then
7              $C := C \setminus \{c_j\}$;
8  else
9      for $c_j \in C$ do
10         if $r > u + d_{ij} + r_j$ then
11             $Res := Res \cup all(c_j)$; $C := C \setminus \{c_j\}$;
12         else if $r < |u - d_{ij}| - r_j$ then
13             $C := C \setminus \{c_j\}$;
14 return $Res$
```

Algorithm 6: *rangeSearch*(p, q, r)

```
Input: $p$ – a node of the N-tree;
    $q$ – a query point;
    $r$ – the search radius
Output: the set of elements within distance $r$ from $q$
1  $Res := \emptyset$;
2  $(c_{nn}, Result, Search) := rangeSearch1(p, q, r)$;
3  if $p$ is a leaf then
4      $Res := Result$;
5  else
6      $Res := Res \cup rangeSearch(node(c_{nn}), q, r)$;
7      $Res := Res \cup Result$;
8      $Res := Res \cup \bigcup_{c \in Search} rangeSearch2(node(c), q, r)$;
9  return $Res$
```

5 k-Nearest-Neighbor Queries

Apart from range search, another important query addressed by metric indexes is to identify the *k-nearest-neighbors (kNN)* from a query point. To process a *kNN* query, one of the three approaches described below is generally taken: [5]

1. Range search is performed several times starting with a very small radius and then the search radius is increased gradually until *k-nearest-neighbors* are found.
2. The search radius is initially set to infinity and then objects in the indexes are visited in the order of increasing distance to the query point, where the search radius is gradually tightened. This is the most commonly taken approach.
3. A set of candidate objects is determined and the distance from q to its k^{th} nearest neighbor in this set is determined. Then a range search with this distance is performed.

In the N-Tree, a combination of these approaches is used, where starting from a small radius, the radius is gradually increased until we find an approximate radius r_{approx} from the query point within which it is guaranteed that all the *k-nearest-neighbors* lie (possibly along with other points). Then *range search* is employed only once with r_{approx}, from which the *k-nearest-neighbors* are obtained. The *kNN* technique is shown in Algorithm 7.

Algorithm 7: $kNN(root, q, k, DE)$

```
Input: $root$ – the root node of the N-tree;
       $q$ – a query object;
       $k$ – the number of neighbors to find;
       $DE$ – the distance estimate DE to choose;
Output: $k$ nearest neighbors from $q$
1 $approxRadius := getApproxRadiusX(root, q, k, DE)$;
2 $Res_1 := rangeSearch(root, q, approxRadius)$;
3 $Res_2 := \emptyset$;
4 for $c_i \in Res_1$ do
5     $dist_i := distance(q, c_i)$;
6     $Res_2 := Res_2 \cup (c_i, dist_i)$;
7 Sort $Res_2$ in increasing order of $dist$;
8 $finalResult :=$ first $k$ elements of $Res_2$;
9 return $finalResult$ ;
```

The backbone of the *kNN* algorithm is to find the approximate radius within which it is guaranteed that all the *k-nearest-neighbors* lie (Line 1). We propose two algorithms *getApprox-Radius1* (Algorithm 8) and *getApproxRadius2* (Algorithm 9) to approximate the radius.

Both algorithms are similar. They differ in whether they use the closest center in certain nodes. It is up to the user to choose either *getApproxRadius1* or *getApproxRadius2*, which is denoted as *getApproxRadiusX* in the *kNN* algorithm. The overall idea of both algorithms is the same which can be described as follows.

We maintain a priority queue Q that stores single data objects as well as partitions (or nodes) based on their distance from the query point q. We start our search from the root node of the NTree. Initially r_{approx} is set to -1, and the root node is pushed into Q. In each iteration, we pop an entry from the head of Q and check if it is a data object or a partition. If it is a data object, we keep track of the number of objects seen so far and also update r_{approx} with the current distance if the current distance is more than the r_{approx} seen so far. If the number of objects seen so far equals k, then we stop execution and return r_{approx}. Otherwise, if the element

```
Algorithm 8: getApproxRadius1(root, q, k, DE)
   Input: root – the root node of the N-tree;
          q – a query point;
          k – the number of neighbors to find;
          DE – DE to choose;
   Output: radius within which all the k points are guaranteed to lie
 1 $Q := priorityQueue()$;
 2 $Q.enqueue((root, 0))$;
 3 $r_{approx} := -1$;
 4 $pointsVisited := 0$;
 5 while $Q \neq \emptyset$ do
 6     $(n, dist) := Q.dequeue()$;
 7     if n is a data object then
 8         $r_{approx} := max(r_{approx}, dist)$;
 9         $pointsVisited := pointsVisited + 1$;
10         if $pointsVisited = k$ then
11             break;
12     else /* n is an internal node or a leaf */
13         let $n.C = \{c_1, ..., c_m\}$;
14         $(c_i, d_{min}) := closestCenter(n.C, q, n.D, n.\{p_1, p_2\}, n.PD)$;
15         $Q.enqueue((c_i, d_{min}))$;
16         if n is an internal node then
17             $Q.enqueue((node(c_i), d_{min} - r_i))$ ;
18         for $j \in \{1, ..., m\} \setminus \{i\}$ do
19             $Q.enqueue((c_j, d_{min} + d_{ij}))$ ;
20             if n is an internal node then
21                 $Q.enqueue((node(c_j), de(d_{min}, d_{ij}, r_j, DE)))$ ;
22 return $r_{approx}$ ;
```

popped from Q is a partition (or node), then we identify a center c_i in the node (depending on which of the two algorithms we use) and find the exact distance between q and c_i. We also find the distance between q and the partition having center c_i, i.e. $node(c_i)$. We then push these two distance information into Q. Next for all other centers c_j in the same node, we find the maximum distance between q and c_j using the triangle inequality since the distance between two centers c_i and c_j is already pre-computed and stored during build time. We also approximate the distance between q and $node(c_j)$ and push it also into Q.

Suppose the exact distance between q and c_i to be d_x, the distance betweens c_i and other centers c_j within a node to be d_{ij} and the radius of the partition with center c_j to be r_j. Thus by the triangle inequality, the distance between q and c_j can be estimated as $d_x + d_{ij}$. Determining a useful distance estimate for a partition is more difficult. Given d_x, d_{ij} and r_j, to estimate the distance between q and the partition $node(c_j)$, various distance estimates **(DE0 – DE8)** are proposed as shown in Table 1 and evaluated experimentally.

Now let us look into *getApproxRadius1* and *getApproxRadius2* and see how the two algorithms are different. Given a partition (or node) that is popped out of Q, the difference is in identifying a center c_i within the partition. In *getApproxRadius1*, c_i is always the closest center from query point q, whereas in *getApproxRadius2* either the closest center or any random center is selected (Line 14 of both Algorithms 8 and 9). We have seen in Section 4 that finding the closest center is not efficient if q is "outside" of the partition. Thus in *getApproxRadius2* we use

Algorithm 9: *getApproxRadius2*$(root, q, k, DE)$

```
Input: $root$ – the root node of the N-tree;
       $q$ – a query point;
       $k$ – the number of neighbors to find;
       $DE$ – distance estimate DE to choose;
Output: radius within which all the k points are guaranteed to lie
1  $Q := priorityQueue()$;
2  $Q.enqueue((root, 0, true))$ /* $isInside$ is set to $true$ */;
3  $r_{approx} := -1$;
4  $pointsVisited := 0$;
5  while $Q \neq \emptyset$ do
6      $(n, dist, isInside) := Q.dequeue()$;
7      if $n$ is a data object then
8          $r_{approx} := max(r_{approx}, dist)$;
9          $pointsVisited := pointsVisited + 1$;
10         if $pointsVisited = k$ then
11             break;
12     else /* $n$ is an internal node or a leaf */
13         let $n.C = \{c_1, ..., c_m\}$;
14         $(c_i, d_x) := chooseCenter(n, q, isInside)$ ;
15         $Q.enqueue((c_i, d_x, isInside))$ ;
16         if $n$ is an internal node then
17             $Q.enqueue((node(c_i), d_x - r_i, isInside))$ ;
18         for $j \in \{1, ..., m\} \setminus \{i\}$ do
19             $Q.enqueue((c_j, d_x + d_{ij}, false))$ ;
20             if $n$ is an internal node then
21                 $Q.enqueue((node(c_j), de(d_x, d_{ij}, r_j, DE), false))$;
22 return $r_{approx}$ ;
```

the closest center only if q is "inside" the partition. Otherwise we randomly select any center. This is shown in the *chooseCenter* algorithm(Algorithm 10).

