The HeightBL Algorithm for Bulk-loading F-Onion-trees

Arthur Emanuel de Oliveira Carosia[1], Ricardo Rodrigues Ciferri[2] and Cristina Dutra de Aguiar Ciferri[1]

[1] Departamento de Ciências de Computação, Universidade de São Paulo
13.560-970, São Carlos - SP, Brazil
arthuremanuel.carosia@gmail.com, cdac@icmc.usp.br
[2] Departamento de Computação, Universidade Federal de São Carlos
13.565-905, São Carlos - SP, Brazil
ricardo@dc.ufscar.br

Abstract. The F-Onion-tree is a robust access method that slices the metric space into disjoint subspaces to provide quick indexing of complex data in the main memory. However, the F-Onion-tree only performs element-by-element insertions into its structure, i.e. it does not introduce a technique to build the index considering all elements of the dataset at once. In this article, we fill this gap. We propose the HeightBL algorithm for bulk-loading F-Onion-trees. Performance tests with real-world data with different volumes and dimensionalities showed that the index produced by the HeightBL algorithm is very compact. Compared with the element-by-element insertion, the size of the index reduced from 53.42% to 71.25%. The experiments also showed that the HeightBL algorithm significantly improved range and k-NN query processing performance. It required from 13.38% up to 99.94% less distance calculations and was from 8.57% up to 99.04% faster than the element-by-element insertion.

Categories and Subject Descriptors: Core Database Foundations and Technology [**Access methods and indexing**]: Databases

Keywords: metric access method, similarity search, bulk-loading, Onion-tree, F-Onion-tree

1. INTRODUCTION

A metric access method (MAM) aims to provide efficient access to a large number of applications that require comparison between complex data, such as images, audio and video. To improve the access to complex data, MAMs reduce the search space, leading the search to portions of the dataset where the stored elements probably have higher similarity with the searched element. The similarity measure between two elements can be expressed by a metric that becomes smaller as the elements become more similar [Hjaltason and Samet 2003]. As MAMs partition the metric space into subspaces, queries do not have to access the complete dataset.

Formally, a metric space is an ordered pair $< S, d >$, such that S is the domain of data elements and d: $S \times S \rightarrow R^+$ is the metric. For any $s_1, s_2, s_3 \in S$, the metric must have the following properties: (i) identity: $d(s_1, s_1) = 0$; (ii) symmetry: $d(s_1, s_2) = d(s_2, s_1)$; (iii) non-negativity: $d(s_1, s_2) \geq 0$; and (iv) triangular inequality: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$ [Chávez et al. 2001]. Providing a metric to enable handling complex data as a metric space helps reducing the problems derived from the curse of dimensionality, because MAMs tend to follow the dimensionality of the element represented by the data (the so-called intrinsic dimensionality) instead of the dimensionality of the space where the element is embedded (the embedded dimensionality) [Traina-Jr. et al. 2010; Pola et al. 2009]. In addition to the fact that the intrinsic dimensionality is usually lower than the embedded one [Korn

This work has been supported by the following Brazilian research agencies: FAPESP, CNPq, CAPES and FINEP. The last author is funded by the grant #2011/23904-7, São Paulo Research Foundation (FAPESP).

et al. 2001], many complex data do not have a defined dimensionality. Thus, handling datasets of low intrinsic-dimensionalities using MAMs is an interesting way to speed up similarity queries.

The two most useful types of similarity queries using MAMs are the range query and the k-NN query. Consider a query element $\mathrm{s}_q \in S$. Given a query radius r_q, the range query returns each element $s_i \in S$ that satisfies the condition $d(s_i, s_q) \leq r_q$. On the other hand, given a value $k \geq 1$, the k-NN query returns the k elements in S that are the nearest from the query element s_q.

The work on MAMs is quite extensive. An important research challenge involved is the development of main-memory MAMs, which is motivated by several factors. Due to hardware advances, the storage capacity of the main memory is increasingly growing, at the same time that its costs are lowering. Another motivation is related to the fact that main-memory MAMs are able to process similarity queries very fast, as they do not need to minimize disk accesses as disk-based MAMs do and, therefore, can provide a better partitioning of the metric space. Furthermore, main-memory MAMs are very useful to optimize subqueries when processing complex queries. In this scenario, the query optimizer of database management systems can generate a main-memory MAM on runtime to process more efficiently parts of a query or the whole query. In this article, we address main-memory MAMs.

There are few main-memory MAMs that have been proposed in the literature, such as the GH-tree [Uhlmann 1991], the GNAT [Brin 1995], the VP-tree [Yianilos 1993] and its extensions [Bozkaya and Ozsoyoglu 1997; 1999; Fu et al. 2000], the MM-tree [Pola et al. 2007] and the Onion-tree [Carélo et al. 2011]. To the best of our knowledge, the Onion-tree is the most efficient main-memory MAM to date [Carélo et al. 2011]. Thus, we focus our work on the Onion-tree.

The main characteristics of the Onion-tree are summarized as follows. It has a partitioning method that indexes complex data by dividing the metric space into several disjoint subspaces by using two pivots per node. It replaces the pivots of a leaf node during insertion operations by using a replacement policy that ensures good partitioning of the metric space. Also, its algorithms for processing similarity queries can efficiently use its partitioning method. The Onion-tree has two versions: (i) the F-Onion-tree, which divides each node of the structure into the same number of subspaces; and (ii) the V-Onion-tree, which applies different numbers of subspaces to the nodes. Here, we are interested in the F-Onion-tree, which according to Carélo et al. [2011], invariably outperformed the V-Onion-tree.

However, the F-Onion-tree only performs element-by-element insertion into its structure. Another important issue is the mass loading technique, called bulk-loading, which builds the index considering all elements of the dataset at once. This technique is useful in the case of reconstructing the index or inserting a large number of elements simultaneously. It is also very useful for the query optimizer of database management systems due to the following factors. As the entire input dataset is already known, it is expected that the bulk-loading generate more compact structures, thus decreasing the memory space required to store the index. It is also expected that the index generated by the bulk-loading provide better performance in the processing of range and k-NN queries, which can be repeated several times after the index is created. Despite the importance of the bulk-loading technique, to the best of our knowledge, there are not in the literature bulk-loading algorithms for the F-Onion-tree.

