2-2022

Storing Data Once in M-trees and PM-trees: Revisiting the Building Principles of Metric Access Methods

Humberto Razente
Universidade Federal de Uberlândia, humberto.razente@ufu.br

Maria Camila N. Barioni
Federal University of Uberlˆandia, camila.barioni@ufu.br

Yasin N. Silva
Loyola University Chicago, ysilva1@luc.edu

Recommended Citation

Razente, Humberto; Barioni, Maria Camila N.; and Silva, Yasin N.. Storing Data Once in M-trees and PM-trees: Revisiting the Building Principles of Metric Access Methods. Information Systems, 104, : 1-13, 2022. Retrieved from Loyola eCommons, Computer Science: Faculty Publications and Other Works, http://dx.doi.org/10.1016/j.is.2021.101896

Storing Data Once in M-trees and PM-trees: Revisiting the Building Principles of Metric Access Methods *

Humberto Razente[1], Maria Camila N. Barioni[1], Yasin N. Silva[2]

[1] *Universidade Federal de Uberlândia (UFU)*
Faculdade de Computação
Campus Santa Mônica, Uberlândia, MG, Brazil
{*humberto.razente,camila.barioni*}*@ufu.br*

[2] *Arizona State University (ASU)*
School of Mathematical and Natural Sciences
Glendale, Arizona, USA
ysilva@asu.edu

Abstract

Since the introduction of the M-tree, a fundamental tree-based data structure for indexing multi-dimensional information, several structural enhancements have been proposed. One of the most effective ones is the use of additional global pivots that resulted in the PM-tree. These two indexing structures, however, can store the same data element in multiple nodes. In this article, we revisit both the M-tree and the PM-tree to propose a new construction algorithm that stores data elements only once in the tree hierarchies. The main challenge to accomplish this, is to properly select data elements when an inner node split is needed. To address it, we propose an approach based on the use of aggregate nearest neighbor queries. The new algorithms enable building the search result set as data elements are evaluated for pruning during traversal, allowing faster retrieval of k-nearest neighbors and range searches. We conducted an extensive set of experiments with different real datasets. The results show that that our proposed algorithms have considerably superior performance when compared with the standard M-tree and PM-tree.

*This work has been supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Finance Code 001, and by the Brazilian National Council for Scientific and Technological Development (CNPq).

Keywords: Metric access methods, Ball-partitioning indexing, M-tree, PM-tree, k-nearest neighbor query, range query

1. Introduction

Since the development of the B+tree, focused on secondary storage, even though some keys are used as routing information in the inner nodes, all keys are stored in the leaf nodes of tree-based methods. The motivation for this tree organization is that the space in an inner node is so valuable that it is better to use it to partition the data than to store the location of the data represented by that key [1]. Moreover, the leaves are connected and form a sequential set, which is of great interest when searching for a range of keys based on the total ordering relation.

The M-tree stores all data elements in the leaf nodes, although a few are also stored in the inner nodes for routing purposes. The leaves are not interconnected. For indexes built for similarity search, rather than having numeric or small text keys, metric data elements may occupy up to a few kilobytes. Although the purpose of an inner node is to allow data partitioning, storing an 8-byte numeric identifier with each entry may result in a minimal disturbance in the indexing structure and allows retrieving the full tuples after retrieving the metric data records in a range or k-nearest neighbor query.

In this article, we propose not to duplicate elements promoted during node splits. Instead, the two elements promoted to the upper level during a split are removed from their original nodes. This algorithm is easily defined for leaf nodes by removing the elements selected for promotion. When splitting an inner node, however, it is not possible to remove a local pivot that needs to be promoted, as it represents a branch. Instead, we have the opportunity to select a better pivot to be promoted from a leaf node. We propose the use of an aggregate nearest query to find an element that better minimizes the covering radius considering the set of ball entries (each composed of an element and a radius) that form an inner node.

This article extends the concepts introduced in [2] presenting a more detailed description of the new Metric Access Method (MAM) indexing algorithms including k-nearest neighbors and range algorithms, and the results obtained with a new set of experiments to evaluate construction and querying times considering: different construction parameters, query cardinality, and

dataset scalability. The main contributions of this article can be summarized as follows:

- a new indexing approach for M-tree and PM-tree that allows building more efficient indexes for k-nearest neighbors and range querying operations;
- k-nearest neighbor and range query algorithms for the new indexing structures;
- an optimization approach, based on aggregate similarity query algorithms, that allows finding suitable elements to be promoted during inner node splits;
- the definition of the minimum aggregate radius property that allows optimizing aggregate range queries;
- an extensive experimental evaluation and discussion to asses diverse aspects of the new indexing algorithms on M-trees and PM-trees.

The remaining part of the article is organized as follows. In Section 2, we describe the fundamental concepts. Section 3 details the new construction algorithms. Section 4 discusses the experimental results and Section 5 presents the final considerations.

2. Fundamental Concepts

A metric space is a pair $< \mathbb{S}, \delta() >$, where $\mathbb{S}$ is a data domain, and $\delta()$ is a distance function that satisfies the following axioms for any elements x, y, and z in $\mathbb{S}$: $\delta(x, x) = 0$ (*identity*); $\delta(x, y) = \delta(y, x)$ (*symmetry*); $0 \leq \delta(x, y) < \infty$ (*non-negativity*); and $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$ (*triangle inequality*). The triangle inequality is used to determine if a ball defined by a data element and a radius covers another element or intersects another ball. It is employed to avoid reading data and computing distances from data elements that the user is not looking for.

Existing metric access methods are divided into compact partitioning techniques and pivot-based techniques [3]. M-tree [4] is the landmark of compact partitioning techniques. It is a ball-partitioning method that results in a tree hierarchy composed of inner and leaf nodes, built in a bottom-up fashion, such as the B+tree. Each inner node contains a set of entries of

the form $< pivot, radius >$, where *pivot* is a data element and *radius* is the branch covering radius (this radius should be greater than the distance between the pivot and any of the data elements under the pivot). Each entry defines a ball that covers all the data elements in the tree branch it represents. The leaf nodes contain all data elements.

Algorithm 1 describes the insertion of an element e. The M-tree is created with an empty leaf node initially defined as the root. In the case of leaf overflow, a split algorithm is used to create a new node and to distribute the elements between them. Each node also promotes one element to the upper level, which stores it together with the associated coverage radius. The upper levels are updated recursively, if necessary. This process guarantees the structure is always balanced. After the first split, the insertion process starts finding out a path from the root to a leaf. Space is not exclusively partitioned as node coverage of different nodes may intersect.

Algorithm 1: INSERT

Input: M-tree root node *node*, new element e

```
1  if node.type is inner then
2      if e is covered by at least one node.entry_i then
3          select the node.entry_i that minimizes the distance from e
4      else
5          select the node.entry_i that minimizes the radius increase
6      INSERT(node.entry_i, e) // recursion
7      if status returned from recursion is promotion then
8          update node.entry_i.pivot and insert new entry
9  else if node.type is leaf then
10     if node is not full then
11         node.add(e)
12     else
13         split(node + e) and promote routing elements
```

Observe that the insertion process can result in an inner node overflow and thus, an inner node split. After the split, a copy of each promoted element is sent to the upper level. These promoted elements will be used as routing elements (the first will replace the previous routing element, and the second will be inserted as a new inner entry). The work in [4] proposes the use of *m_RAD*, an algorithm that finds a pair of pivots that split a node by

minimizing the sum of the covering radii. Its time complexity is $O(n^2)$, where n is the number of elements in a node. Another interesting split strategy [5] computes a minimum spanning tree and removes its longest edge to split a node with time complexity of $O(n \cdot log\ n)$.

Several related contributions have been proposed to enhance the M-tree's performance. Some of them explored the reorganization of the trees [5], the reinsertion of elements [6] and the use of short-term memories during the construction [7]. Other contributions aimed to explore metric properties to propose interesting new data structures, such as the Dynamic Spatial Approximation Trees [8], iDistance [9], GroupSim [10], Omni-R [11], M-Index [12], and PM-tree [13, 14, 15]. All these contributions demonstrated high performance levels considering different scenarios. A comprehensive review of pivot-based methods can be found at [3] and an extensive review of the area can be found at [16].

Pivot-based techniques consider a static dataset $S \subseteq \mathbb{S}$ to find a constant set of pivots. A naive pivot selection algorithm is to randomly select n elements as pivots. Finding the optimal pivot set takes polynomial time, and consequently it is, in many cases, impractical. Several heuristics with linear time complexity were proposed recently, such as Maximum of Minimum Distances (MMD) and Maximum of Sum of Distances (MSD) [17]. These heuristics start by randomly selecting the first pivot and then they add pivots incrementally, maximizing $p_i = argmax_{s \in S - \{p_1, \ldots, p_{i-1}\}} \min_{j=1}^{i-1} \delta(s, p_j)$ (MMD) or $p_i = argmax_{s \in S - \{p_1, \ldots, p_{i-1}\}} \sum_{j=1}^{i-1} \delta(s, p_j)$ (MSD) regarding the previously selected pivots.