In *getApproxRadius2* the priority queue Q is used to store triplets $(n, dist, isInside)$, whereas *getApproxRadius1* stores the pair $(n, dist)$. Here n refers to a data object or a node, *dist* is the distance between q and n. The *isInside* variable denotes if q lies "inside" the partition or not. Now let us explain how it is identified whether q lies "inside" a partition or not.

Along with the partition (or node) and its distance from q, the *isInside* variable is also pushed into Q. Since the search starts from the root node that holds the entire data space, q will definitely lie within this. Hence we start *getApproxRadius2* by inserting $(root, 0, true)$ into Q (Line 2). While there are items in Q, we *pop* items from Q. We check the status of that item. If it is a node then we invoke *chooseCenter* which returns a center c_i and the distance between q and c_i as d_x (Line 14). Then we push the triplet $(c_i, d_x, isInside)$ into Q (Line 15). Then for an internal node we push the partition having c_i as center, along with is distance from q and *isInside* into Q (Lines 16 – 17). For all other centers c_j within the same node, we consider the c_j and all the nodes along this path in the tree to be "outside". Thus we push c_j, the maximum distance between q and c_j and $false$ into Q. We also push the partition having c_j as center, estimated distance of q to this partition, along with $false$ status (Lines 18 – 21). Thus for all nodes encountered in this path, the center will be chosen randomly. The algorithm *getApproxRadius1* is similar to this algorithm, just without the use of *isInside* and

Table 1: Distance Estimates $de(r_x, d_{ij}, r_i, DE)$

DE0	$\lvert d_x - d_{ij}\rvert - r_j$
DE1	$\lvert d_x - d_{ij}\rvert$
DE2	$\lvert d_x - d_{ij}\rvert + r_j$
DE3	$max(d_x, d_{ij}) - r_j$
DE4	$max(d_x, d_{ij})$
DE5	$max(d_x, d_{ij}) + r_j$
DE6	$d_x + d_{ij} - r_j$
DE7	$d_x + d_{ij}$
DE8	$d_x + d_{ij} + r_j$

Algorithm 10: $chooseCenter(p, q, isInside)$

```
Input: $p$ – a node of the N-tree;
       $q$ – a query point;
       $isInside$ – a Boolean value identifying whether $q$ lies inside or outside of partition;
Output: a center $c_i$;
       $d_x = distance(q, c_i)$
1 if $isInside$ then
2     $(c_i, d_x) := closestCenter(p.C, q, p.D, p.\{p_1, p_2\}, p.PD)$ ;
3 else
4     choose a random center $c_i$ from $p.C$ ;
5     $d_x := distance(q, c_i)$ ;
6 return $(c_i, d_x)$ ;
```

$chooseCenter$.

6 Experimental evaluation

The N-tree metric index is compared with three other popular metric indexes, namely M-tree [6], GNAT [2] and MVPT [1]. M-tree is chosen because it is perhaps the most well-known metric index. GNAT and MVPT have been found in a recent survey [5] to be among the best performing main memory metric indexes. We employ three real-world datasets and three different metric distance functions as shown in Table 2.

Table 2: Datasets used in the Experiments

Dataset	**Trips**	**X-Rays**	**Words**
Distance Function Used	Hausdorff Distance	L_1-Norm	Jaccard Distance
Object Type	Trajectory	Image	Text
Number of Objects	49,981	55,000	355,000
(Avg.) Object Size	38.2 points	32 x 32 pixels	
Total Size	1,909,385 points		

The *Trips*[2] dataset represents 49,981 trips of New York taxis. The distance between two trips is measured using Hausdorff Distance. *X-Rays*[3] represents 55,000 de-identified images of chest X-Rays [28] in PNG format provided by the National Institute of Health (NIH) Clinical Center. The distance between two images is measured using L_1-norm. *Words*[4] represents 355,000 single

[2]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[3]https://nihcc.app.box.com/v/ChestXray-NIHCC
[4]https://web.archive.org/web/20170930060409/http://icon.shef.ac.uk/Moby/

words taken from the Moby project. The distance between two words is measured using Jaccard Distance.

We break the evaluation into several sections. First, we try to understand the geometry of all the datasets by measuring their distance distributions. This is important because the performance of metric indexes highly relies on the data distribution. Second, we compare different partitioning schemes of the N-tree for choosing centers and find out which scheme is the most efficient. Third, we identify the best version of the kNN algorithm for the N-tree. As we have already seen, there are 2 algorithms to identify the approximate radius within which the *k-nearest-neighbours* must lie and each of them has 9 distance estimates (DE's), resulting in 18 different versions of the kNN algorithm. We identify which combination of algorithm and DE is the best for the N-tree. Fourth, we vary the node sizes of each of the four tree index structures that we compare and identify the size for which each of the structures performs the best. Fifth, using all the parameters that we identified so far we evaluate the performance of the N-tree against all the other structures on *range* and *kNN search.* The performance metrics are the query execution time and the number of distance evaluations and we show that the N-tree outperforms the other structures in these metrics. Lastly, to emphasize the superior performance of the N-tree, we show the effect of the *radius* associated with each center and how it is effective in reducing the number of distance evaluations. Throughout all experiments, the leaf node degree is always maintained to be 100, unless specified otherwise.

Table 3: Parameter Settings

Parameter	**Values/Methods**	**Default**
Number k	5, 10, 20, 50, 100	20
N-tree Partition Scheme	Random, Greedy, BP, HF	Greedy
N-tree kNN version	kNN1-0, kNN2-0, ..., kNN1-8, kNN2-8	kNN2-3

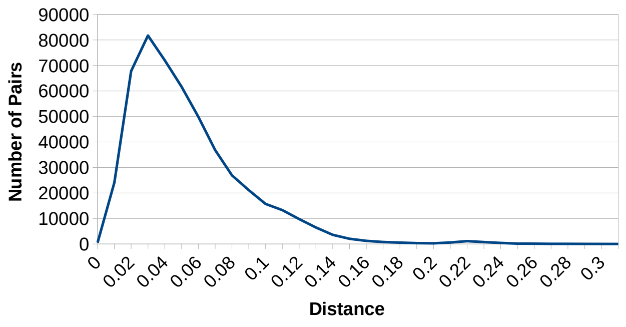
All the four indexes and the associated similarity search algorithms were implemented in Java. All experiments were conducted on an Intel Xeon E5472 CPU having 16GB RAM. Each measurement we report is an average over 100 query points chosen randomly from the respective dataset.

6.1 Distance Distributions Of The Different Datasets

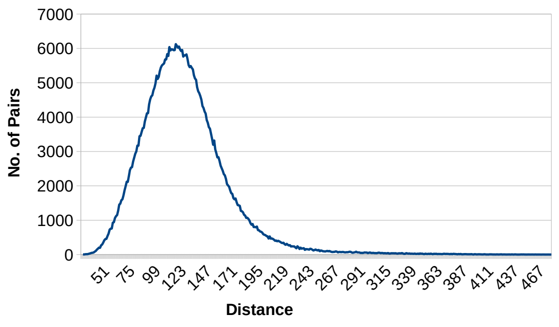
In order to find the distance distributions of the different datasets, 500K unique pairs were randomly selected and the distances between them were calculated using the distance functions shown in Table 2. The distance distributions of the three different datasets obtained are shown in Figure 13. The Y-axis shows the number of data object pairs that have the corresponding distance value. For the *Trips* and *X-Rays* dataset, the distributions are similar to a Gaussian curve, whereas for *Words*, the distribution contains a number of "spikes". For *Trips*, most of the data points are close to each other and the distance between any two trips mostly lies within the interval [0, 0.16] having a peak at distance 0.03. For *X-Rays*, the images are also close to each other, but not as close as in the *Trips* dataset. The majority of the image distances lies within the interval [51, 267], the peak being around the value 123. The distance distribution of the *Words* dataset is quite different from the other two datasets. Here most of the points are far away from each other, whereas some points are very close to each other forming a cluster which are denoted by the "spikes" in the distribution.