In this article, we fill this gap. We propose the HeightBL algorithm for bulk-loading F-Onion-trees. The proposed algorithm calculates a priori the estimated height of the index, according to the number of elements to be inserted into the structure and the quantity of subspaces of the F-Onion-tree. It selects, for each node, the pair of elements of the dataset that will generate a structure with approximately the estimated height. Also, to avoid the need to verify each pair of elements, the algorithm chooses samples to be tested. By combining these characteristics, the proposed HeightBL algorithm fulfills the requirements of the bulk-loading technique: compared with the element-by-element insertion, performance tests showed that the HeightBL algorithm produced more compact indices and guaranteed expressive performance gain in range and k-NN query processing.

This article is organized as follows. Section 2 reviews related work, Section 3 details the main characteristics of the F-Onion-tree, Section 4 introduces the proposed bulk-loading algorithm, Section 5 validates the algorithm through performance tests, and Section 6 concludes the article.

2. RELATED WORK

The first dynamic disk-based MAM is the M-tree [Ciaccia et al. 1997]. Its leaf nodes store all the elements of the dataset, while its internal nodes store selected elements called representatives, each having a covering radius. The bulk-loading algorithm for the M-tree described in [Ciaccia and Patella 1998] randomly chooses k elements from the dataset as samples and assigns the remaining elements to the nearest sample, thus producing k groups. The algorithm is recursively applied to each group until the subset is small enough to fit in one node. The bulk-loading algorithm for the M-tree introduced in [Sexton and Swinbank 2004] clusters the input data so that the generated M-tree reflects the performance requirements of the structure. Further, the Slim-tree [Traina-Jr et al. 2002] was the first disk-based MAM explicitly designed to reduce the overlap degree between nodes in a metric tree. Its bulk-loading algorithm [Vespa et al. 2007; 2010] builds the structure in a top-down fashion, based on sampling techniques, and creates balanced trees with little overlap in each node, using the metric domain's distance function and a bound limit to group and determine the number of elements in each partition of the dataset at each step of the algorithm. As the M-tree and the Slim-tree are disk-based MAMs, their bulk-loading algorithms are designed to reduce the overlap between the nodes, which is a problem that is not faced by the main-memory F-Onion-tree.

Bercken and Seeger [2001] introduce two generic bulk-loading algorithms, in the sense that they can be applied to access methods based on trees, including MAMs. The basic idea behind these algorithms is to recursively partition the dataset by using a main-memory index of the same type as the target index to be built. In these algorithms, elements (i.e., samples) of the dataset are inserted into the index maintained in the main memory until the available memory is filled up. Then, a bucket on disk is associated with each leaf node, and the remaining elements of the dataset are inserted into the index guided to the buckets of the corresponding leaf node. When all the elements have been processed, the nodes in the main memory are written to disk. These algorithms can not be applied to bulk-loading F-Onion-trees because they are based on the premise that the amount of data to be inserted in the index does not fit entirely in the main memory and therefore the index should be stored on disk. Also, these algorithms do not explore the characteristics of the Onion-tree, such as its partitioning method.

In addition to the aforementioned related work, it is also important to survey proposals for bulk-loading multidimensional access methods, especially the R-tree. The TGS algorithm [García et al. 1998] partitions the input data into subtrees in a top-down fashion, and at each level of the tree, it rearranges the input data that should be positioned under a node in construction according to a cost function. The OMT algorithm [Lee and Lee 2003] first determines the topology of the resulting R-tree and then groups the input data to create the entries of the root node, aiming to minimize the overlapped area. The remaining nodes are constructed by recursively partitioning each entry to create lower level nodes. The algorithm described in [Arge et al. 1999] is based on a buffering technique that attaches buffers to the nodes of the R-tree. An operation is first guided to the buffers of each node and, when any buffer is full, the operation is performed in the index. Finally, Bercken et al. [1997] describe an algorithm based on the buffering technique to perform the bulk-loading of multidimensional structures, which is based on the use of the split and the merge operations of the index to which the bulk-loading is being applied. Although these related work are used as a basis for the proposal of several bulk-loading techniques, they can not be directly applied to bulk-loading F-Onion-trees as they are based on the characteristics of multidimensional access methods, such as the use of specific partitioning methods, the use of approximations such as the minimum bound rectangle (MBR) and the sort of MBRs to reduce their size and area of overlap.

To generate more compact indices and to provide better query performance than the element-by-element insertion, bulk-loading algorithms should take advantage of the particular characteristics of

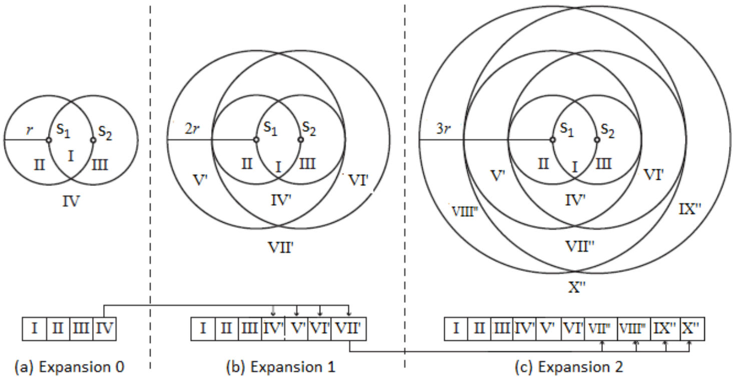

Fig. 1. Generation of a F-Onion-tree's node with 2 expansions, adapted from Carélo et al. [2011]

the index structure. In this article, we use the well-known concept of *sampling* as a basis of our bulk-loading algorithm, but we apply new ideas to it, which respect the characteristics of the F-Onion-tree. First, the proposed HeightBL algorithm chooses samples in a subspace of the metric space by avoiding pair of pivots whose distance is too far or too close. Second, it selects two pivots per node according to their distribution in the metric space, differently from the algorithms described in [Ciaccia and Patella 1998; Vespa et al. 2010], which randomly select only one pivot per node after clustering the remaining elements. Third, the HeightBL algorithm defines a strategy to estimate the ideal height that a F-Onion-tree should have in order to generate a better index structure.