In addition to creating hierarchies based on local minimum bound rectangles (spatial) or pivot (metric) representations, Omni-R [11] proposed the use of a set of static global pivots to enhance pruning balls on R-trees, while PM-tree [14] proposed the use of a set of static global pivots to dynamically store cut-region information on M-trees. Cut-regions allows better pruning as they are able to exclude dead regions inside balls. The cut-region concepts and algorithms were later formalized in [18, 15]. The PM-tree is the state-of-the-art indexing structure of dynamic ball-partitioning metric access methods.

The PM-tree insert algorithm is based on the M-tree insert (Algorithm 1). Additionally, cut-regions are computed and stored in inner nodes. Also, when a node is split, the cut-regions may be updated in the upper nodes, as well as the routing elements. Figure 1 illustrates the cut-region of a PM-tree

node. The cut-region is composed of the minimum and maximum distances from each node's element to each of the global pivots. In this example, one can notice that although the query q with radius r_q overlaps the ball defined by o and r_0, the cut-region allows excluding the dead space, thus enhancing the pruning ability.

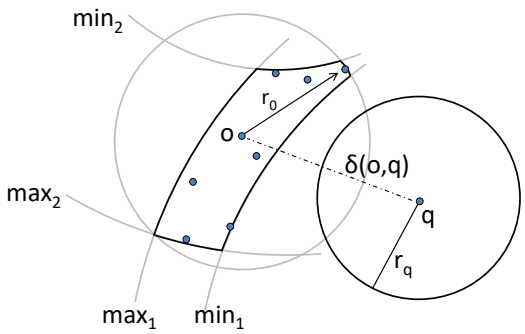

Figure 1: PM-tree: overlap of an inner node and a ball region. The ball defined by pivot o and distance r_0 overlaps the ball defined by q and r_q. However, the cut-region allows excluding the dead space, thus enhancing the pruning ability. Adapted from [15].

3. New M-tree and PM-tree algorithms

A metric index is a metric access method that organizes the elements s_i of a dataset $S \subseteq \mathbb{S}$ using a metric $\delta()$. Both M-tree and PM-tree indexes support dynamic insertion of elements and optimization opportunities for the execution of similarity queries. Among these queries are range and k-nearest neighbor queries from a query element $s_q \in \mathbb{S}$ based on a limit ℓ that is either the radius (τ) or the number of neighbors (k). The construction of a PM-tree also considers a set of static global pivots $P \subseteq \mathbb{S}$. In the next sections, we refer to the standard algorithms as M-tree and PM-tree and the proposed algorithms as M#tree and PM#tree.

To store data elements once in the hierarchy of an M-tree or a PM-tree, we must propose a new insertion algorithm. When inserting a new element, the algorithm starts from the root node and searches for a leaf to hold the element, employing a heuristic to choose a suitable branch to follow. If the node has enough space, it inserts the element, and the coverage radius may be updated in the upper levels, if necessary. This process is similar to the steps performed in the original M-tree and PM-tree insertion algorithm. However,

in the case of a leaf overflow in M#tree or PM#tree, our algorithm removes the promoted element from the leaf node, as it will store the element in the upper level.

When inserting the first elements in a new tree, the leaf node is also the root node. In the case of an overflow, the algorithm creates a new leaf and distributes the entries between them. It also creates a new inner node (the new root node) that receives the pivots promoted from the pair of leaves. For every leaf split after the first overflow, there will be a prior pivot that was used to represent the leaf. Instead of maintaining the original pivot, we propose selecting a new one, allowing the process to find an element that better represents the portion of the data that remained in the node (the algorithm assigned some of the elements of the original node to the newly created node). Since a new pivot is selected/removed from the leaf, and promoted, the algorithm reinserts the old pivot so that it can find a suitable leaf node (reinsertion is necessary as there is no other copy of the element in the index). For the new node, the algorithm selects/removes and promotes a pivot. The insertion algorithm is recursive and applies the described steps in a bottom-up fashion. A node split may promote an element that may cause another split in the upper level.

A key challenge in the insertion algorithm is the selection of the pair of pivots when an inner node needs to be split. When splitting an inner node, selecting an element to be promoted and removing it from the node is not possible, as each element is a pivot that represents a branch. Our proposed algorithm employs the aggregate nearest neighbor query to solve this issue. The goal is to find an element that minimizes the covering radius considering the set of ball entries (each composed of an element and a covering radius) that form an inner node. The algorithm searches the branch for this element and removes it from its leaf. The aggregate nearest neighbor query allows finding, for instance, the element in the branch that minimizes the sum of distances to the set of ball pivots, among other aggregation functions. Then, this element is promoted and stored in the upper level. This strategy can be applied to both M-tree and PM-tree insertion algorithms. Observe that the aggregate query does not start at the index root node, instead it traverses from the inner node being split.

Algorithm 2 describes the insertion of an element e in M#tree. The differences are lines 8 to 14, where after an inner node split, the two elements returned by the aggregate nearest neighbor queries rooted at both split nodes will be set as routing elements, being one of them a replacement. Thus the

reinsertion of the previous routing element is needed. For the PM#tree, it is also necessary to update the cut-regions when a node is split.

Algorithm 2: INSERT

Input: M#tree root node *node*, new element *e*

```
1  if node.type is inner then
2      if e is covered by at least one node.entry_i then
3          select the node.entry_i that minimizes the distance from e
4      else
5          select the node.entry_i that minimizes the radius increase
6      INSERT(node.entry_i, e) // recursion
7      if status returned from recursion is promotion then
8          if promoted from inner split then
9              aggregate nearest neighbor(node.entry_i)
10             aggregate nearest neighbor(new node)
11             reinsert node.entry_i.pivot element
12             update node.entry_i.pivot and insert new entry
13         else if promoted from leaf split then
14             update node.entry_i.pivot and insert new entry
15 else if node.type is leaf then
16     if node is not full then
17         node.add(e)
18     else
19         split(node + e) and promote routing elements
```

When splitting a node, the new methods can employ m_RAD, MST (M#tree and PM#tree), or GrowthOfCutRegionExtension (PM#tree). For leaf nodes, the complexity is the same as proposed by the original works on M-tree and PM-tree. For an inner node split, after the node is split into two nodes, an aggregate nearest neighbor is run for each of them to find the elements that will be removed from the leaves and promoted to the upper levels. One of them will replace the previous routing element, which will need to be reinserted in the index. In short, the complexity of the split of an inner node is the complexity of the clustering method (m_RAD, MST, GrowthOfCutRegionExtension, etc.) plus the complexity of two aggregate nearest neighbor queries (optimized as described in Section 3.1.1), and one reinsertion in the branch being split.

Figure 2 illustrates the selection of promoted pivots when an inner node is split. It represents the set of balls stored in the node. In (a), the node entries $\{ < s_1, r_1 >, < s_2, r_2 >, < s_3, r_3 > \}$ represent the pivots and the covering radii, each one covering a branch of the tree. In the standard M-tree (b), s_1 is promoted to the upper level with radius r_p, as among the options $\{s_1, s_2, s_3\}$, the element s_1 results in the minimization of r_p that covers all entries in this node. The new strategy (c) searches downward to find the aggregate first-nearest neighbor p with respect to query elements $Q = \{s_1, s_2, s_3\}$ by minimizing an aggregation of distances, such as the sum or the mean square distance.

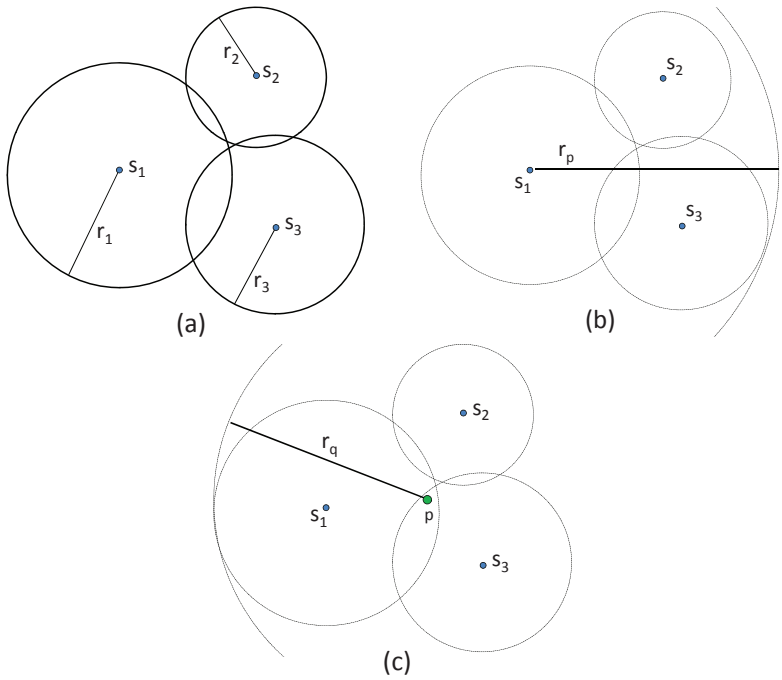

Figure 2: M-tree inner node representation. (a) Node entries $\{ < s_1, r_1 >, < s_2, r_2 >, < s_3, r_3 > \}$. (b) in the standard M-tree, s_1 is promoted to the upper level with radius r_p: among the options $\{s_1, s_2, s_3\}$, promoting s_1 results in the minimization of r_p. (c) New strategy: search downward to find the aggregate first-nearest neighbor p with respect to query elements $Q = \{s_1, s_2, s_3\}$, remove it from its leaf and promote it with radius r_q.