6.2 Comparison Of N-tree Based On Different Partitioning Schemes

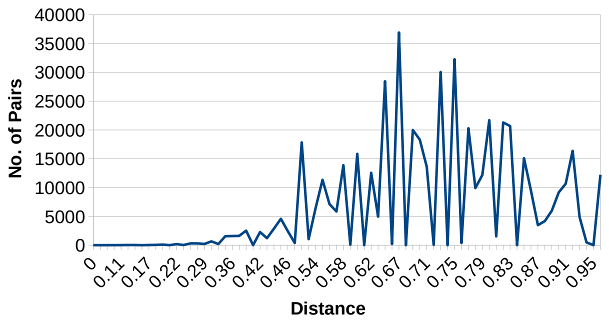
In this section we evaluate different partitioning schemes that can be employed in the construction of the N-tree, i.e., *Random*, *Greedy* [2], *Base-prototypes selection (BP)* [15] and *Hull of Foci (HF)* [24]. In the *Random* approach, the k centers within each node are randomly selected. In



(a) Trips



(b) X-Rays



(c) Words

Figure 13: Distance distributions of the different datasets

the *Greedy* method, the center selection is the same as that used for the construction of GNAT, where for each node k points are selected (using a greedy algorithm), which are farthest apart out of $3k$ randomly chosen candidate points. In the *Base-prototypes selection* technique, the point whose accumulated distance from the previous centres is maximum is selected as the new center, until k centers are found. The *Hull of Foci* approach aims to choose centers that are near the hull of the dataset.

We use the *Trips* dataset and compare the performance of build time, range search and kNN search for each variant of the N-tree. The N-tree is built having node size **36** (we will show in Section 6.4 that this is the best node size we consider). For range search, the search radius was varied as $\{74, 148, 296, 592, 1185\} \times 10^{-4}$. For kNN search, the **kNN2-3** algorithm was used (we will show in Section 6.3 that this is the best kNN algorithm). The values of k were varied as per Table 3. The N-tree build times for different indexes are shown in Figure 14, whereas the performance of *range* and *kNN search* are shown in Figures 15 and 16, respectively.

From Figure 14 it is seen that building the N-tree with *Random* partitioning takes the least time. The *Greedy* approach takes slightly more time than *Random*. But the build time of the N-tree using *BP* and *HF* is comparatively large. From Figures 15 and 16 it is evident that the *Greedy* partitioning scheme is slightly better in terms of query execution time and distance evaluations for both range and kNN search. This helps us to conclude that the *Greedy* partitioning scheme is the best among the four partitioning schemes due to its low build time, query execution time and distance evaluations. Henceforth, for all evaluations, the N-tree is constructed with the **Greedy** partitioning scheme by default. This is also shown in Table 3.

6.3 Comparison Of Different Versions Of kNN search

In this experiment, we compare all the 9 different distance estimates (DE's) for the *getApprox-Radius1* and *getApproxRadius2* algorithms. For this purpose, we use the *Trips* dataset and vary the k values as shown in Table 3. The N-tree is built having node size **36** (we will show in Section 6.4 that this is the best node size we consider). Figure 17 shows the query time and the distance evaluations for all the 18 versions of the kNN algorithm (9 different distance

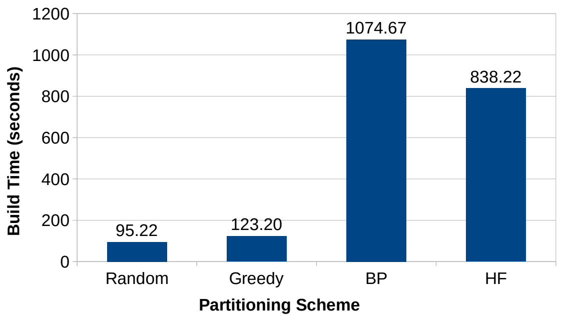
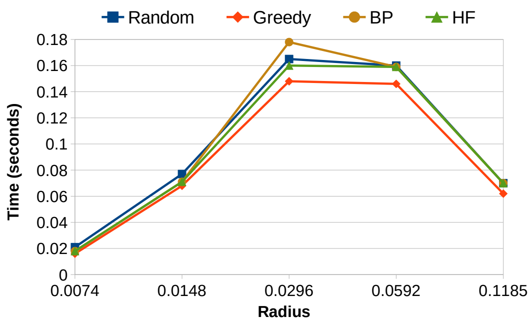
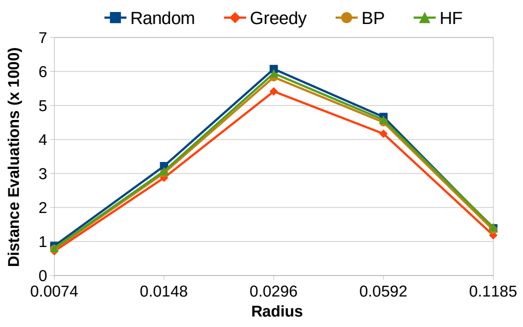


Figure 14: N-tree build time for different partition schemes

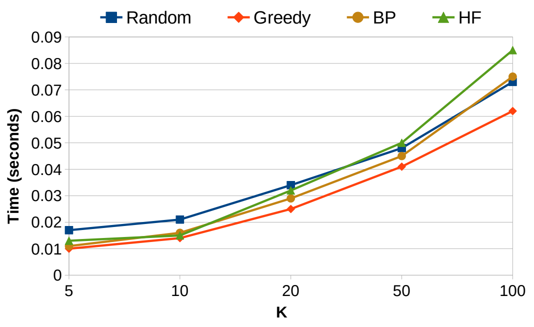


(a) Execution Time

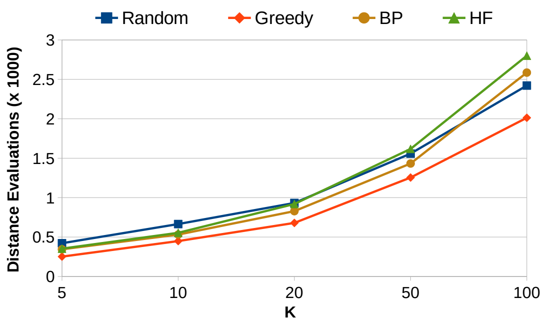


(b) Distance Evaluations

Figure 15: Comparison using range search varying partition schemes

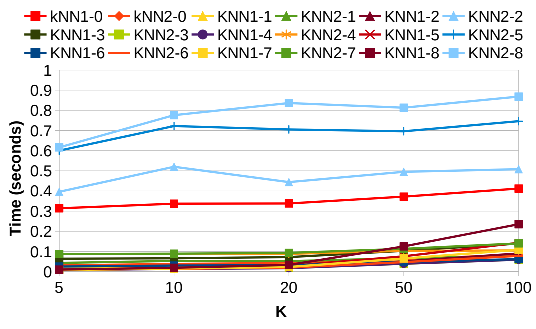


(a) Execution Time

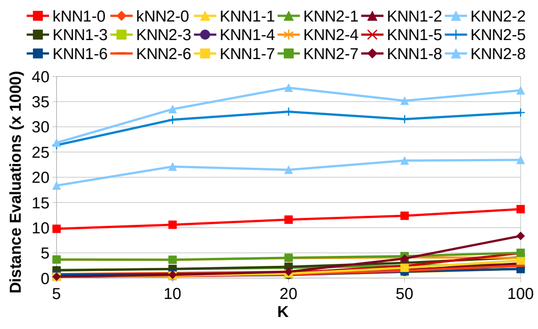


(b) Distance Evaluations

Figure 16: Comparison using kNN search varying partition schemes

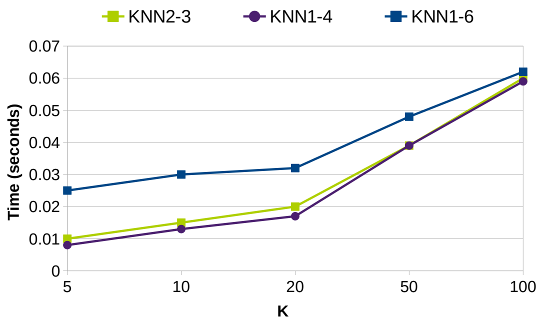


(a) Execution Time

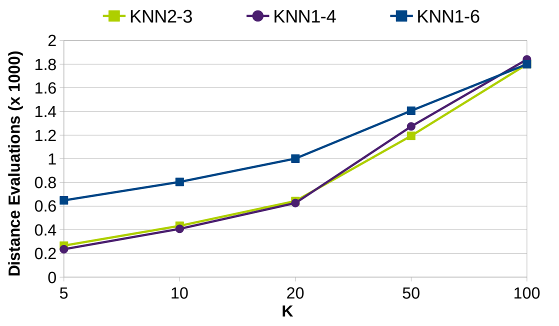


(b) Distance Evaluations

Figure 17: Comparison of the different kNN algorithms for the N-tree



(a) Execution Time



(b) Distance Evaluations

Figure 18: Comparison of the best three algorithms from Figure 17

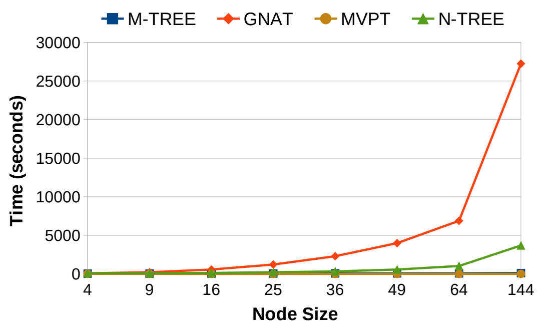
estimates for each of the two algorithms). The versions are labeled *kNNX-Y* where *X* denotes the selection of algorithm *getApproxRadiusX* and *Y* the distance estimate **DEY**. Since it is not clear from Figure 17 which is/are the best algorithm(s) for the N-tree, we take the best 3 plots from the figure, which are *kNN2-3*, *kNN1-4* and *kNN1-6* and create Figure 18.