3. THE F-ONION-TREE

The F-Onion-tree [Carélo et al. 2011] is a main-memory MAM that divides the metric space into disjoint subspaces (i.e. regions) by selecting two pivots per node. Its partitioning method is based on the concept of *expansion*, which determines the number of *disjoint regions* that the nodes of a F-Onion-tree should have. In detail, the number of expansions is equal to F, the number R of disjoint regions is determined by $R = F * 3 + 4$, and all nodes of a given F-Onion-tree have the same number of disjoint regions.

Figure 1c depicts a node of a F-Onion-tree ($F = 2$) indexing the set $S = s_1, s_2, ..., s_n$, using s_1, s_2 as pivots. It is composed of ten disjoint regions, which are generated by the partitioning method as detailed as follows: (i) expansion 0 (Figure 1a) represents the initial structure of a node with four regions I, II, III and IV; (ii) expansion 1 (Figure 1b) generates the node with seven regions I, II, III, IV', V', VI' and VII'; and (iii) expansion 2 (Figure 1c) generates the node with ten regions I, II, III, IV', V', VI', VII", VIII", IX" and X". The distance $r = d(s_1, s_2)$ between the pivots defines the initial radius of the ball centered at each pivot. The other radii are determined using the multiplicity of r, i.e. $2r$ for expansion 1 and $3r$ for expansion 2. Each expansion adds three regions to the node, since the previous external region becomes the first region of the expansion.

Let s_i be an element of S to be inserted into the index. The element-by-element insertion performs as follows. During the insertion of s_i, the index is traversed to search for an appropriate node to hold the element. At each node, the insertion calculates two distances: d_1, which is the distance between s_i and the first pivot of the node, and d_2, which is the distance between s_i and the second pivot of the node. The region to hold s_i is determined according to the following rules: (i) if $d_1 < r$ and $d_2 < r$, then s_i is associated with region I; (ii) if $d_1 < r$ and $d_2 > r$, then s_i is associated with region II; (iii) if $d_1 > r$ and $d_2 < r$, then s_i is associated with region III; and (iv) if $d_1 > r$ and $d_2 > r$, then s_i is associated with the external region of the F-Onion-tree, which can be region IV or any other region generated by the expansions. If s_i is associated with an external region of a current expansion

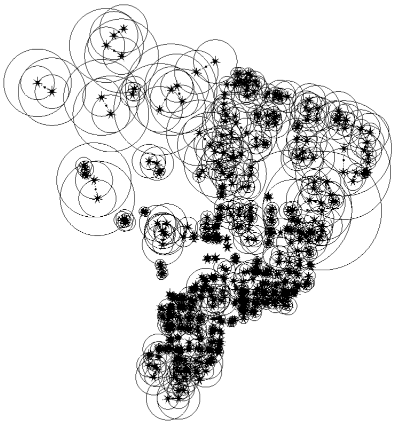

Fig. 2. Visualization of the level 4 of a F-Onion-tree with 1 expansion.

E ($0 \leq E < F$), the previous rules are applied considering the multiplicity of the initial radius r according to the value of the next expansion E + *1*. Also, if s_i should be inserted into a region that already has two pivots, the element-by-element insertion is called recursively to insert this element as a child of the region pivots.

The new element s_i is assigned to a leaf node when an empty leaf node is reached or when a leaf node with only one element is reached. Also, the F-Onion-tree may replace one of the pivots of a full leaf node just before inserting s_i into this node. This is based on a replacement policy that performs a combinatorial analysis between the distances of s_i and the two pivots, and changes any of the pivots with s_i if needed. It also updates the radius of the node with the distance between s_i and the non-chosen pivot. The Onion-tree may use three different replacement policies: (i) the keep-small, which states that the distance between the pivots should be the closest to half of the parent node's radius; (ii) the maximize-expansions, which chooses as pivots the elements that are the closest; and (iii) the minimize-expansions policy, which chooses as pivots the elements that are the most distant. Here, we are interested in the keep-small replacement policy, which according to Carélo et al. [2011], invariably outperformed the other replacement policies.

Figure 2 graphically illustrates a F-Onion-tree for a real-world dataset that contains the geographical coordinates of Brazilian cities (`www.ibge.gov.br`). We used the tool introduced by Traina-Jr et al. [2002] to show a visualization of the level 4 of the structure using the keep-small replacement policy. We applied only one expansion to the F-Onion-tree's nodes. Thus, it is possible to easily see regions I, II, III, IV', V', VI' and VII' of some nodes.

Regarding the range query algorithm, for each pivot of a node, it first analyses if the distance between the query element s_q and the pivot is smaller than the query radius r_q. If this comparison returns true, then the pivot is added to the output set. Next, for each region of the node, the algorithm is called recursively if this region is covered by r_q. On the other hand, the k-NN query algorithm starts calculating the distances between s_q and the pivots of the node. An active radius is maintained with its value equals to the distance of the farthest element of the result set from the moment that the algorithm finds k elements. If the distance between s_q and any of the pivots is smaller than the active radius, the corresponding pivot is added to the result set, keeping it sorted by the distances. Next, the algorithm is called recursively for each region that intersects the active radius. The k-NN query algorithm visits expansions and regions as follows. First, it visits the expansion E in which s_q is assigned. The first region of E to be visited is where s_q lies. The remaining regions are visited according to their proximity to s_q, such that the closest regions to s_q are visited before the farthest regions. Then, the k-NN query algorithm visits expansions E - *1* and E + *1*, and for each expansion,

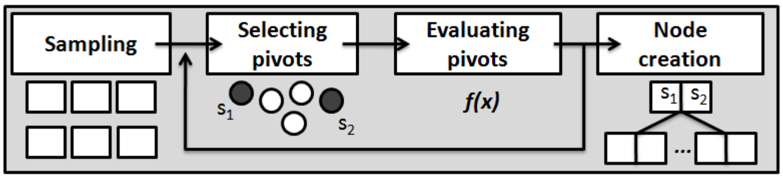

Fig. 3. General view of the HeightBL algorithm.

applies the same visit order for their regions. The algorithm performs recursively for the remaining expansions until k elements be recovered.