Observe that the promoted radius r_p from (b) is greater than the promoted radius r_q from (c), thus (c) results in less dead space. The definition and properties of these queries are presented in Section 3.1. The optimization of the aggregate nearest neighbor query, presented in Section 3.1.1, employs

the triangle inequality to prune branches that do not overlap with the search space. Thus, it is possible to implement general (1) best-first aggregate k-nearest neighbor and (2) depth-first aggregate range query algorithms by replacing the filtering step with the lower bound of the aggregate distance function. In Section 3.2, we present new query algorithms that consider that each data element appears only once in an index.

3.1. The Aggregate Similarity Query

An aggregate similarity query [19] is a relational selection operation that retrieves the most similar elements of a dataset $S \subseteq \mathbb{S}$ to a set of query elements $Q \subseteq \mathbb{S}$ considering a similarity aggregation function $d_g()$. This function evaluates the aggregate similarity of each element $s_i \in S$ based on its similarity, measured by the metric $\delta()$, to every element $s_q \in Q$. Limits can be expressed as a similarity threshold ξ (aggregated radius) or based on a number k of elements. In this article, we present a refined version of the general algorithm.

Observe that in this algorithm, the well-known similarity range and k-nearest neighbor queries correspond to special cases of the aggregate queries, where the set of query elements has only one element $Q = \{s_q\}$ and the limit ℓ is either the range or the number of neighbors. As the set of query elements Q cardinality may be greater than one, the distances $\delta(s_i, s_q)$ from each query element $s_q \in Q$ to the element $s_i \in S$ must be aggregated. Given a distance function $\delta()$, the set of query elements Q, a dataset element s_i, and a non-zero real value $g \in \mathbb{R}^*$ ($\lim_{g \to 0} d_g = \infty$), the aggregate distance function $d_g()$ is defined by Equation 1. This equation provides several interesting cases. For instance, $g = 1$ allows finding the minimization of the sum of the distances; $g = 2$ allows finding the minimization of the mean square distance; $g = \infty$ allows finding the minimization of the maximum distance; and $g = -\infty$ allows finding the minimization of the minimum distance.

$$d_g(Q, s_i) = \sqrt[g]{\sum_{s_q \in Q} \delta(s_q, s_i)^g} \tag{1}$$

3.1.1. Lower Bounding the Aggregate Distance Function

The time complexity of a sequential scan to solve the aggregate range and aggregate k-NN is $O(n * |Q|)$ distance calculations, where n is the number of elements in the dataset and $|Q|$ is the cardinality of Q. As it is done in range queries, the triangle inequality property can be employed to discard branches

of ball-partitioning based metric access methods. Let us consider Figure 3 as an example of an aggregate range query in a 2-dimensional Euclidean space and $g = 1$. In this case, the query Q is composed of two elements (q_1, q_2). The figure shows a branch centered at s_t with covering radius r_t, and an unknown element h that minimizes $d_g()$ with respect to q_1 and q_2. The challenge in this case is to compute the lower bound aggregate similarity from centers q_1 and q_2 to h to decide if the region defined by the aggregate range $\xi = \sqrt[1]{w^1 + x^1} = w + x$ overlaps the region covered by the ball centered at s_t.

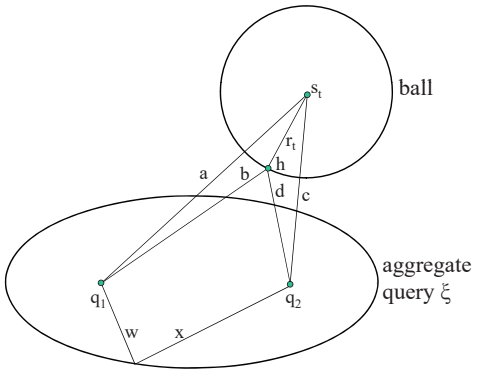

Figure 3: Ball region and aggregate range for $g = 1$, $Q = \{q_1, q_2\}$ (query), s_t is a branch representative with radius r_t.

From Figure 3, r_t is known and $\{a, c\}$ can be computed, but $\{b, d\}$ cannot. To assure that a branch centered at s_t with covering radius r_t can be pruned, we need to determine if $d_g(Q, h) = \sqrt[g]{b^g + d^g}$ is less than or equal to the limiting aggregated radius $\xi = \sqrt[g]{w^g + x^g}$ that generates the query region, i.e., if the two regions overlap. If they do not overlap, the branch centered at s_t can be pruned. From the definition of distance functions, the following triangle inequalities hold when $g = 1$ and $d_g = b + d$:

$$b \geq |a - r_t| \tag{2}$$

$$d \geq |c - r_t| \tag{3}$$

$$b + d \geq |a - r_t| + |c - r_t| \tag{4}$$

Generalizing, it can be stated for $g > 0$ that:

$$b^g \geq |a - r_t|^g \tag{5}$$

$$d^g \geq |c - r_t|^g \tag{6}$$

$$b^g + d^g \geq |a - r_t|^g + |c - r_t|^g \tag{7}$$

From Equation 7 we can build Equation 8, where Q is the set of query centers, ξ is the aggregate query radius, s_t is a branch representative, and r_t is a branch covering radius. Equation 8 enables verifying if an aggregated range overlaps a ball with no false dismissals, providing exact answers. This verification will replace the single center triangle inequality comparisons to discard branches (when the aggregation is greater than ξ) during, for instance, a depth-first traversal in a range query or in a best-first approach employed for nearest-neighbor algorithms.

$$\sqrt[g]{\sum_{s_q \in Q} \mid \delta(s_q, s_t) - r_t \mid^g} \leq \xi \tag{8}$$

3.1.2. The Minimum Aggregate Radius Property

In this section, we present an important property of aggregate range queries. Given a set Q of query centers and the aggregate function defined in Equation 1, there exists a minimum aggregate radius that defines a non-null region where a query can find elements. The minimum radius depends on g, the number of elements in Q and the pairwise distances among the elements in Q. Whenever a query with an aggregate radius less than the minimum is posed, it will result in an empty set regardless of the distribution of elements in the dataset. This property can be employed by a query plan optimizer to avoid searching the index.

For illustration purposes and without loss of generality, let us consider an aggregate range query over a set of four query centers $Q = \{q_1, q_2, q_3, q_4\}$ as shown in Figure 4. Let h_1 be the element that minimizes the similarity aggregation function for Q. In this figure, $\{a, b, c, d, e, f\}$ are the pairwise distances between all the pairs of query centers. However, as the element h_1 is not known (or may not even exist), the distances $\{v, w, x, y\}$ cannot be computed. Therefore, the minimum aggregate radius of element h_1 to the query centers is defined as the minimization of the function $d_g(Q, h_1) = \sqrt[g]{v^g + w^g + x^g + y^g}$. The minimization problem for the example of Figure 4 can be stated in the following way.