From Figure 18 it is seen that *kNN1-6* performs the worst among the best 3 plots. The performance of *kNN2-3* and *kNN1-4* is similar in terms of query time and the number of distance evaluations. For small k ($k \leq 20$), *kNN1-4* performs best, after which *kNN2-3* takes the lead in terms of distance evaluations. Generally the number of distance evaluations governs the query execution time. So we consider ***kNN2-3*** as our default kNN algorithm, since its performance is similar to *kNN1-4* for small k and is best for large k. Henceforth for the rest of the evaluations, ***kNN2-3*** is used as the default kNN algorithm. This is also shown in Table 3.

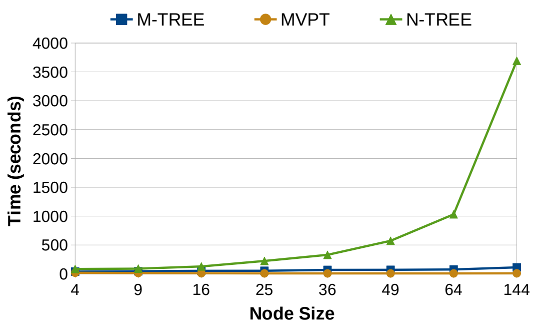
6.4 Comparison Of All Index Structures Varying Their Node Sizes

In this experiment we compare the N-tree with all other index structures by varying the node sizes. The node sizes were maintained to be a square value to maintain uniformity among all the indexes, since the size of a node in MVPT is always a perfect square. We vary the node sizes as 4, 9, 16, 25, 36, 49, 64 and 144. As mentioned before, the leaf node size in all cases is maintained at 100, except for node size 144 where the leaf node size is kept at 200.

The evaluation is performed using the *Trips* dataset. All the indexes were compared based on the performance of *range* and *kNN search*. For *range* search, the search radius is fixed at 0.0074, where the number of data points returned for each query point is around 250 on average. For *kNN search*, k is fixed to 20, the default value as per Table 3. The index build times are shown in Figure 19. From Figure 19(a) we see that due to the high build time of GNAT, the build times of the other structures are not clear. To get a clear understanding of the build times of the other structures, a separate plot is shown in Figure 19(b) which is actually Figure 19(a)

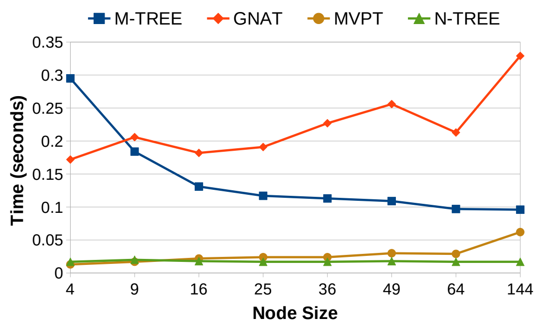


(a) Build Time Of All Indexes

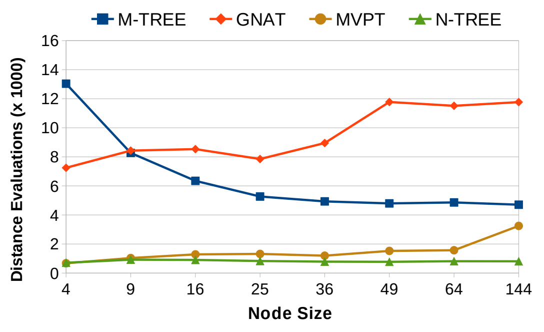


(b) Figure 19(a) without GNAT

Figure 19: Index build time varying node size



(a) Execution Time



(b) Distance Evaluations

Figure 20: Comparison using range search varying node size

without GNAT. The performance of *range* and *kNN search* are shown in Figures 20 and 21.

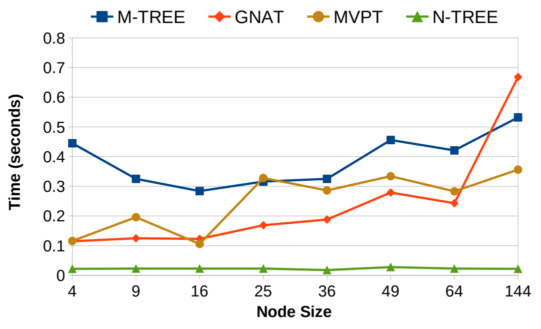
From Figure 20 it is seen that for *range search* on the M-tree the query time and number of distance evaluations decreases as the node size increases. From node size 25 and onwards, the execution time and number of distance evaluations do not differ too much. From Figure 21 we see that in case of *kNN search* the query time and number of distance evaluations first decreases and then increases with the node size. The execution time is minimum at node size 16 whereas the number of distance evaluations appears to be the least at both node sizes 16 and 25. Thus it seems that node size 25 is a good compromise between *range* and *kNN search*. For *kNN search* it is a minimum (equal to 16) and for *range search* it is not much more than the minimum at 144.

In case of GNAT it is clearly seen from Figures 20 and 21 that the query execution time and number of distance evaluations for both *range* and *kNN search* increases in general with the increase of node size. The least value is found to be with node size 4.

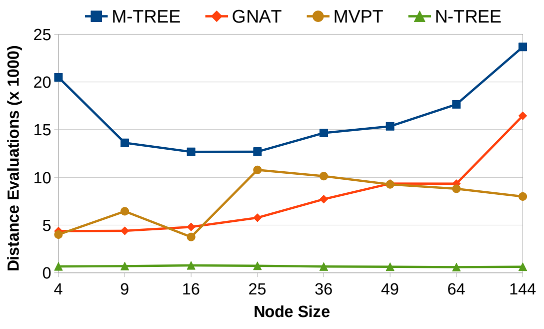
Similar to GNAT, in case of MVPT too it appears that the general trend is that the performance of *range* and *kNN search* decreases with increase of node size. We see that for node size 16, performance of kNN search is the best, but this appears to be an outlier. Since the performance with node size 4 is quite similar to node size 16, node size of 4 is considered to be the best here.

In case of the N-tree we see that the performance is mostly invariant of the node size. On close inspection we see that nodes sizes 36 and 49 are the two best options for range search, whereas it is 64 for kNN search. As seen from Figure 19, since the build time of the N-tree increases with the increase in node size, we consider node size 36 to be the best here.

To conclude, we can say that 25, 4, 4 and 36 are the best node sizes for M-tree, GNAT, MVPT and N-tree, respectively, which we are going to maintain as default for further evaluations.



(a) Execution Time



(b) Distance Evaluations

Figure 21: Comparison using kNN search varying node size

6.5 Comparison Among Different Structures

In this set of experiments, the node degrees were set as mentioned in Section 6.4; the search radius and k were varied to compare the performances of all the indexes. The size of each dataset and search radius for *range* queries were varied as shown in Table 4. For *range search*, the evaluation is further sub-divided into two parts:

1. Keeping the search radius low such that the selectivity is also quite low, i.e., the cardinality of the dataset returned after applying the *range search* is within 250. For each dataset, the first set of radii in the third column of Table 4 denotes the search radii for which the selectivity is low. The radius set $\{148, 296, 444, 592, 741\} \times 10^{-5}$ in the Trips dataset is such an example.

2. Varying the search radius from low to high to compare the behaviour of all the index structures. The radii are increased such that the selectivity varies between 250 to nearly the entire dataset. The second set of radii in the third column of Table 4 denotes such radii. For the Trips dataset, $\{74, 148, 296, 592, 1185\} \times 10^{-4}$ is such an example.

For *kNN* queries, k values were varied as per Table 3, i.e., $k = \{5,\ 10,\ 20,\ 50,\ 100\}$. For both range search and *kNN* search, we compare the running time and the number of distance functions evaluated for N-tree with other structures.