4. THE PROPOSED HEIGHTBL ALGORITHM

In this section, we detail the HeightBL (**Height Bulk-Loading**) algorithm, which performs a top-down bulk-loading of F-Onion-trees. It organizes the elements to be inserted into the index in advance to define the best insertion order of these elements. Also, it builds the index structure by using samples to choose the pivots of the nodes. Furthermore, it defines a strategy to estimate the ideal height that a F-Onion-tree should have, so that the final index structure has approximately this estimated height.

In a glance, the HeightBL algorithm works as follows (Figure 3). In the *sampling task*, the algorithm selects elements from the dataset, which are used as samples. Then it starts an iterative process to analyze the samples in order to identify pivots. It generates pairs of samples and evaluates them to investigate which one is the best pair to be chosen, according to the estimated height. The chosen pair of samples is inserted into the index, creating a new node whose pivots are these samples. The tasks of *selecting pivots*, *evaluating pivots* and *node creation* are repeated recursively to obtain subtrees. Section 4.1 details the *sampling task*, and Section 4.2 describes the proposed HeightBL algorithm.

4.1 The Sampling Task

To avoid the need to verify each pair of elements of the dataset to identify the best pair of pivots to create a new node, the *sampling task* chooses elements to be tested, which are used as samples. Therefore, it only investigates these samples to identify the best pair of pivots, pruning a large amount of data to be analyzed and reducing the time spent to build the F-Onion-tree.

The notion used by the *sampling task* to generate samples is described as follows. It discards pairs of elements from the dataset that would generate highly unbalanced structures, i.e. structures that would provide a poor query performance. Consider a pair of elements. There are two situations in which this occurs. The first one occurs when the distance between the elements is too far so that the elements are located in extreme portions of the dataset. If this pair of elements were chosen, then most of the remaining elements would be associated with the region I of the F-Onion-tree. The second situation occurs when the distance between the elements is too close, generating a F-Onion-tree with most of the remaining elements associated with its region IV (or another external region).

Consider the dataset $S = s_1, s_2, ..., s_n$. To choose samples, the *sampling task* performs four sequential subtasks: (1) finding an approximated medoid; (2) calculating the median of the approximated medoid; (3) building a ring of samples; and (4) filtering the samples. They are described as follows.

Subtask 1. Finding an approximated medoid

Subtask 1 is aimed to find an approximated medoid. First, it randomly chooses an element $s_x \in S$ $(1 \leq x \leq n)$. It also chooses two other elements, $s_v \in S$ and $s_w \in S$, such that s_v is the farthest element from s_x and s_w is the farthest element from s_v. Then, it calculates m_v, which is the median of the distances between s_v and each remaining element of S, and m_w, which is the median of the distances between s_w and each remaining element of S.

The generation of the approximated medoid is an iterative process that analyzes the intersection region determined by the balls centered at s_v and s_w, using m_v and m_w as radii, respectively (Figure 4a). Two different situations can occur. In the first one, the intersection region contains elements, which are selected as candidates. In the second situation, there are no elements in the intersection region. In this case, both radii m_v and m_w are incremented by a value determined by Equation 1 to increase the intersection region. These radii may be incremented several times, until an intersection region that contains elements be generated and the candidates be selected. The subtask selects the *approximated medoid* from the candidates by choosing the element whose sum of distances to the remaining elements of S is the smallest.

Equation 1 represents the *distance policy*, which is used by the subtasks of the *sampling task* to select near elements and to increase and decrease values. Intuitively, for datasets with high dimensionality and a large number of elements, there is a high probability to find close elements, even if we consider short distances. Thus, this equation uses the *number of elements* of the dataset and the *dimensionality* of the dataset to select near elements. These elements should be discarded, as we discuss in subtask 4. This equation also determines a value that is used as an increment and a decrement in situations where it is necessary to build a ring of samples, as we discuss in subtask 3. The motivation of Equation 1 is to generate values between 0 and 1 using the characteristics of the dataset, such as data volume and data dimensionality. For non-dimensional datasets, the *dimensionality* can be defined as the value of its intrinsic dimensionality.

$$DistancePolicy = \frac{NumberOfElements}{Dimensionality + NumberOfElements} \tag{1}$$

Subtask 2. Calculating the median of the approximated medoid

To calculate the median of the approximated medoid, subtask 2 determines the median *med* of the distances between the *approximated medoid* and the remaining elements of S.

Subtask 3. Building a ring of samples

Subtask 3 builds a ring of samples as described as follows. First, it builds a ball centered at the *approximated medoid* using *med* as radius (i.e. the ball drawn with dashed line in Figure 4b). Lets consider m_sup and m_inf two other balls that are copies of the first ball, i.e. their radii are initially *med* and their centers are the approximated medoid. Then, the subtask decrements the radius m_inf by the value determined by Equation 1, and increments the radius m_sup by the value determined by Equation 1. The two new balls are those drawn with continuous line in Figure 4b. The *ring* is the area delimited by these balls.

In the following, subtask 3 counts the number of elements l contained in the ring. If $2 \leq l \leq dimensionality\ of\ S$, these elements are selected as *samples*. Otherwise, two different situations can occur. When $l < 2$, m_sup and m_inf are respectively incremented and decremented again by the value determined by Equation 1, until generating a ring that contains an appropriate number of elements. When $l > dimensionality\ of\ S$, the subtask selects as *samples* the l nearest elements to the *approximated medoid* that are inside the ring, such that $l = dimensionality\ of\ S$.