Problem definition. Minimize $d_g(Q, h_1) = \sqrt[g]{v^g + w^g + x^g + y^g}$ subject to the following rules, which are based on the triangle inequality property (considering h_1 and a different combination of two elements of Q at a time):

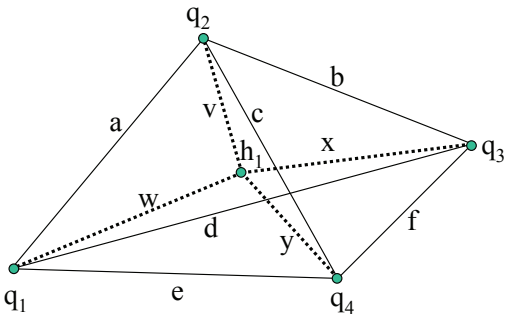

Figure 4: Minimum aggregate radius example.

$$
\begin{array}{ccccccccccccc}
\mathrm{v} & + & \mathrm{w} & & & & & & - & \mathrm{a} & \geq & 0 \\
\mathrm{v} & & & + & \mathrm{x} & & & & - & \mathrm{b} & \geq & 0 \\
\mathrm{v} & & & & & + & \mathrm{y} & & - & \mathrm{c} & \geq & 0 \\
 & & \mathrm{w} & + & \mathrm{x} & & & & - & \mathrm{d} & \geq & 0 \\
 & & \mathrm{w} & & & + & \mathrm{y} & & - & \mathrm{e} & \geq & 0 \\
 & & & & \mathrm{x} & + & \mathrm{y} & & - & \mathrm{f} & \geq & 0
\end{array}
\tag{9}
$$

For $g = 1$ (which allows minimizing $d_g(Q, h_1) = v + w + x + y$) and $g = \infty$ (which allows minimizing $d_g(Q, h_1) = max(v, w, x, y)$), this is the well-known problem of finding minima of arbitrary multidimensional functions. The problem of multidimensional minimization requires finding a point such that the scalar function $f(q_1, q_2, ..., q_n)$ takes a value lower than at any of its neighbors, where n is the number of query centers in Q. An example of multidimensional minimization algorithm is the well-known linear programming Simplex algorithm [20], which maintains $n + 1$ trial parameter vectors as the vertices of an n-dimensional simplex. In each iteration step it tries to improve the worst vertex by a simple geometrical transformation until the size of the simplex falls below a given tolerance.

Lemma. Considering $g = 1$ or $g = \infty$, the minimization function $d_g(Q, h_1)$ subjected to a set of equations derived from the triangle inequality property will always lead to a consistent system.

We show the intuition of this property using the example of Figure 4 to define Equation 10. In this example, all constants $\{a, b, c, d, e, f\}$ and variables $\{v, w, x, y\}$ involved are greater than or equal to zero, as they are distances. Therefore, since

$$
\begin{aligned}
&0 \leq v \leq max\{a, b, c\} \text{ and} \\
&0 \leq w \leq max\{a, d, e\} \text{ and} \\
&0 \leq x \leq max\{b, d, f\} \text{ and} \\
&0 \leq y \leq max\{c, e, f\},
\end{aligned}
\tag{10}
$$

there is no set of values for variables $\{v, w, x, y\}$ that generates an inconsistent system of equations. Thus, it is always possible to compute the minimum aggregate radius for a given query with a linear programming solver such as Simplex, if $g = 1$ or $g = \infty$.

3.2. Range and k-nearest Neighbor Algorithms

The range search algorithm for tree-based metric access methods employs the depth-first traversal strategy while pruning tree branches based on the triangle inequality property. Algorithm 3 presents the new algorithm for M#tree. The main difference is presented in lines 4 to 5, where as a distance is computed to allow for pruning, it is also used to check if the routing element is a candidate to be inserted in the result set. Line 6 checks if a branch can be pruned, based on the triangle inequality. Additionally, for PM#tree, line 6 checks if the query intersects the cut-region. The cut-region information of an inner entry is a vector containing the minimum and maximum distances in the branch for each of the global pivots. Checking if the range query intersects the cut-region does not require new distance calculations, as all distances were already computed.

Algorithm 4 presents the new k-nearest neighbor algorithm for M#tree. Line 1 initially adds all the entries from the root node to a priority queue ordered by their distances to the query element q. In this algorithm, as a distance is computed for an inner node entry following the priority queue (line 7), it is also used to check if the routing element is a candidate to be inserted in the result set (lines 10 to 14). This step has two advantages: first, it reduces the number of distances computed, and second, it anticipates the inclusion of an element into the result set, accelerating the *range* convergence. Lines 14 and 22 update the radius (*range*) to enable the convergence toward the range query that retrieves the k-nearest neighbors. Note that Line 8 adds a inner entry to the queue if the entry intersects the search space. Additionally, for PM#tree, line 8 also checks if the query intersects the cut-region.

Algorithm 3: RangeQ

Input: root node *node*, query element q, query radius r
Output: result set *result*

```
1  if node.type is inner then
2      for each routing element s_i in node do
3          distance = δ(q, node.s_i)
4          if distance ≤ r then
5              result.Add(distance, node.s_i)
6          if distance − node.radius_i ≤ r then
7              RangeQ(node.entry_i, q, r) // recursion
8  else if node.type is leaf then
9      for each data element s_i in node do
10         distance = δ(q, node.s_i)
11         if distance ≤ r then
12             result.Add(distance, node.s_i)
```

4. Experiments

All the access methods evaluated in the experiments were implemented in C++ using the same core data structures [21]. We run the experiments on a Linux 64-bit computer with 8 GB of main memory, Intel Core $i7{-}4770@3.40GHz$ processing unit, and 1 TB of hard disk drive.

Table 1 shows the dataset metadata. The Nasa and the Colors datasets are available in [22], and the Covertype dataset is available in [23]. The Colorstructure dataset is a subset of the CoPHir dataset [24]. CoPHir is composed of several features extracted from 100 million images from Flickr. We selected the first 8 million elements based on the *Colour Structure* extractor of CoPHir to compose the Colorstructure dataset.

The embedded dimensionality E in Table 1 represents the number of dimensions of the dataset spaces. It is well known that the existing correlations among dimensions decrease the intrinsic dimensionality – that is, the intrinsic characteristics of the data – regardless of the space where it is embedded [25]. In order to compute the intrinsic dimensionality D we employed the strategy proposed by Chávez *et al.* [26]. It defines the dimensionality $D = \mu^2/2\sigma^2$, where μ is the mean and σ is the standard deviation of the distance matrix values. In Table 1, due to the quadratic time complexity of the

Algorithm 4: NearestNeighborQ

Input: query element q, number of elements k
Output: result set *result*

```
1  queue = newPriorityQueue(root, q)
2  range = ∞
3  while not queue.Empty() do
4      node = queue.GetNextNode()
5      if node.type is inner then
6          for each routing element s_i in node do
7              distance = δ(q, node.s_i)
8              if distance − node.radius_i ≤ range then
9                  queue.Add(d, node.entry_i)
10             if distance ≤ range then
11                 result.Add(distance, node.s_i)
12                 if result.Count ≥ k then
13                     result.RemoveLast()
14                     range = result.MaxDistance()
15     else if node.type is leaf then
16         for each data element s_i in node do
17             distance = δ(q, node.s_i)
18             if distance ≤ range then
19                 result.Add(distance, node.s_i)
20                 if result.Count ≥ k then
21                     result.RemoveLast()
22                     range = result.MaxDistance()
```

method, we present an approximation of D as the average D computed from random samples of 20k elements. We employed the closest integer to D to determine the number of global pivots of the PM-trees and PM#Trees. Previous related studies used the intrinsic dimensionality for pivot selection in pivot-based indexes [27, 25] as this value tends to create fine-tuned indexes.

All the experiments employed the Euclidean distance. We employed the MaxSum algorithm [17] to select D global pivots (Table 1, # of pivots) for PM-tree and PM#Tree. All indexes were built considering the optimistic forced reinsertion strategy with parameters $maxRemoved = 5$ and $recursionDepth = 10$ [6]. M-tree and M#tree employed m_RAD for node

Table 1: Datasets.

Dataset	Cardinality	Embedded dim. E	Intrinsic dim. D	# of pivots
Nasa	40,150	20	5.18	5
Colors	112,682	112	2.75	3
Covertype	581,012	54	9.08	9
Colorstructure	8,000,000	64	7.99	8

split. PM-tree and PM#tree employed the SingleWayForCutRegions to find the path to the leaf nodes and GrowthOfCutRegionExtension as the metric for node split [15]. M#tree and PM#tree employed $g = 1$ to compute the aggregate distance when choosing the element to be promoted when an inner node split occurs.

4.1. Effect of the node size

In this experiment, we evaluate the methods regarding the size of the nodes during index construction (Section 4.1.1) and searches (Section 4.1.2).

4.1.1. Construction

For all the evaluated methods, a disk page stores a single node. As the page size grows, there may be more elements per node, potentially resulting in hierarchies with lower heights. On the other hand, node split becomes expensive due to the time complexity of the split strategies. Figure 5 presents the heights of the indexes. As shown in this figure, the proposed method resulted in indexes with the same height as the indexes of standard algorithms for most of the configurations (or at most a difference of 1), which is important to allow for a fair comparison of the querying features.

Figure 6 presents the index file sizes. The number of nodes is the ratio between the file size and the page size. PM-tree and PM#tree indexes store fewer entries in the inner nodes when compared to M-tree and M#tree due to the storage of cut-regions. Thus, the inner nodes of PM-trees and PM#trees have a smaller capacity to partition the data. Nevertheless, PM-tree and PM#tree resulted in more compact indexes when compared to M-tree and M#tree in terms of file size and the respective number of nodes. In some cases, the smaller number of entries in the inner nodes of PM-tree and PM#tree indexes generated higher tree heights (e.g., see 4KB and 12KB for Covertype, and 4KB and 8KB for Colorstructure in Figure 5).