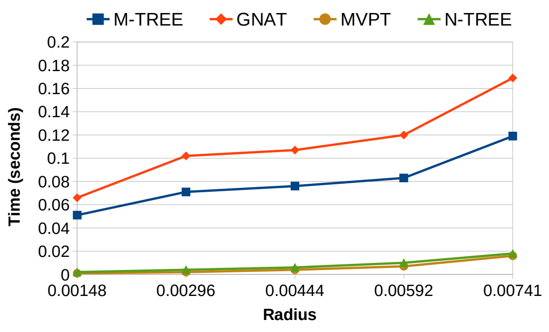
Table 4: Search Radius Setting

Dataset	Cardinality	Radius for range search
Trips	49,981	$\{148, 296, 444, 592, 741\} \times 10^{-5}$, $\{74, 148, 296, 592, 1185\} \times 10^{-4}$
X-Rays	55,000	{30, 34.5, 39, 43.5, 48}, {80, 110, 140, 170, 200}
Words	355,000	{0.08, 0.11, 0.14, 0.17, 0.2}, {0.3, 0.45, 0.6, 0.75, 0.9}

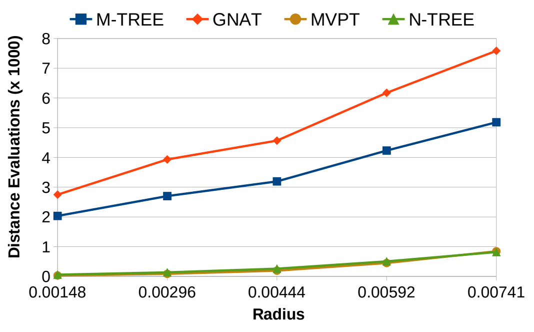
The results for *range search* are shown in Figures 22 and 23. We see that the performance of GNAT and M-tree are the worst for *range search*. The performance of MVPT is closest to the N-tree, hence we only explain the performances of MVPT compared to the N-tree in case of range search.

From Figures 22 and 23 it is evident that for the *Trips* dataset, the performances of MVPT and N-tree are nearly the same. For low-to-high radius, the N-tree always outperforms all other structures. In case of the *X-Rays* dataset, with radius as low as 30, MVPT performs the best. But with gradual increase in radius the N-tree starts performing better. With low-to-high radius, the N-tree again performs the best. With the *Words* dataset also we see a similar behaviour for very low radius. With low-to-high radius, the number of distance evaluations is the least though the running time is not the best in all cases.

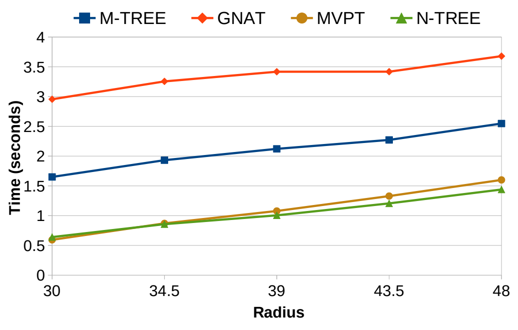
From Figure 24, we see that for *kNN* search the N-tree outperforms all the other structures in all the datasets in terms of both running time and the number of distance evaluations. For the



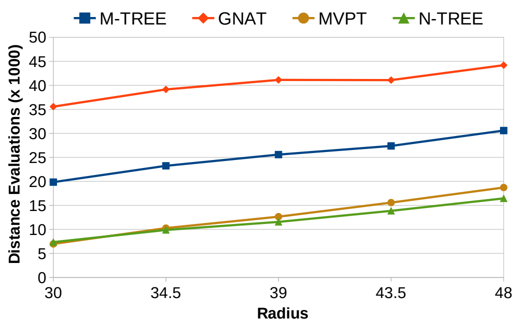
(a) Trips



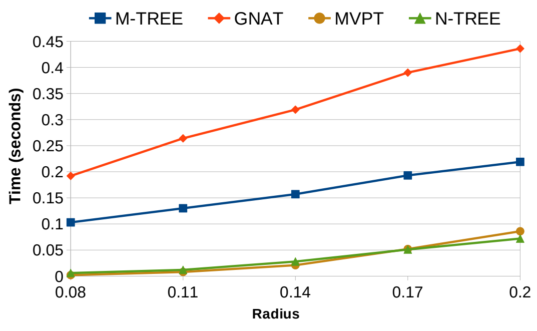
(b) Trips



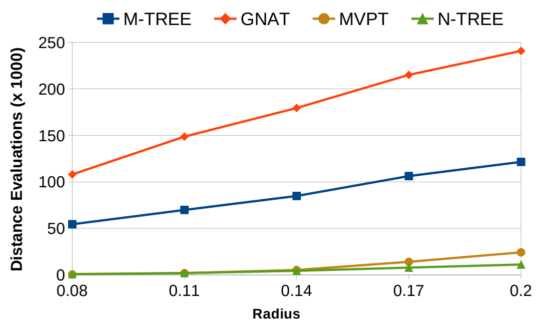
(c) X-Rays



(d) X-Rays



(e) Words

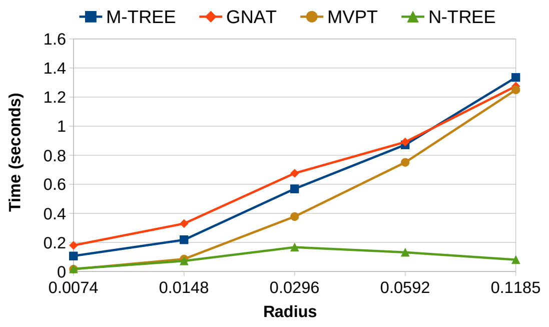


(f) Words

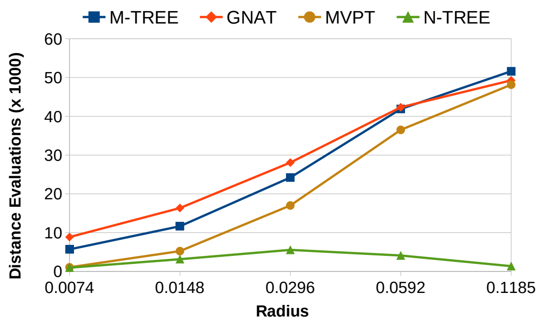
Figure 22: Comparison using range search with very low radius

Trips dataset, the M-tree performs the worst, whereas the performance of GNAT and MVPT are similar. For the *X-Rays* dataset, MVPT performs the worst in terms of running time whereas the M-tree is worst in terms of distance evaluations. For the *Words* dataset, the running time of MVPT is huge compared to the other trees, although the number of distance evaluations is second to best. With k=5, MVPT takes around 6s, whereas the running time of the other trees are within 0.3s. With k=100, MVPT takes nearly 50s to run, but the other trees are within 0.5s. Since the running time for all the other structures are not clear in Figure 24(e) due to the high running time of MVPT, a separate plot (Figure 24(g)) is made without MVPT. From this plot it is clear that the N-tree takes the least running time among all structures.