Subtask 4. Filtering the samples

Subtask 4 reduces the number of samples selected in the previous subtask by filtering the samples. To this end, it removes samples that are too close, i.e. the samples whose distance between them is

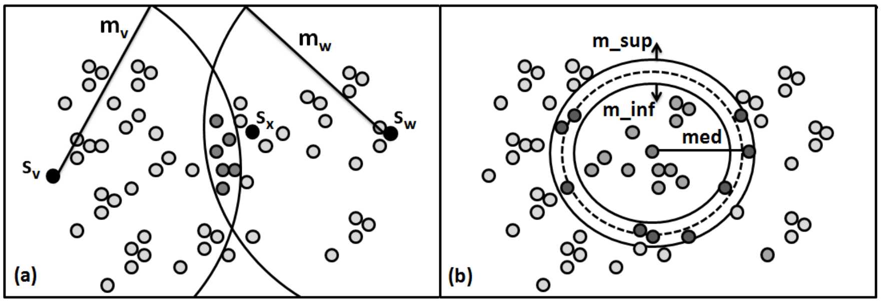

Fig. 4. (a) Example of an intersection region. (b) Example of a ring of samples.

less than the result provided by Equation 1. Otherwise, the number of samples to be tested could be very large.

4.2 Detailing the Algorithm

Before performing the tasks described in Figure 3, the HeightBL algorithm determines the estimated height that a F-Onion-tree should have, i.e. the height that a balanced F-Onion-tree should have. Equation 2 defines this estimated height, using the notion that the total number of elements n that can be stored in a completely full F-Onion-tree is given by the sum of the number of elements that can be stored at each level h of the index, i.e. $n = \sum_{h=0}^{h=H} 2 \times R^h$, where R is the number of regions, and *2* represents that there are two pivots per node. Equation 2 is obtained from this sum, considering that it can be seen as a sum of the $H + 1$ first terms of a geometric progression with ratio R and the first term *2*, and isolating h.

$$EstimatedHeight = \lceil \log_R^{(\frac{n \times (R-1)}{2}+1)} - 1 \rceil \tag{2}$$

Algorithm 1 details the HeightBL algorithm. Its inputs are the elements of the dataset, the estimated height of the F-Onion-tree calculated using Equation 2, the number F of expansions to be applied to the nodes of the index and a reference to the parent node. In the first execution of the algorithm, the reference to the parent node is *null*.

Initially, the algorithm selects the samples to be analyzed as pairs of pivots (line 2). To this end, it performs the four subtasks of the *sampling task* detailed in Section 4.1. Then, the algorithm starts a loop to determine the pair of pivots that should be chosen to create a new node in the index (lines 4 to 15). In detail, in line 6 it selects a pair of pivots from the sample, in line 7 it associates the remaining elements of the dataset with the regions of the node, in line 8 it calculates for each region of the node the height of its subtree, and selects the highest height as the *calculatedHeight*, and in line 9 it updates the value of *calculatedHeight* so that this value also consider the current level of the index. Next, if the calculated height is less or equal than estimated height, then the algorithm chooses the pair of pivots of the current node and ends the loop (lines 11 to 14). The notions of selecting the highest height and increasing it by the current level aim to guarantee that the height of each subtree does not exceed the estimated height of the final F-Onion-tree. For instance, suppose that $EstimatedHeight = 3$. In the first execution of the algorithm, the current level is equal to 0, and the highest height should be at most 3; in the second execution of the algorithm, the current level is equal to 1, and the highest height should be at most 2; and so on.

In the following, the algorithm creates the new node using the selected pivots (line 17). Furthermore, for each region of the created node, it verifies the number of elements to be associated with this region (line 19). If this number is greater than 2, the algorithm is called recursively to that region using as input the elements to be associated with the region, the estimated height calculated before the execution of the algorithm, the number F of expansions and the new node (lines 20 to 22). Otherwise the elements of that region are inserted into a new node that is a child of the current node (line 24).

Algorithm 1: `HeightBL`

```
   Input  : Elements {elements of the dataset}
            EstimatedHeight {height estimated to the F-Onion-tree}
            Expansions {number of expansions of the F-Onion-tree, i.e. F}
            ParentNode {reference to the parent node}
   Output: F-Onion-tree {the final F-Onion-tree}
1         // the sampling task
2  sample ← sampling(Elements, sizeSample);
3  findPivots ← false;
4  while existsPivots(sample) and findPivots = false do
5             // the selecting pivots task
6      pivots ← selectPivots(sample);
7      elementsRegions ← verifyRegions(pivots, Elements - pivots, Expansions);
8      calculatedHeight ← estimatesHeight(elementsRegions, Expansions);
9      calculatedHeight ← calculatedHeight + getCurrentLevel();
10            // the evaluating pivots task
11     if calculatedHeight ≤ EstimatedHeight then
12         findPivots ← true;
13         selectedPivots ← pivots;
14     end
15 end
16        // the node creation task
17 newNode ← insertNode(selectedPivots,ParentNode);
18 for each region in NumberOfRegions(Expansions) do
19     input ← elementsByRegion(elementsRegions,region);
20     if sizeOf(input) > 2 then
21         HeightBL(input, EstimatedHeight, Expansions, newNode);
22     end
23     else
24         insertChildNode(input, newNode, region);
25     end
26 end
```

The complexity of the HeightBL algorithm is determined as follows. Consider n the number of elements of the dataset, and m the number of samples generated from the *sampling task*. The complexity of the *sampling task* is $O(n * log(n))$, since the distances between the elements are sorted to generate the median, and the complexity of the loop between the tasks of *selecting pivots* and *evaluating pivots* is m^2. Also, the complexity of the *node creation* task is $O(1)$. As m is much smaller than n, the complexity of the HeightBL algorithm is $O(n * log(n))$.

5. EXPERIMENTS AND RESULTS

In this section, we detail the experiments carried out to validate the proposed HeightBL algorithm. In the experiments, we used three datasets, with different dimensionalities (i.e. from 32 to 117) and number of elements (i.e. from 2,536 to 102,240). Table I describes each dataset, indicating its name, number of elements, dimensionality and description. Note that these datasets are the most representative ones used in the original article of the Onion-tree [Carélo et al. 2011].