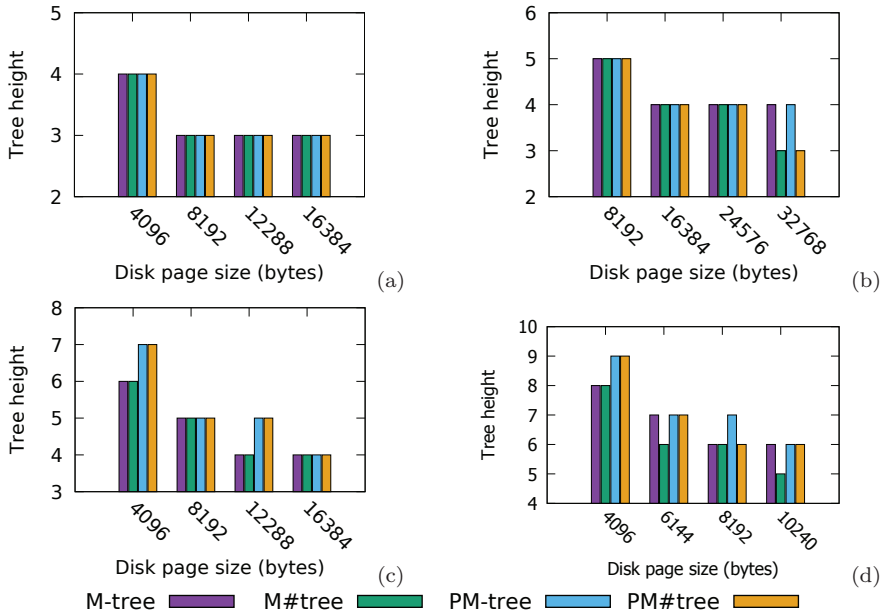

Figure 5: Effect of the node size: index height. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

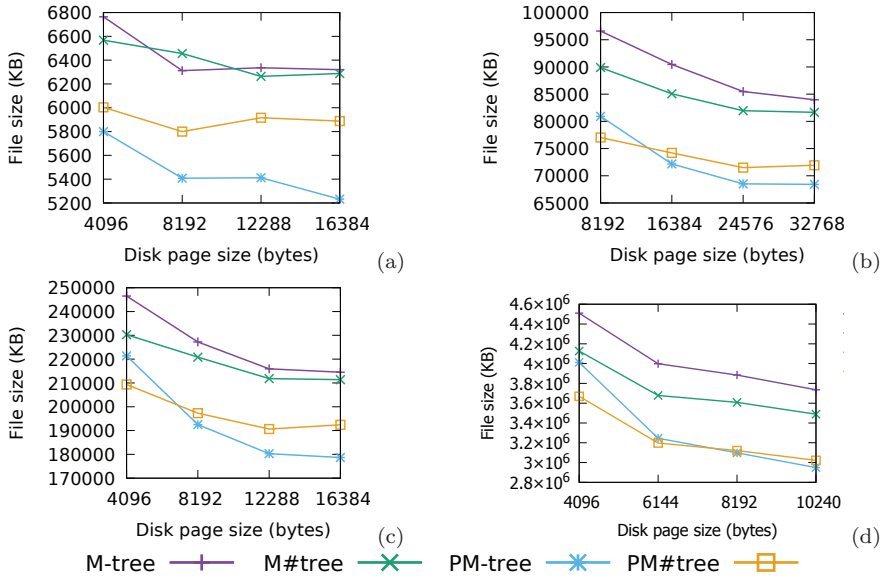

Figure 6: Effect of the node size: file size in KB. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

Figure 7 shows the index construction times. Although M#Tree's and PM#Tree's inner node split requires the execution of an aggregate nearest neighbor query, the execution of this step is usually delayed (in comparison to M-Tree and PM-Tree) due to a small reduction in the number of tree nodes resulting from data elements being stored only once. Thus, we did not observe any significant increase in index creation time. The fluctuations in Figures 6 and 7 are due to several factors, such as the resulting overlap among sibling branches, index heights, and internal fragmentation.

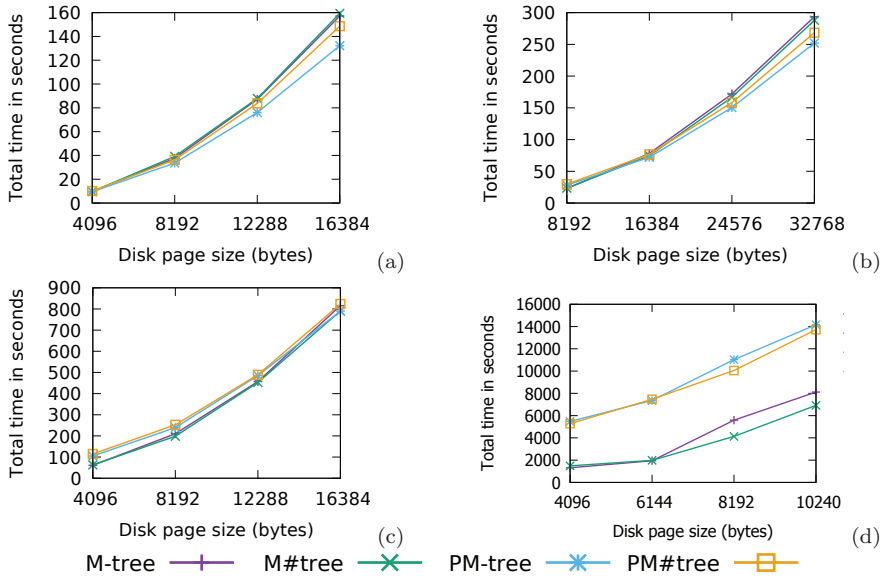

Figure 7: Effect of the node size: time to build the indexes. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

4.1.2. k-Nearest Neighbor Queries

The k-nearest neighbor query algorithm employed in the experiments uses a priority queue as proposed for R-trees in [28]. We randomly selected 100 elements of each dataset to be the query elements. Figure 8 presents the average number of disk accesses to run 100 k-nearest neighbor queries with k=10. We count all page accesses, regardless of any system/hardware cache or page fault. Moreover, as we are comparing structures built with different page sizes, we should also consider the number of bytes read to process the queries, as shown in Figure 9. As shown in Figure 8 and Figure 9, one can notice the decrease in the number of node accesses and, consequently, the

number of bytes read when comparing M-tree to M#tree and PM-tree to PM#tree for the two larger datasets. These figures also reveal that, across all of the methods, as the node size increases, the number of nodes that are effectively accessed decreases. However, this does not necessarily result in a significant decrease in the number of bytes effectively read.

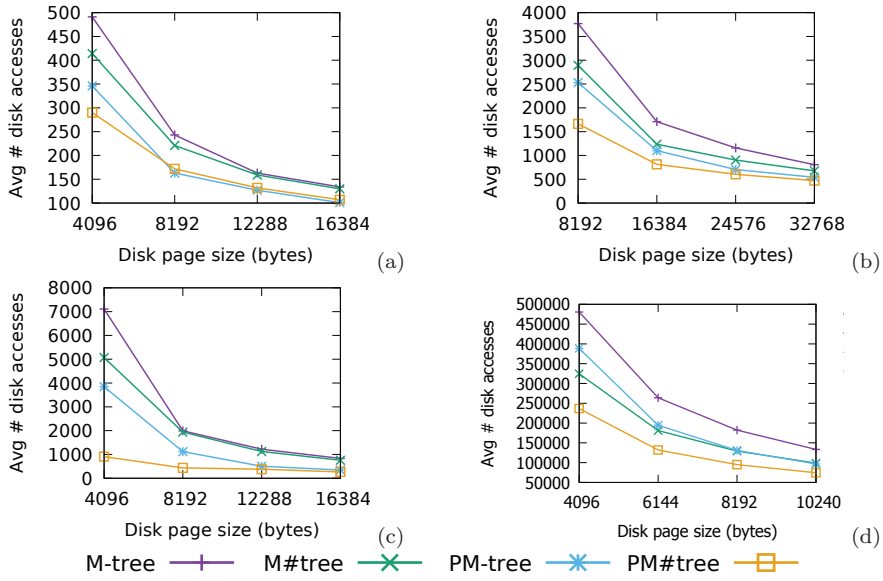

Figure 8: Effect of the node size: average disk accesses to run 100 k-nearest neighbor queries with k=10. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