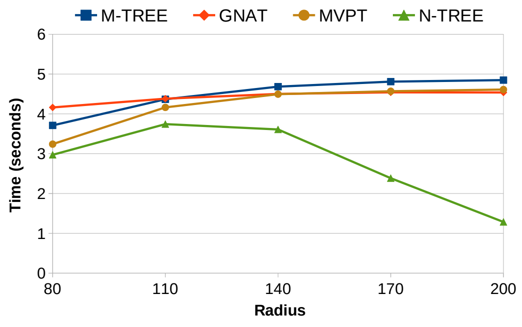
To conclude, we can say that the N-tree always performs better compared to M-tree and GNAT both in *range* and *kNN search*. In case of *range search*, with very low search radius, the performance of the N-tree is similar to that of MVPT, but as the search radius gradually increases, the N-tree performs better than MVPT. For *kNN search*, the N-tree always outperforms the other structures. An interesting observation that is found in the N-tree but not in other structures is that after a certain point, the number of distance evaluations decreases with the increase in search radius for *range* search. E.g. from Figure 23(b), we see that for the *Trips* dataset up to a radius of 0.0296 the number of distance evaluations increases with the increase in search radius, after which it keeps on decreasing, whereas for the other structures the number



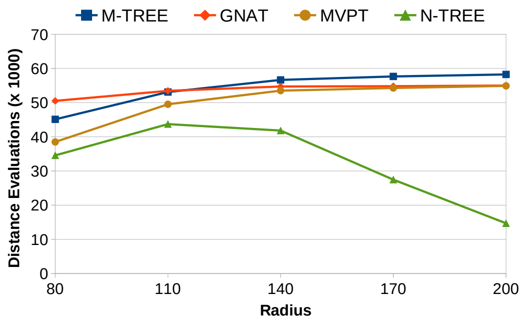
(a) Trips



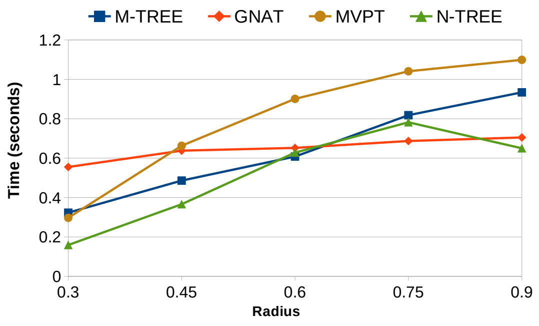
(b) Trips



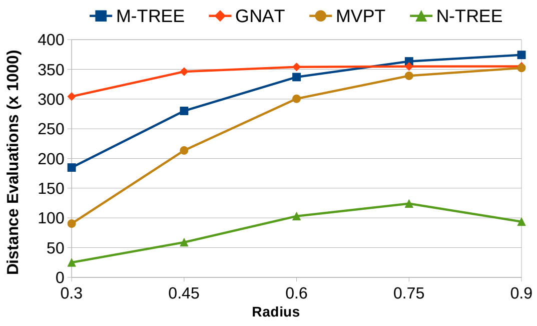
(c) X-Rays



(d) X-Rays

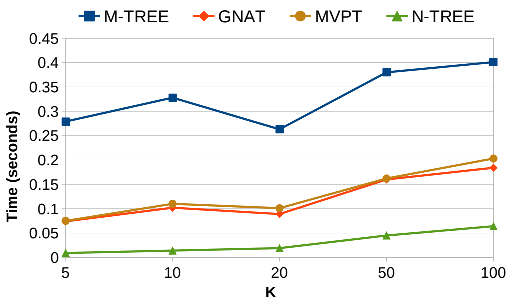


(e) Words

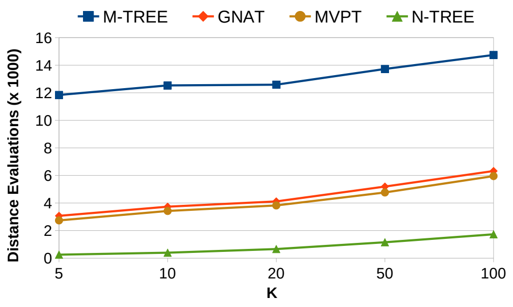


(f) Words

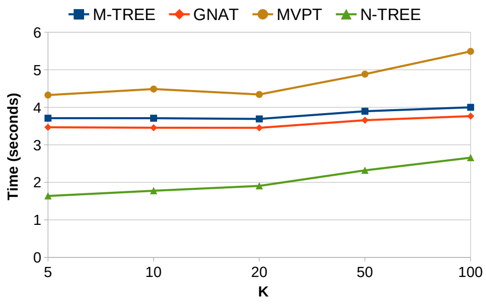
Figure 23: Comparison using range search with low to high radius



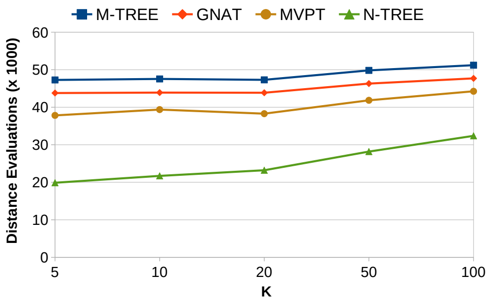
(a) Trips



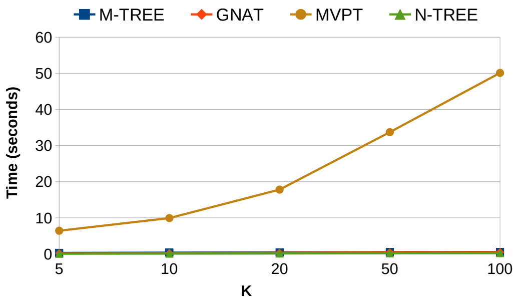
(b) Trips



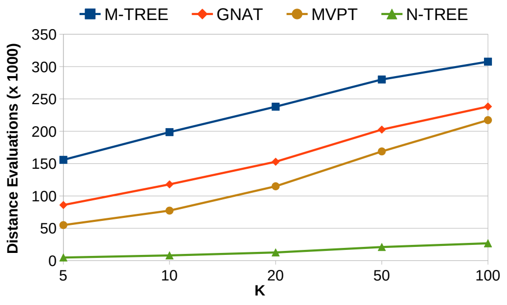
(c) X-Rays



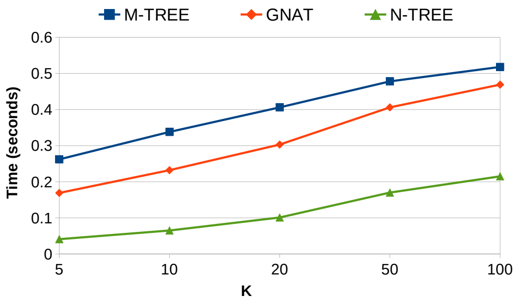
(d) X-Rays



(e) Words



(f) Words



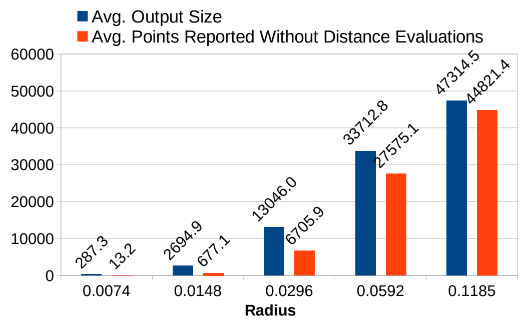
(g) Figure 24(e) without MVPT

Figure 24: Comparison using kNN search

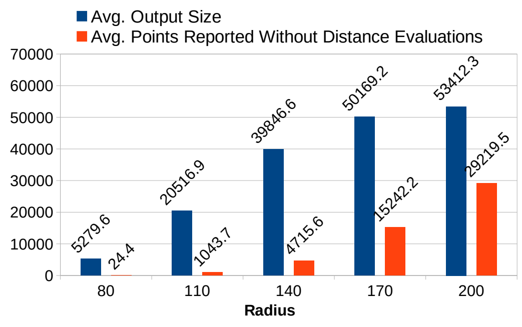
of distance evaluations keeps increasing with increase in search radius. Normally the number of distance evaluations should increase with the increase in search radius, since the number of data points falling within the search radius also increases. But our structure efficiently uses the *radius* associated with each center c_i within a node in the N-tree to include an entire partition into the final result set, without any distance evaluation, thereby reducing the number of distance evaluations in spite of an increase in the search radius. We call this property the *U-Turn effect*, which, to the best of our knowledge, is not present in other metric indexes. Due to this property, the number of distance evaluations (along with the query time, since the query time is mostly dependent on the number of distance evaluations) in the N-tree remains similar for both very small as well as very large search radius. The effect of the efficient handling of *radius* is shown in Section 6.6.

6.6 Effect of Radius

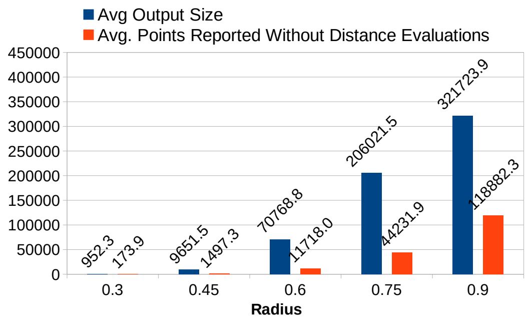
As discussed in the algorithms for range search, the radius associated with each center can be utilized to include all the points which lie within the search radius, which is the reason for the *U-Turn effect*. Thus we can include many data points without any distance evaluation at all. The more the data points are closely related to each other in a dataset, the larger is the number of points that can be included without any distance evaluations, since a lot of points lie within a partition. To emphasize this feature, we try to compare the average output size with the number of points included without any distance evaluations for all the three datasets for range search, varying the radii from low to high. The effect of this feature is best seen when the search radius is a bit large, so that a larger number of points lies within the search radius. The results are shown in Figure 25. We do not consider *kNN* search here because *kNN* search internally utilizes *range* search.