Table I. Characteristics of the datasets

Dataset	#Elements	Dimensionality	Description
Color Histograms	68,025	32	Images histograms from the KDD repository of the University of California at Irvine (`kdd.ics.uci.edu`)
Ozone	2,536	73	Time-series from 1998 to 2004 for ozone level detection (`archive.ics.uci.edu/ml/datasets/Ozone+Level+Detection`)
KDD Cup	102,240	117	Dataset containing cancer images (`www.kddcup2008.com`)

We compared the HeightBL algorithm with the element-by-element insertion of the F-Onion-tree because, to the best of our knowledge, there are no algorithms for bulk-loading F-Onion-trees in the literature. We implemented the proposed algorithm using the C++ language, and used the original implementation in C++ of the insertion-by-insertion algorithm. The source code of the HeightBL algorithm and the Onion-tree can be downloaded from `gbd.dc.ufscar.br/download/HeightBL` and `gbd.dc.ufscar.br/download/Onion-tree`, respectively.

In the tests, we considered the following values of expansions: 7 for the Color Histograms dataset, 7 for the Ozone dataset, and 11 for the KDD Cup dataset. These values guarantee the best index performance for each dataset [Carélo et al. 2011]. We applied the metric L_2 [Wilson and Martinez 1997] to index the Color Histograms and the KDD Cup datasets, and the costly *dynamic time warping* [Berndt and Clifford 1994] to index the Ozone dataset. Also, we applied the keep small technique as the replacement policy (Section 3).

The experiments were performed on a computer with an Intel Core i7 2.67 GHz processor and 12 GB of main memory. We analyzed the cost to build the index and the cost to process similarity queries. We collected the average number of distance calculations and the average elapsed time in seconds, which were recorded building the index 10 times and issuing 500 queries centered at elements randomly chosen from the datasets. We also collected the size of the indices in kilobytes. The range queries were performed varying the radii to recover nearly from 1% to 10% of the elements of each dataset, and the k-NN queries were performed varying the value of k from 2 to 20, encompassing the most common values of k used when performing similarity queries.

5.1 Building the Index

Figure 5 depicts the performance results to build the indices, and also shows the performance differences regarding these results. As expected, the element-by-element insertion outperformed the HeightBL algorithm with regard to the number of distance calculations (Figures 5a to 5c) and the elapsed time (Figures 5d to 5e). The HeightBL algorithm required more distance calculations as it calculates the distance between all elements of a level during the construction of the index. Also, the HeightBL algorithm was slower because it analyzes more elements during the bulk-loading. Considering the number of distance calculations, as the data volume and the data dimensionality increased, the performance losses of the HeightBL algorithm also increased, ranging from 33.33% to 42.86%. As for the elapsed time, the use of a costly metric for the Ozone impaired the proposed algorithm. For this dataset, the HeightBL algorithm was 53.33% slower.

On the other hand, the HeightBL algorithm generated much more compact structures, as depicted in Figure 6. This is due to the fact that the HeightBL algorithm is aimed to find pivots that guarantee a better division of the metric space, while the element-by-element insertion may generate unbalanced structures that require the storage of several pointers to empty regions. According to the performance differences shown in this figure, as the data volume and the data dimensionality increased, the performance gain of the HeightBL algorithm also increased, ranging from 53.42% up to 71.25%.

Not only the impressive reduced size of the indices but also the great improvement in query process-

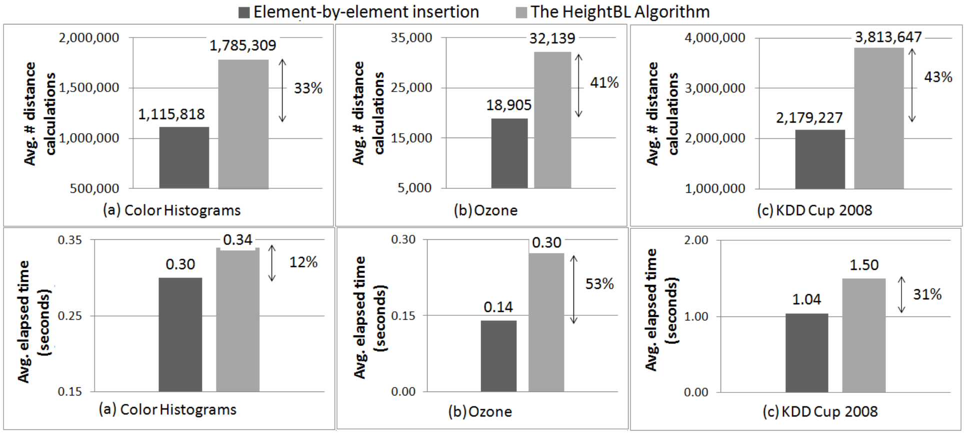

Fig. 5. Number of distance calculations and elapsed time to build the indices.

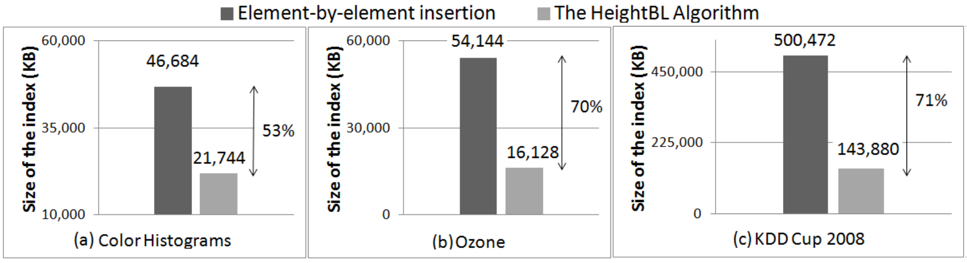

Fig. 6. Size in kilobytes of the indices.

ing (Section 5.2) provided by the HeightBL algorithm over the element-by-element insertion overcome the performance losses of our proposal for building the index. These positive aspects demonstrate the applicability of our algorithm to index real-world data.