Figures 10 and 11 show the average number of distance calculations and the total time to run the queries, respectively. When traversing the index, as M#tree and PM#tree compute the distance to an inner entry to decide if a branch must be visited, it uses this distance information to check if the data element is a candidate to be inserted in the result set. In the standard M-tree and PM-tree, as all elements are in the leaf nodes, the distances to the copies of the elements promoted to inner nodes are not used to consider elements to be inserted in the result set. Thus, they need to compute the distance to all unpruned elements in the leaf nodes. As shown in these figures, both the average number of distances computed and the time to execute the queries decrease as the disk page size increases. This behavior is due to several factors, such as the decrease in the tree heights (Figure 5), the reduction in the overlap among sibling branches, the number of nodes accessed (Figure 8), and the internal fragmentation, at the cost of more

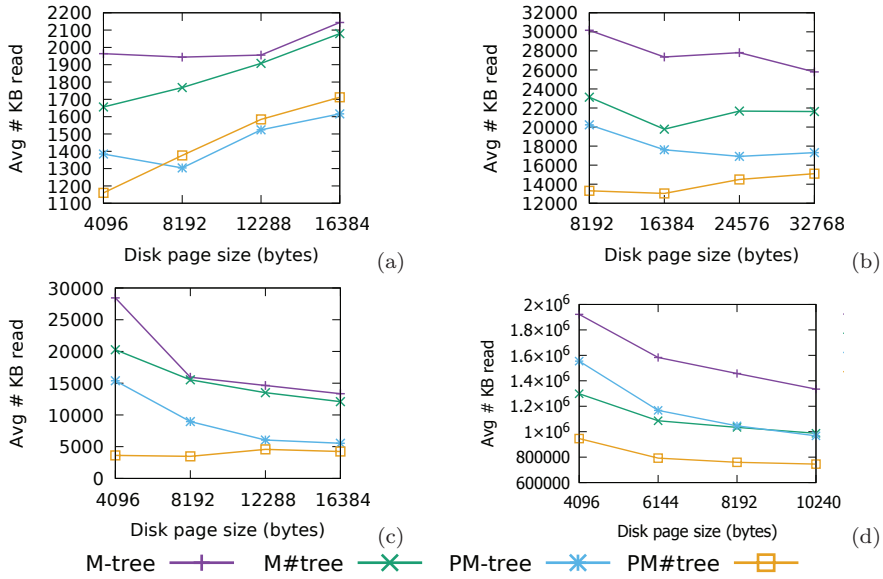

Figure 9: Node size: average KB read when running 100 k-nearest neighbor queries with k=10. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

expensive index construction. Overall, we observe that the execution times of the proposed methods are significantly better that those of the original approaches. For instance, the queries on M#tree run 26% to 44% faster than those on the M-tree, and on PM#tree from 26% to 40% faster than those on the PM-tree for the Colorstructure dataset. The performance gains when comparing M-tree *versus* M#tree and PM-tree *versus* PM#tree is detailed in Table 2.

Figures 9 and 10 are the main components of the query times presented in Figure 11. In Figure 11 we can see the query time decreases as the page size increases.

4.2. Effect of k

In this set of experiments, we evaluate k-nearest neighbor queries regarding k. We created the indexes with page sizes of 8 KB for Nasa, 16 KB for Colors, and 8 KB for both Covertype and Colorstructure (see the abscissa of Figures 5 to 11). As shown in Figure 12, when k increases, the total time spent to run the k-nearest neighbor queries increases as a logarithmic function for all the evaluated indexes. Moreover, the indexes built with the proposed methods perform significantly better than the original indexes

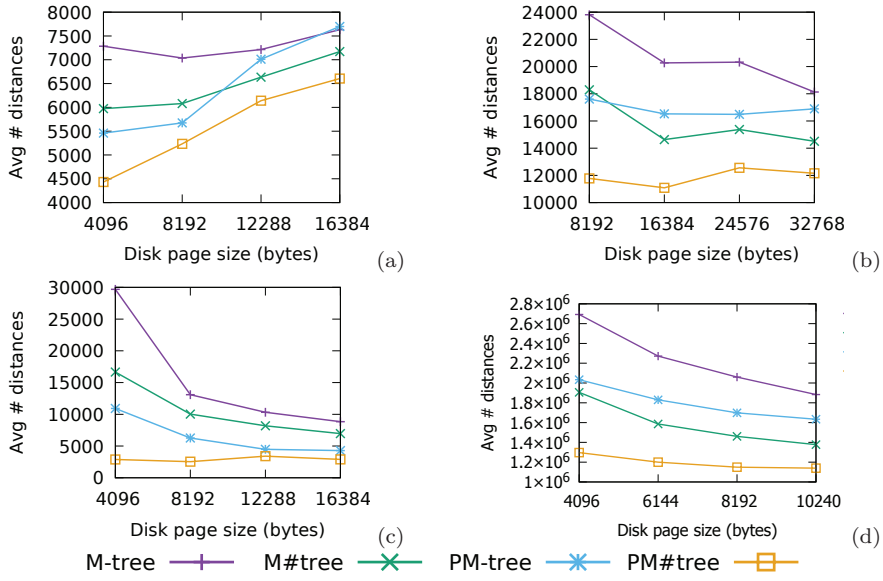

Figure 10: Effect of the node size: average distance calculations to run 100 k-nearest neighbor queries with k=10. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

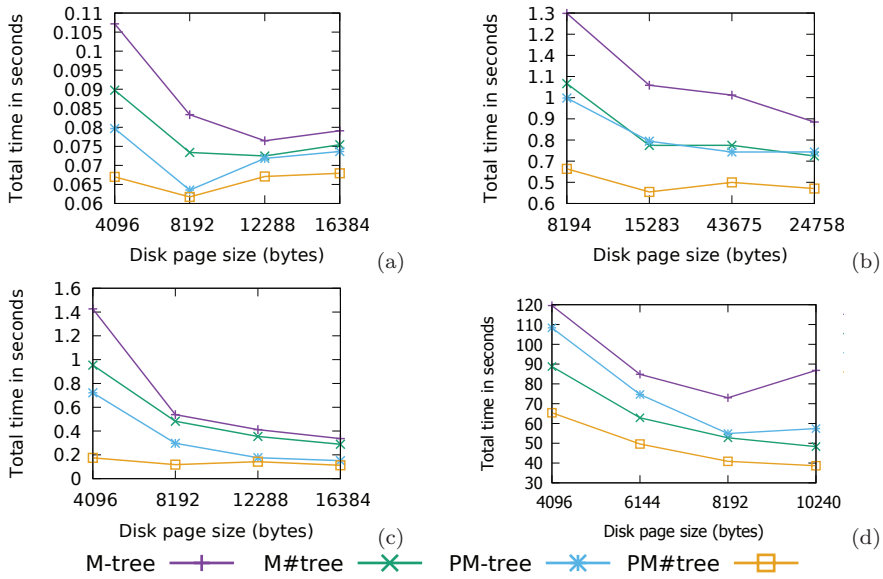

Figure 11: Effect of the node size: time to run 100 k-nearest neighbor queries with k=10. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

Table 2: Comparison of gains in M-tree/M#tree and PM-tree/PM#tree: time to run 100 k-nearest neighbor queries with k=10.

Dataset	Node size (KB)	Time in seconds					
		M-tree	M#tree	%	PM-tree	PM#tree	%
Nasa	4	0.107	0.090	-16%	0.080	0.067	-16%
	8	0.083	0.073	-12%	0.064	0.062	-3%
	12	0.076	0.072	-5%	0.072	0.067	-7%
	16	0.079	0.075	-5%	0.074	0.068	-8%
Colors	8	1.399	1.067	-24%	0.998	0.664	-34%
	16	1.059	0.775	-27%	0.794	0.555	-30%
	24	1.012	0.775	-23%	0.743	0.600	-19%
	32	0.885	0.725	-18%	0.744	0.571	-23%
Covertype	4	1.427	0.954	-33%	0.721	0.175	-76%
	8	0.538	0.482	-10%	0.297	0.118	-60%
	12	0.412	0.355	-14%	0.176	0.143	-19%
	16	0.337	0.289	-14%	0.152	0.113	-26%
Color structure	4	119.570	88.776	-26%	108.335	65.417	-40%
	6	84.846	62.870	-26%	74.651	49.637	-34%
	8	72.741	52.748	-27%	54.909	40.874	-26%
	10	86.806	48.298	-44%	57.407	38.656	-33%

across all values of k. Considering the Colorstructure dataset, for example, the time difference (performance gain) is approximately 19.5s when we compare M-tree *versus* M#tree, and approximately 15s when we compare PM-tree *versus* PM#tree.