(a) Trips



(b) X-Rays



(c) Words

Figure 25: Points reported without Distance Evaluations for Range Search using low to high radius

From the figure we see that the percentage of points included without any distance evaluations (out of the average output) generally increases with the increase in search radius. In case of the *Trips* dataset, the percentage of points reported without any distance evaluations is

4.6%, 25.1%, 51.4%, 81.8% and 94.7%, respectively, for each radius. For the *X-Rays* dataset, the values are 0.5%, 5.1%, 11.8%, 30.4% and 54.7%. Finally for the *Words* dataset 18.3%, 15.5%, 16.6%, 21.5% and 37% of the total points are reported without any distance evaluations. It is evident that the percentage of points reported without any distance evaluations depends on the underlying distance distribution. If the majority of the points are close to each other, then a higher number of points is reported, since the larger the search radius, the more partitions fall within the search radius due to which more points get reported. That is why we observe that for the *Trips* dataset the best results are observed, since most of the trips are close to each other. In this case, for the largest search radius (which reports nearly 47.5K points out of 50K), nearly 94.7% of the points get reported without any distance evaluations. Since in the *X-Rays* dataset, the images are not too close to each other, only around 55% of the points are reported without any distance evaluations. The distance distribution of the *Words* dataset suggests that the dataset forms several clusters. Since most of the data points are far from each other, for the largest search radius around 37% of the points are reported without any distance evaluations. Thus it can be concluded that even with different data distributions, the N-tree performs better compared to the other structures.

7 Related Work

A significant body of previous work investigated the topic of indexing metric spaces. Chen et al. [5] conducted a comprehensive survey of such indexing techniques. These approaches can be grouped into three categories: compact-partitioning based metric indexes (CMI), pivot-based metric indexes (PMI), and hybrid metric indexes (HMI) incorporating ideas from CMI and PMI.

Compact-partitioning based metric indexes The indexes in the CMI category utilize the principle of partitioning the search space and they aim to attain this partitioning as compactly as possible. Such partitioning allows filtering out unqualified partitions during query execution. Three different kinds of partitioning techniques have been proposed, namely, ball partitioning, generalized hyperplane partitioning and hash partitioning. CMI indexes usually utilize one of these techniques, while a few use hybrid partitioning by leveraging both ball partitioning and hyperplane partitioning. The ball partitioning divides the search space into two subsets *S1* and *S2* using a spherical cut [30]. Given an arbitrarily chosen point p and radius r, all the objects that are at a distance less than or equal to r from p belong to *S1* and the rest of the objects are in *S2*. The generalized hyperplane partitioning divides the search space into two subsets *S1* and *S2* using two arbitrarily chosen reference points *p1* and *p2*. The objects are assigned to either *S1* or *S2* depending on their proximity to the reference points. The hash partitioning leverages a hash function for partitioning the search space. A popular hash function is the ρ-split function that partitions the search space into three subsets *S1*, *S2* and *S3*. This leaves out points near a particular threshold when determining membership in *S1* and *S2*, whereas *S3* includes the excluded points. This is why this technique is also known as excluded middle partitioning [29].

One of the earliest indexing approaches based on generalized hyperplane partitioning is the Bisector Tree (BST) [13]. It is a binary tree that is built recursively using two reference points, such that objects closer to the first reference point belong to the first subtree and the objects nearer to the second reference point are in the second subtree. The covering radii corresponding to each of the reference points are maintained in the nodes. The Monotonous BST (MBST) [19] is a follow-up work of BST. It is based on the idea that every internal node (except for the root node) inherits one of the reference points from its parent node. The Voronoi Tree (VT) [7] is another extension of BST. It attempts to pack the objects more compactly by decreasing the covering radii while moving downwards in the tree. The Generalized Hyperplane Tree (GHT) proposed by [25] is based on similar concepts as the BST. However, instead of using covering radii for pruning, it uses a criterion involving the hyperplane between the reference points to decide which subtrees to visit.

Among the ball partitioning based techniques, the M-tree [6] is perhaps the most well-known. It is a height-balanced index structure, and can support efficient external memory operations. In the M-tree, the internal nodes maintain pointers to the next level nodes, while the objects are kept at the leaf nodes. Each internal node entry also includes information regarding covering radius and parent distance. Several variants of M-tree have been proposed, such as the MM-tree [20], M^+-tree [31] and BM^+-tree [32]. The List of Clusters (LC) index [3] leverages a list of clusters in which each cluster is identified by a center and a radius. The main idea of LC is to store each object within a particular cluster whose distance to the center of that cluster is not larger than the radius. The LC index can be of two variants: fixed radius and fixed size. The Dynamic LC (DLC) [17] is an extension of LC, which is essentially a dynamic version of it.

The hash partitioning approaches include the MB^+-tree [12]. The MB^+-tree partitions the search space into two subsets using hash partitioning. Alternatively, it can also use generalized hyperplane partitioning recursively. The MB^+-tree utilizes two data structures: block tree and B^+-tree. The block tree is used to maintain partition information. For each object a key is generated by the MB^+-tree, which is indexed by the B^+-tree.

Pivot-based metric indexes (PMI) The indexing approaches in this category leverage pre-computed distance, since distance computation during query execution is an expensive operation. Sometimes, this distance pre-computation is performed by first selecting a set of pivots and then calculating the distance of every object to those pivots and storing those distances.

The Approximating and Eliminating Search Algorithm (AESA) [26] is one of the earliest approaches in the PMI category. It maintains a table storing the distances between all pairs of objects. During query processing, the pre-computed distances can be used to filter out objects. Whereas AESA can improve search performance, a key drawback is that this requires scanning the pre-computed table to find a match. Another drawback is the space requirements to store the pairwise distances. To address these limitations with AESA, several approaches were subsequently proposed. Among them, the Reduced-Overhead AESA (ROAESA) [27] sorts the pre-computed distances during the search and applies heuristics to potentially avoid unnecessary table scans.

The Linear AESA (LAESA) [9] index utilizes a fixed number of pivot points and only stores pre-computed distances from objects to those pivot points. However, selecting suitable pivots becomes an issue. This can be addressed by other techniques, such as [15], which attempts to identify pivots that are as far away from each other as possible [30].

The Extreme Pivot Table (EPT) [21] approach uses different pivot points for different objects, rather than leveraging the same set of pivot points for all objects. This is done by selecting a set of pivot groups, each of which consists of a fixed number of pivots. EPT uses a group inclusion criteria for each object based on the expected value of the distance between the object and the pivot points, and a threshold α, which aims to maximize. EPT maintains the data and pre-computed distance information in main memory. For larger datasets, CPT [16] proposes an I/O efficient approach. The objects are maintained on disk using an M-tree. The pre-computed distance table keeps pointers to the leaf node entries in the M-tree.

The Vantage Point Tree (VPT) [25] is based on partitioning a dataset into two subsets based on a vantage point or pivot. The pivot is chosen as the root node. All objects that are located within a distance from the pivot less than the median distance are kept in the left subtree, while the rest of the objects are maintained in the right subtree. The partitioning process is recursively applied to form a balanced binary tree. VPT is primarily targeted for continuous distance functions, and discrete distance functions can also be supported. Instead of using the median distance, an alternative approach is to use the mean of distances from the pivot to all objects. This approach is known as *middle point* in [4], which may perform better with high-dimensional data, however, it may result in an unbalanced tree. A dynamic variant of

VPT, called DVPT [10], was proposed that supports insertion and deletion operations. Another approach, Multi Vantage Point Tree (MVPT) [1], extends the VPT approach. It utilizes multiple pivots (typically 2 or 3) to partition each node, rather than one as with VPT. On the other hand, with MVPT the children at the lower level leverage the same pivots, whereas with VPT there are different pivots at lower levels.

The Omni-family [24] of indexes utilizes the pivot mapping technique that is used to represent objects as vectors of their distances to pivots [5]. Following this pivot mapping, the pre-computed distances with respect to the pivots are indexed using an existing external-memory index. The OmniB$^+$-tree employs a B$^+$-tree, whereas the OmniR-tree utilizes an R-tree.

Hybrid metric indexes (HMI) The approaches in the hybrid metric index category leverage both compact partitioning and pivots. The Geometric Near-Neighbor Access Tree (GNAT) [2] utilizes m pivots in each internal node. Based on the shortest distance of the objects to one of these pivots, the dataset is partitioned accordingly using generalized hyperplane partitioning (Voronoi partitioning). This process is applied recursively to build an m-ary tree. GNAT maintains pre-computed distances from the objects to their corresponding pivots.