5.2 Processing Range and k-NN Queries

The HeightBL algorithm greatly outperformed the element-by-element insertion for all the datasets, with regard to the number of distance calculations and the elapsed time in query processing. This is due to the fact that the proposed algorithm provides a better organization of the elements among the regions of the F-Onion-tree, guaranteeing a better division of the metric space and generating more efficient index structures. In detail, calculating a priori the estimated height of the index and generating a structure with approximately the estimated height provides a better distribution of the elements in the metric space. Thus, range and k-NN queries usually require fewer distance calculations and spend less time to be processed over index structures generated by the HeightBL algorithm, improving query performance. On the other hand, the index structures produced by the element-by-element insertion require that range and k-NN queries perform more recursive calls to reach a leaf node or to find elements that answer these queries, impairing query performance.

Table II shows that the performance gains of the HeightBL algorithm in range query processing varied from 18.75% up to 99.94% in the number of distance calculations and from 10.00% up to 99.04% in the elapsed time. Also, as the data volume and the data dimensionality increased, the performance gain of the HeightBL algorithm also increased. These results are detailed in Figure 7, which depicts the measures average number of distance calculations and average elapsed time for range queries

Table II. The HeightBL algorithm's performance gains (range queries)

Dataset	Distance calculations		Elapsed time	
	minimum	maximum	minimum	maximum
Color Histograms	28.25%	38.73%	10.00%	20.00%
Ozone	27.75%	96.43%	11.43%	76.92%
KDD Cup	18.75%	99.94%	16.43%	99.04%

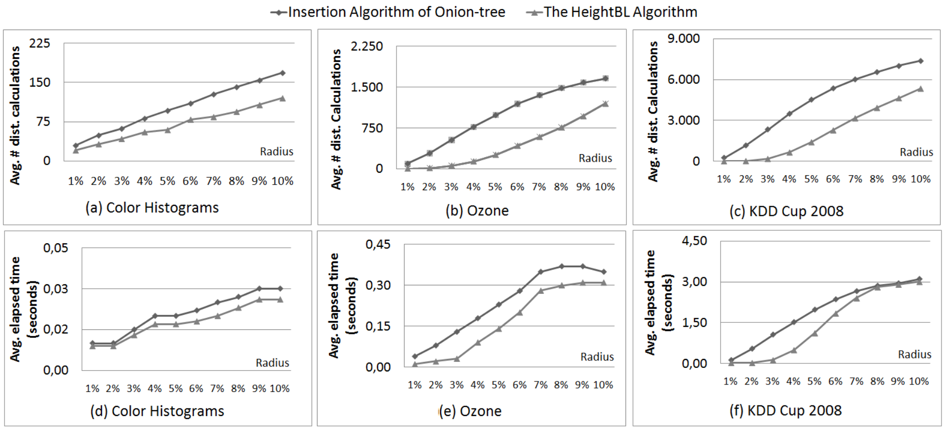

Fig. 7. Range queries results.

Table III. The HeightBL algorithm's performance gains (k-NN queries)

Dataset	Distance calculations		Elapsed time	
	minimum	maximum	minimum	maximum
Color Histograms	13.38%	30.38%	8.57%	16.67%
Ozone	16.88%	51.66%	13.33%	30.00%
KDD Cup	37.45%	39.53%	11.67%	15.44%

recovering from 1% to 10% of the elements of the datasets.

Table III shows that the performance gains of the HeightBL algorithm in k-NN query processing ranged from 13.38% to 51.66% in the number of distance calculations and from 8.57% to 30% in the elapsed time. The best case referred to the Ozone dataset, which was indexed using the costly *dynamic time warping*. These results are detailed in Figure 8, which depicts the measures average number of distance calculations and average elapsed time for the value of k ranging from 2 to 20.

6. CONCLUSIONS AND FUTURE WORK

In this article, we proposed the HeightBL, an algorithm for bulk-loading F-Onion-trees. The proposed HeightBL algorithm introduces the following distinctive properties. It is top-down, and organizes the elements to be inserted into the index in advance to define the best insertion order of these elements. It also builds the structure by using samples to choose the pivots of the nodes. Furthermore, it defines a strategy to estimate the ideal height that a F-Onion-tree should have, so that the final structure has approximately this estimated height.

The HeightBL algorithm was validated through performance testes using real-world data with different volumes and dimensionalities. The results showed that the HeightBL algorithm generated very compact indices. Compared with the element-by-element insertion of the F-Onion-tree, the size of the

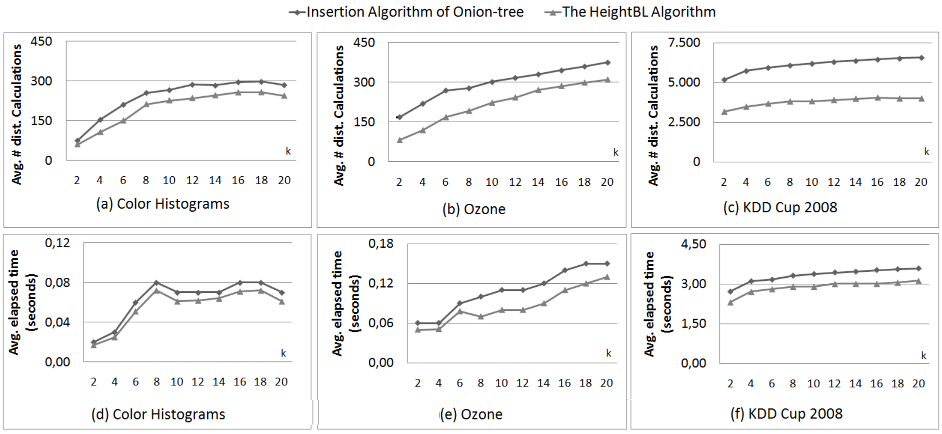

Fig. 8. k-NN queries results.

index reduced from 53.42% to 71.25%. The results also demonstrated that the HeightBL algorithm greatly improved the similarity query processing in comparison to the element-by-element insertion. It was from 10.00% to 99.04% faster to process range queries and was from to 8.57% to 30.00% faster to process k-NN queries. It also reduced the number of distance calculations from 18.75% to 99.94% to process range queries and from 13.38% to 51.66% to process k-NN queries. We can conclude that the HeightBL algorithm fulfills the requirements of the bulk-loading technique: it produces more compact indices and guarantees expressive performance gain in range and k-NN query processing.