4.3. Effect of radius

In this set of experiments, we evaluate range queries regarding the search radius. For these experiments, we created the indexes as described in Section 4.2. The query radii employed were interpolated between the distances to the 10^{th} and the 100^{th} elements retrieved for a random k-nearest neighbor query, resulting in different result set cardinalities for each dataset. Figure 13 presents the average number of elements retrieved, up to 293, 169, 926 and 6,270 for the Nasa, Covertype, Colors and Colorstructure datasets, respectively. Although it is not a requirement for range queries, our result-set data structure maintains the retrieved elements sorted by their distances to the query element. As shown in Figure 14, the indexes built with the proposed methods also perform better that the original methods as the radius increases. For example, considering the Colorstructure dataset, the time difference is approximately increases from 16.7s (r=167) to 17.9s (r=186) when

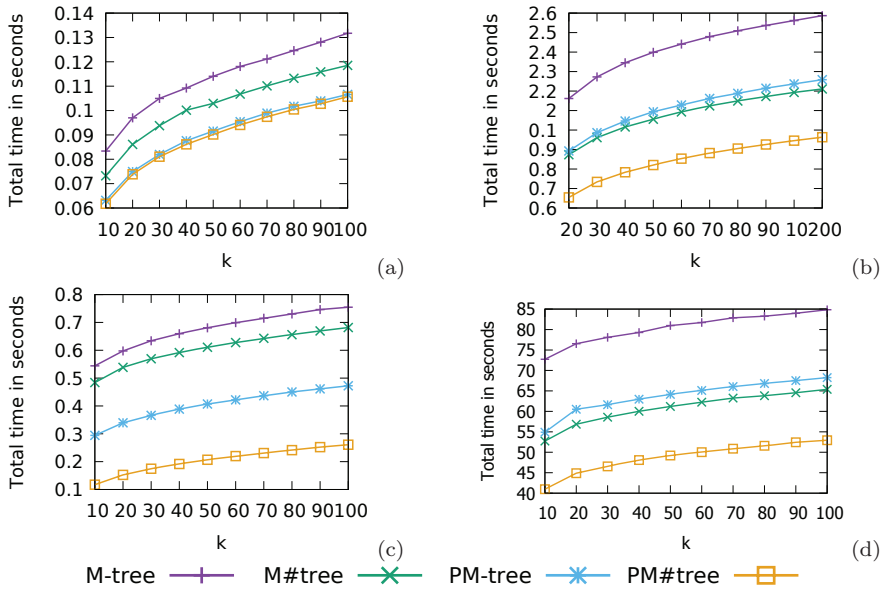

Figure 12: Effect of k: time to run 100 k-nearest neighbor queries. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

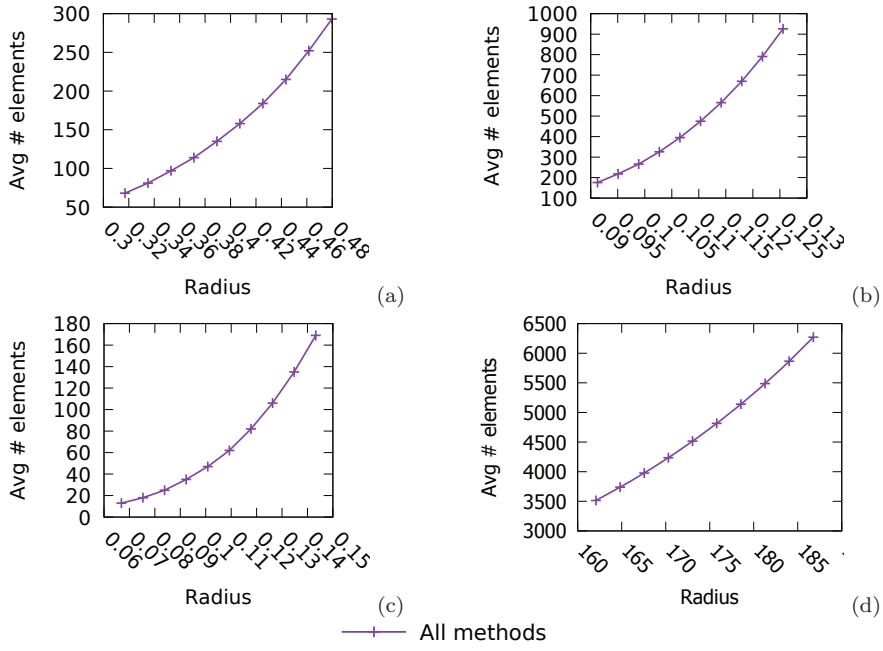

Figure 13: Effect of radius: average number of retrieved elements when running 100 range queries. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

we compare M-tree *versus* M#tree.

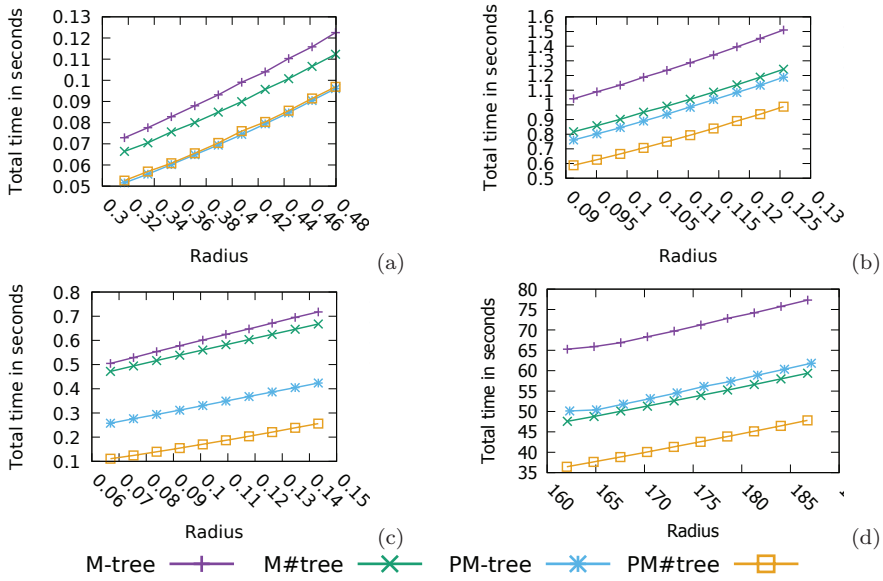

Figure 14: Effect of radius: time to run 100 range queries. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

4.4. Scalability

In this set of experiments, we evaluate the index scalability increasing the dataset in batches of 10% of the total size of each dataset. For these experiments, we defined the page size of the indexes page as described in Section 4.2. After inserting each batch, we executed the queries. Figure 15 presents the indexes' heights. As shown in this figure, in the case of Nasa and Colors, all indexes achieved their final heights after 10% and 20% of the data elements were inserted, respectively. In the case of Covertype, both PM-tree and PM#tree achieved their final heights after 20%, while for M-tree and M#tree it happened after we inserted 50% of the elements. Moreover, the experimental results obtained showed that the file sizes and the time to build the indexes grow linearly with each batch to the values presented for the respective configurations in Figures 6 and 7.

Figure 16 presents the execution time of running 100 k-nearest neighbor queries with $k = 10$ after each insertion batch. As shown in this figure, as the dataset cardinality increases, the differences between the time spent by M-tree and PM-tree compared to their improved versions M#-tree and

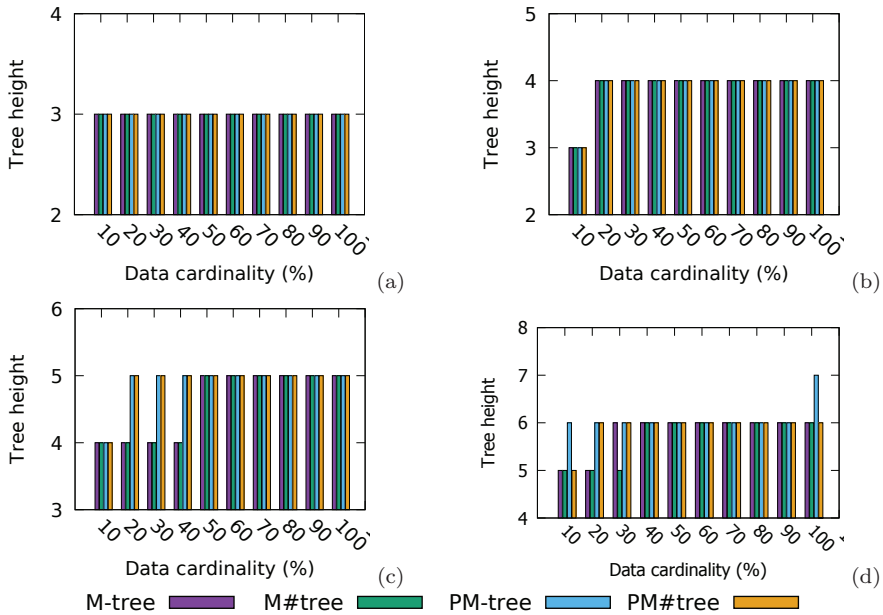

Figure 15: Scalability: index height. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

PM#tree also increase. For instance, considering the Colorstructure dataset, the time difference increases from 1.6s to 17.1s for M-tree/M#tree and from 1.1s to 16.4s for PM-tree/PM#tree.