The Evolutionary Geometric Near-Neighbor Access Tree (EGNAT) [18] is an extension of GNAT and adapted for external memory. It can support insertion and deletion operations. The EGNAT index includes two types of nodes: buckets (leaves) and gnats (internal nodes), where internal nodes are similar to internal GNAT nodes. The index construction process is carried out by recursively selecting the closest pivot for a new object until reaching the leaf level. At the leaf level, objects are inserted into the buckets, without internal structure, which can result in reduction in storage compared to GNAT.

The D-index [8] uses a combined hash partitioning and pivot mapping based approach. It utilizes several ρ-split functions, one at each level, to construct a multilevel structure. The Pivoting M-tree (PM-tree) [23] is an extension of the M-tree that utilizes pivoting to reduce metric region volumes. The PM-tree first selects a set of pivots. For each inner node in the tree, a routing entry is defined that includes an array of hyper-rings. Each hyper-ring is the smallest interval encompassing distances between the pivot and each of the objects stored in leaves of the subtree. Each leaf node in the PM-tree maintains an array of pivot distances.

Discussion Our approach, the N-tree, can be considered as a hybrid approach. It leverages both compact partitioning and pivoting techniques. The N-tree uses generalized hyperplane partitioning with many centers (Voronoi partitioning). Within each node, it maintains all pair-wise distances between centers as well as distances to the two pivot elements. The N-tree uses various pruning criteria during query processing.

As explained in more detail in the introduction, the approach that appears to be most closest to the N-tree is GNAT which also uses Voronoi partitioning. In the N-tree, this is combined with ideas also present in AESA to precompute all distances between centers and do iterative pruning based on sequential distance evaluations to the query point. However, in contrast to AESA this is not done globally, but per node.

8 Conclusions

We have presented the N-tree, a new index structure for metric search based on a hierarchical Voronoi partitioning of metric space. Its second main feature is the use of pre-computed distances between all elements of a node for pruning in range search and kNN search. In our experimental evaluation, the N-tree shows excellent results for range queries and kNN queries in comparison to state-of-the-art structures; for kNN queries it clearly outperforms the competitors we have evaluated.

The N-tree has a property that we did not observe in other index structures: with increasing query radius for range queries, at some point the number of distance evaluations begins to *decrease* so that for very large query radii very few distances evaluations are needed (the U-turn effect).

In this paper we have examined the N-tree, first of all, as a main memory index structure. Due to the fact that the performance is equally good for large node sizes, we expect it to be also well suited as an external index. An experimental evaluation of this case is a subject for future work.

In designing the index structure, we have observed a strong duality between the use of a Voronoi partitioning in structuring an index and for parallel/distributed computation. Based on the range distribution property, a similarity join can easily be implemented in a distributed manner. One possibility is to implement the join in each partition as a nested loop join using an index such as the N-tree. Partitioning the data set can be done in parallel and efficiently using the *closestCenter* algorithm of the N-tree. This is another area for future work.

Whereas the experimental evaluation and comparison with other structures was based on a stand-alone implementation in Java, an implementation within the DBMS SECONDO is also freely available for experiments and practical use.

Acknowledgement Thanks to Catherine Higgins for preparing the New York trips data set and her support in the early stages of this project.

References

[1] Tolga Bozkaya and Meral Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, sep 1999.

[2] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, page 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[3] Edgar Chávez and Gonzalo Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, pages 75–86, Washington, DC, USA, 2000. IEEE Computer Society.

[4] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, sep 2001.

[5] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. Indexing metric spaces for exact similarity search. *ACM Comput. Surv.*, may 2022.

[6] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann.

[7] Frank Dehne and Hartmut Noltemeier. *Voronoi Trees and Clustering Problems*, page 185–194. Springer-Verlag, Berlin, Heidelberg, 1988.

[8] Vlastislav Dohnal. An access structure for similarity search in metric spaces. In *Current Trends in Database Technology - EDBT 2004 Workshops, EDBT 2004 Workshops PhD, DataX, PIM, P2P&DB, and ClustWeb, Heraklion, Crete, Greece, March 14-18, 2004, Revised Selected Papers*, volume 3268 of *Lecture Notes in Computer Science*, pages 133–143, Berlin, Heidelberg, 2004. Springer.

[9] Karina Figueroa, Edgar Chavez, Gonzalo Navarro, and Rodrigo Paredes. Speeding up spatial approximation search in metric spaces. *ACM J. Exp. Algorithmics*, 14, jan 2010.

[10] Ada Wai-Chee Fu, Polly Mei-Shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9:154–173, 2000.

[11] Ralf Hartmut Güting, Thomas Behr, and Jan Kristof Nidzwetzki. Distributed arrays: an algebra for generic distributed query processing. *Distributed Parallel Databases*, 39(4):1009–1064, 2021.

[12] Masahiro Ishikawa, Hanxiong Chen, Kazutaka Furuse, Jeffrey Xu Yu, and Nobuo Ohbo. Mb+tree: A dynamically updatable metric index for similarity searches. In *Web-Age Information Management, First International Conference, WAIM 2000, Shanghai, China, June 21-23, 2000, Proceedings*, volume 1846 of *Lecture Notes in Computer Science*, pages 356–373, Berlin, Heidelberg, 2000. Springer.

[13] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, SE-9:631–634, 1983.

[14] Maria Luisa Micó, Jose Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recogn. Lett.*, 17(7):731–739, jun 1996.

[15] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, 1994.

[16] Juraj Mosko, Jakub Lokoc, and Tomás Skopal. Clustered pivot tables for i/o-optimized similarity search. In *Fourth International Conference on Similarity Search and Applications, SISAP 2011, Lipari Island, Italy, June 30 - July 01, 2011*, pages 17–24, New York, NY, USA, 2011. ACM.

[17] Gonzalo Navarro and Nora Reyes. New dynamic metric indices for secondary memory. *Inf. Syst.*, 59(C):48–78, jul 2016.

[18] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Inf. Syst.*, 36(4):734–747, jun 2011.

[19] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. Monotonous bisector* trees - A tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, volume 594 of *Lecture Notes in Computer Science*, pages 186–203, Berlin, Heidelberg, 1992. Springer.

[20] Ives Rene Venturini Pola, Caetano Traina Jr., and Agma J. M. Traina. The mm-tree: A memory-based metric tree without overlap between nodes. In *Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29-October 3, 2007, Proceedings*, volume 4690 of *Lecture Notes in Computer Science*, pages 157–171, Berlin, Heidelberg, 2007. Springer.

[21] Guillermo Ruiz, Francisco Santoyo, Edgar Chávez, Karina Figueroa, and Eric Sadit Tellez. Extreme pivots for faster metric indexes. In *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 115–126, Berlin, Heidelberg, 2013. Springer.

[22] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *Proc. VLDB Endow.*, 7(12):1059–1070, 2014.

[23] Tomás Skopal, Jaroslav Pokorný, and Václav Snásel. Pm-tree: Pivoting metric tree for similarity search in multimedia databases. In *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Local Proceeding*, pages 803–815, Berlin, Heidelberg, 2004. Springer.

[24] Caetano Traina, Roberto F. Filho, Agma J. Traina, Marcos R. Vieira, and Christos Faloutsos. The omni-family of all-purpose access methods: A simple and effective way to make similarity search more efficient. *The VLDB Journal*, 16(4):483–505, oct 2007.

[25] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[26] Enrique Ruiz Vidal. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recogn. Lett.*, 4(3):145–157, jul 1986.

[27] Juan Miguel Vilar. Reducing the overhead of the aesa metric-space nearest neighbour searching algorithm. *Inf. Process. Lett.*, 56(5):265–271, dec 1995.

[28] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M. Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 3462–3471, Washington, DC, USA, 2017. IEEE Computer Society.

[29] Peter Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge: Near Neighbor Searches (ALENEX)*, Berlin, Heidelberg, 09 1999. Springer.

[30] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, New York, NY, USA, 2006.

[31] Xiangmin Zhou, Guoren Wang, Jeffrey Xu Yu, and Ge Yu. M+-tree : A new dynamical multidimensional index for metric spaces. In *Database Technologies 2003, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003*, volume 17 of *CRPIT*, pages 161–168, Sydney, Australia, 2003. Australian Computer Society.

[32] Xiangmin Zhou, Guoren Wang, Xiaofang Zhou, and Ge Yu. Bm^{+}-tree: A hyperplane-based index method for high-dimensional metric spaces. In *Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings*, volume 3453 of *Lecture Notes in Computer Science*, pages 398–409, New York, NY, USA, 2005. Springer.