We are currently developing a bottom-up algorithm for bulk-loading F-Onion-trees. We also plan to run experiments using new real-world datasets with different metrics and characteristics, as well as to investigate experiments with synthetic data. Furthermore, we plan to propose an algorithm that performs deletions of elements from the F-Onion-tree. Another future work is to apply the F-Onion-tree and its HeightBL algorithm to execute condition-extended similarity queries over complex data, such as those queries described by Soares and Kaster [2013].

REFERENCES

Arge, L., Hinrichs, K., Vahrenhold, J., and Vitter, J. S. Efficient Bulk Operations on Dynamic R-trees. In *Proceedings of the International Workshop on Algorithm Engineering and Experimentation*. London, UK, pp. 328–348, 1999.

Bercken, J. V. d. and Seeger, B. An Evaluation of Generic Bulk Loading Techniques. In *Proceedings of the International Conference on Very Large Data Bases*. Roma, Italy, pp. 461–470, 2001.

Bercken, J. V. d., Seeger, B., and Widmayer, P. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *Proceedings of the International Conference on Very Large Data Bases*. Athens, Greece, pp. 406–415, 1997.

Berndt, D. J. and Clifford, J. Using Dynamic Time Warping to Find Patterns in Time Series. In *Knowledge Discovery in Databases Workshop*. Seattle, USA, pp. 359–370, 1994.

Bozkaya, T. and Ozsoyoglu, M. Distance-Based Indexing for High-Dimensional Metric Spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Tucson, USA, pp. 357–368, 1997.

Bozkaya, T. and Ozsoyoglu, M. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Transactions on Database Systems* 24 (3): 361–404, 1999.

Brin, S. Near Neighbor Search in Large Metric Spaces. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, USA, pp. 574–584, 1995.

Carélo, C. C. M., Pola, I. R. V., Ciferri, R. R., Traina, A. J. M., Traina-Jr, C., and Ciferri, C. D. A. Slicing the Metric Space to Provide Quick Indexing of Complex Data in the Main Memory. *Information Systems* 36 (1): 79–98, 2011.

Chávez, E., Navarro, G., Baeza-Yates, R. A., and Marroquín, J. L. Searching in Metric Spaces. *ACM Computing Surveys* 33 (3): 273–321, 2001.

Ciaccia, P. and Patella, M. Bulk Loading the M-tree. In *Proceedings of the Australasian Database Conference*. Perth, Australia, pp. 15–26, 1998.

Ciaccia, P., Patella, M., and Zezula, P. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, USA, pp. 426–435, 1997.

Fu, A. W.-C., Chan, P. M.-S., Cheung, Y.-L., and Moon, S. Y. Dynamic VP-tree Indexing for n-Nearest Neighbor Search Given Pair-Wise Distances. *VLDB Journal* 9 (2): 154–173, 2000.

García, R., Yván, J., López, M. A., and Leutenegger, S. T. A Greedy Algorithm for Bulk Loading R-trees. In *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*. New York, USA, pp. 163–164, 1998.

Hjaltason, G. R. and Samet, H. Index-Driven Similarity Search in Metric Spaces. *ACM Transactions on Database Systems* 28 (4): 517–580, 2003.

Korn, F., Pagel, B.-U., and Faloutsos, C. On the "Dimensionality Curse" and the "Self-Similarity Blessing". *IEEE Transactions on Knowledge and Data Engineering* 13 (1): 96–111, 2001.

Lee, T. and Lee, S. OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree. In *CAiSE Short Paper Proceedings*. Klagenfurt, Austria, pp. 69–72, 2003.

Pola, I. R. V., Traina, A. J. M., and Traina-Jr., C. Easing the Dimensionality Curse by Stretching Metric Spaces. In *Proceedings of the International Conference on Scientific and Statistical Databases Management*. New Orleans, USA, pp. 417–434, 2009.

Pola, I. R. V., Traina-Jr., C., and Traina, A. J. M. The MM-Tree: A Memory-Based Metric Tree Without Overlap Between Nodes. In *Proceedings of the East European Conference on Advances in Databases and Information Systems*. Varna, Bulgaria, pp. 157–171, 2007.

Sexton, A. P. and Swinbank, R. Bulk Loading the M-Tree to Enhance Query Performance. In *Proceedings of the British National Conference on Databases*. Edinburgh, UK, pp. 190–202, 2004.

Soares, L. C. and Kaster, D. S. cx-Sim: A Metric Access Method for Similarity Queries with Additional Conditions. *Journal of Information and Data Management* 4 (3): 437–452, 2013.

Traina-Jr., C., Traina, A. J. M., and Faloutsos, C. Fast Feature Selection Using Fractal Dimension - Ten Years Later. *Journal of Information and Data Management* 1 (1): 17–20, 2010.

Traina-Jr, C., Traina, A. J. M., Faloutsos, C., and Seeger, B. Fast Indexing and Visualization of Metric Data Sets Using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering* 14 (2): 244–260, 2002.

Uhlmann, J. K. Satisfying General Proximity/Similarity Queries with Metric Trees. *Information Processing Letters* 40 (4): 175–179, 1991.

Vespa, T. G., Traina-Jr., C., and Traina, A. J. M. Bulk-Loading Dynamic Metric Access Methods. In *Proceedings of the Brazilian Symposium on Databases*. João Pessoa, Brazil, pp. 160–174, 2007.

Vespa, T. G., Traina-Jr., C., and Traina, A. J. M. Efficient Bulk-Loading on Dynamic Metric Access Methods. *Information Systems* 35 (5): 557–569, 2010.

Wilson, D. R. and Martinez, T. R. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research* 6 (1): 1–34, 1997.

Yianilos, P. N. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, USA, pp. 311–321, 1993.