5. Conclusion

The design of efficient and dynamic metric access methods is fundamental for many search and analysis processes based on similarity comparison operations. In this article, we present a new construction strategy for M-trees and PM-trees that does not duplicate elements during node split. To achieve this goal, we employed an aggregate k-nearest neighbor query to select the elements to be promoted during an inner node split. We also present an optimized algorithm to solve this query based on the aggregation of triangle inequality relations.

In our experiments, we thoroughly compare the standard M-tree and PM-tree against the proposed indexing methods. We empirically show that our strategy allows building indexes that significantly increase the performance of k-nearest neighbors and range queries. This performance improvement is

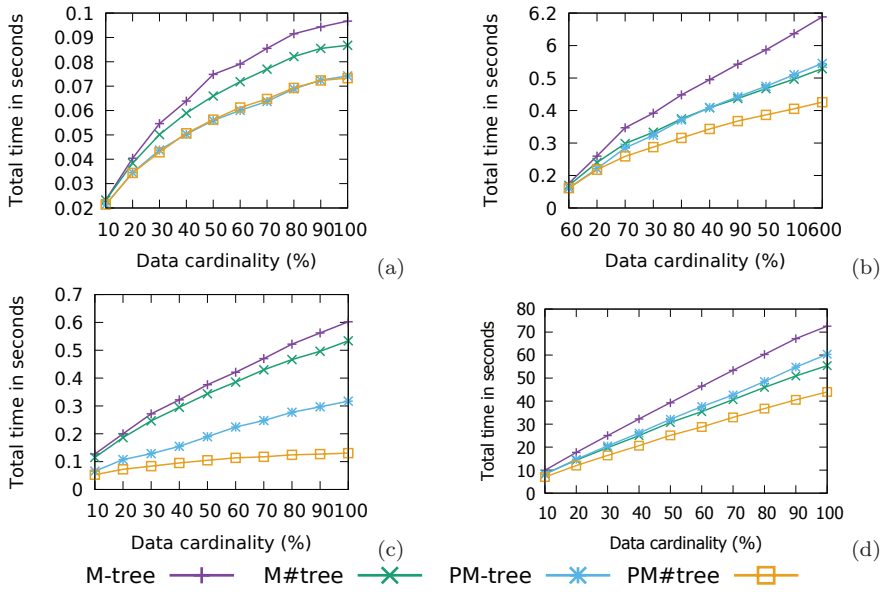

Figure 16: Scalability: time spent to run 100 k-nearest neighbor queries. (a) Nasa. (b) Colors. (c) Covertype. (d) Colorstructure.

achieved due to better data partitioning and a reduction in the number of distance calculations and disk accesses during query execution.

References

[1] D. Comer, The ubiquitous b-tree, ACM Computing Surveys 11 (1979) 121–137. doi:10.1145/356770.356776.

[2] H. L. Razente, M. C. N. Barioni, Storing data once in M-tree and PM-tree, in: Int'l Conf. on Similarity Search and Applications (SISAP), LNCS 11807, Springer, Newark, NJ, 2019, pp. 18–31. doi:10.1007/978-3-030-32047-8_2.

[3] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, K. Yang, Pivot-based metric indexing, Proc. VLDB Endowment (PVLDB) 10 (2017) 1058–1069. doi:10.14778/3115404.3115411.

[4] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: Int'l Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997, pp. 426–435.

[5] C. Traina-Jr, A. Traina, C. Faloutsos, B. Seeger, Fast indexing and visualization of metric data sets using Slim-trees, IEEE Trans. Knowl. Data Eng. 14 (2002) 244–260. doi:10.1109/69.991715.

[6] J. Lokoč, T. Skopal, On reinsertions in M-tree, in: Int'l Workshop on Similarity Search and Applications (SISAP), IEEE, Cancun, Mexico, 2008, pp. 121–128. doi:10.1109/SISAP.2008.10.

[7] H. L. Razente, R. M. S. Sousa, M. C. N. Barioni, Metric indexing assisted by short-term memories, in: Int'l Conf. on Similarity Search and Applications (SISAP), LNCS 11223, Springer, Lima, Peru, 2018, pp. 107–121. doi:10.1007/978-3-030-02224-2_9.

[8] G. Navarro, N. Reyes, New dynamic metric indices for secondary memory, Inf. Syst. 59 (2016) 48–78. doi:10.1016/j.is.2016.03.009.

[9] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, R. Zhang, idistance: An adaptive b+-tree based indexing method for nearest neighbor search, ACM Trans. Database Syst. 30 (2005) 364–397. doi:10.1145/1071610.1071612.

[10] H. Razente, R. Lima, M. C. Barioni, Similarity search through one-dimensional embeddings, in: Symp. Applied Comp. (SAC), ACM, Marrakech, Morocco, 2017, pp. 874–879. doi:10.1145/3019612.3019674.

[11] C. Traina-Jr, R. F. S. Filho, A. J. M. Traina, M. R. Vieira, C. Faloutsos, The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient, The VLDB Journal 16 (2007) 483–505. doi:10.1007/s00778-005-0178-0.

[12] D. Novak, M. Batko, P. Zezula, Metric index: An efficient and scalable solution for precise and approximate similarity search, Inf. Syst. 36 (2011) 721–733. doi:10.1016/j.is.2010.10.002.

[13] T. Skopal, J. Pokorný, V. Snásel, PM-tree: Pivoting metric tree for similarity search in multimedia databases, in: East European Conf. on Advances in Datab. Inf. Syst. (ADBIS), Budapest, Hungary, 2004.

[14] T. Skopal, J. Pokorný, V. Snásel, Nearest neighbours search using the PM-tree, in: Int'l Conf. Database Syst. Adv. Applic. (DASFAA), LNCS 3453, Springer, Beijing, 2005, pp. 803–815. doi:10.1007/11408079_73.

[15] J. Lokoč, J. Mosko, P. Cech, T. Skopal, On indexing metric spaces using cut-regions, Inf. Syst. 43 (2014) 1–19. doi:10.1016/j.is.2014.01.007.

[16] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, San Francisco, 2006.

[17] R. Socorro, L. Mico, J. Oncina, A fast pivot-based indexing algorithm for metric spaces, Pat. Recog. Let. 32 (2011) 1511–1516. doi:10.1016/j.patrec.2011.04.016.

[18] J. Lokoč, P. Čech, J. Novák, T. Skopal, Cut-region: A compact building block for hierarchical metric indexing, in: Int'l Conf. on Similarity Search and Applications (SISAP), Springer, Toronto, 2012, pp. 85–100. doi:10.1007/978-3-642-32153-5_7.

[19] H. Razente, M. C. Barioni, A. Traina, C. Faloutsos, C. Traina-Jr, A novel optimization approach to efficiently process aggregate similarity queries in metric access methods, in: Int'l Conf. Inf. Knowledge Manag. (CIKM), ACM, Napa Valley, CA, 2008, pp. 193–202. doi:10.1145/1458082.1458110.

[20] J. A. Nelder, R. Mead, A simplex method for function minimization, Computer Journal 7 (1965) 308–315.

[21] A. Traina, C. Traina-Jr, D. Kaster, E. Seraphim, F. Chino, M. Vieira, M. Bedo, W. Oliveira, The Database Group at ICMC/USP Arboretum Library, `https://bitbucket.org/gbdi/arboretum`, 2017. Accessed December, 2020.

[22] K. Figueroa, G. Navarro, E. Chaves, Metric spaces library, `http://www.sisap.org/metricspaceslibrary.html`, 2007. Accessed December, 2020.

[23] D. Dua, C. Graff, UCI Machine Learning Repository, Univ. California, Irvine, School Inf. Comp. Sciences, `http://archive.ics.uci.edu/ml`, 2017. Accessed December, 2020.

[24] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, F. Rabitti, CoPhIR: a test collection for content-based image retrieval, CoRR abs/0905.4627 (2009). URL: `http://arxiv.org/abs/0905.4627`.

[25] G. Navarro, R. Paredes, N. Reyes, C. Bustos, An empirical evaluation of intrinsic dimension estimators., Inf. Syst. 64 (2017) 206–218. doi:10.1016/j.is.2016.06.004.

[26] E. Chávez, G. Navarro, R. Baeza-Yates, J. L. Marroquín, Searching in metric spaces, ACM Comput. Surv. 33 (2001) 273–321. doi:10.1145/502807.502808.

[27] C. Traina-Jr, A. Traina, L. Wu, C. Faloutsos, Fast feature selection using fractal dimension, J. Inf. Data Manag. (JIDM) 1 (2010) 3–16.

[28] G. R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Trans. Database Syst. 24 (1999) 265–318. doi:10.1145/320248.320255.