M-Chord: A Scalable Distributed Similarity Search Structure

David Novak and Pavel Zezula
Masaryk University, Brno, Czech Republic
{xnovak8,zezula}@fi.muni.cz

Abstract

The need for a retrieval based not on the attribute values but on the very data content has recently led to rise of the metric-based similarity search. The computational complexity of such a retrieval and large volumes of processed data call for distributed processing which allows to achieve scalability. In this paper, we propose M-Chord, a distributed data structure for metric-based similarity search. The structure takes advantage of the idea of a vector index method iDistance in order to transform the issue of similarity searching into the problem of interval search in one dimension. The proposed peer-to-peer organization, based on the Chord protocol, distributes the storage space and parallelizes the execution of similarity queries. Promising features of the structure are validated by experiments on the prototype implementation and two real-life datasets.

1. Introduction

The field of similarity data retrieval has recently made a rapid progress. Generally, the objective of this search mechanism is to retrieve all indexed data that are similar with a given query object a digital image, a text document, etc.

One way towards the similarity search is adapting the traditional attribute-based retrieval which usually leads to complex queries in multi-dimensional vector spaces. The standard index structures that are able to process such queries, e.g. kd-tree or quadtree, seem to become inefficient for high number of dimensions that are common for current data. Furthermore, data types that do not use the Euclidian distance to measure similarity (e.g., digital images compared by the Earth Moover's Distance) and some special data types (e.g., texts or DNA sequences) cannot be indexed efficiently by vector-based data structures at all.

These research challenges have led to the development of the area of *metric*-based similarity search. This approach considers the data space as a *metric space* dataset together with a *distance function* applicable to every pair of objects. The queries in this model are defined by a *sample query object* and a constraint on the required proximity to the query object. Many principles and index structures have been proposed in this field, summarized in several comprehensive surveys [14, 21].

In real-life applications, the distance function is typically expensive to compute. Unfortunately, even with sophisticated index structures [9, 10] the similarity search is expensive and the increase of the costs is linear with respect to the size of the dataset indexed. Since the volumes of managed data become still larger, there is an evident need for distributed processing that brings two benefits distribution of the storage and parallelization of the time-consuming query execution. Most of the recent effort in the field of distributed data structures has focused on the vector-based approach [20, 8, 12, 2, 6, 1]. As far as we know, the only metric-based distributed structure published are the *GHT** index [4, 3] and, very recently, *MCAN* [11].

In this paper, we introduce a new distributed data structure called *M-Chord*. It maps the metric space into one-dimensional domain using a generalized and adapted variant of *iDistance* [15] a vector index method for nearest-neighbors search. In this way, *M-Chord* reduces the issue of similarity search to the interval search problem. The data is divided among the nodes of the structure and the Peer-to-Peer protocol *Chord* [17] is utilized for intra-system navigation. Finally, the similarity search algorithms are designed for the proposed architecture.

The paper is organized as follows. Section 2 reminds the theoretical background of the metric-based searching and describes the *iDistance* and *Chord* techniques. In Section 3, we provide both the ideas behind and the architecture of the proposed structure. Section 4 presents results of the performance trials and the paper concludes with related work and a future work outline in Sections 5 and 6.

2. Preliminaries

In this section, we shortly remind basic concepts of the similarity searching and indexing in metric spaces and then mention two techniques important for this work the *iDistance* and the *Chord*.

INFOSCALE '06. Proceedings of the First International Conference on Scalable Information Systems, May 29-June 1 2006, Hong Kong
© 2006 ACM 1-59593-428-6/06/05...$5.00

2.1. Metric-based searching

Mathematically, metric space $\mathcal{M}$ is a pair $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is the *domain* of objects and d is the total *distance function* $d : \mathcal{D} \times \mathcal{D} \longrightarrow \mathbb{R}$ satisfying the following conditions for all objects $x, y, z \in \mathcal{D}$:

$$\begin{array}{ll} d(x,y) \geq 0 & (\textit{non-negativity}), \\ d(x,y) = 0 \text{ iff } x = y & (\textit{identity}), \\ d(x,y) = d(y,x) & (\textit{symmetry}), \\ d(x,z) \leq d(x,y) + d(y,z) & (\textit{triangle inequality}). \end{array}$$

Let us define two types of similarity queries [21] we that focus on. Let $I \subseteq \mathcal{D}$ be a finite set of indexed objects.

Definition 1: Given an object $q \in \mathcal{D}$ and a *maximal search radius* r, *range query* $\mathbf{Range}(q, r)$ selects a set S_A of indexed objects: $S_A = \{x \in I | d(q, x) \leq r\}$.

Definition 2: Given an object $q \in \mathcal{D}$ and an integer $k \geq 1$, *k-nearest neighbors query* $\mathbf{kNN}(q, k)$ retrieves a set $S_A \subseteq I : |S_A| = k, \forall x \in S_A, \forall y \in I \setminus S_A : d(q, x) \leq d(q, y)$.

2.2. Indexing the distance

The *iDistance* [15] is an index method for similarity search in vector spaces. It partitions the data space into n clusters and selects a reference point p_i for each cluster C_i, $0 \leq i < n$. Every data object is assigned a one-dimensional *iDistance* key according to the distance to its cluster's reference object. Having a constant c to separate individual clusters, the *iDistance* key for an object $x \in C_i$ is

$$iDist(x) = d(p_i, x) + i \cdot c. \qquad (1)$$

Expecting that c is large enough, all objects in cluster C_i are mapped to the interval $[i \cdot c, (i + 1) \cdot c)$ see Figure 1(a) for the mapping visualization. The data is then stored in a B$^+$-tree according to the *iDistance* keys.

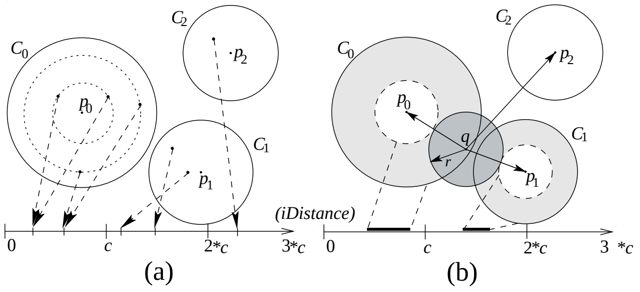

Figure 1. The principles of iDistance

Although the *iDistance* is primarily proposed as a $\mathbf{kNN}$ search method, the algorithm for $\mathbf{Range}(q, r)$ queries is quite straightforward. It searches separately those clusters that possibly contain objects from the query range these are all clusters C_i that satisfy

$$d(q, p_i) - r \leq \text{max-dist}_i$$

where max-dist$_i$ is the maximum distance between p_i and objects in cluster C_i. Figure 1(b) shows the clusters influenced by the query (C_0, C_1) and specifies more precisely the space areas to be searched. Such an area within cluster C_i corresponds to the *iDistance* interval

$$[d(p_i, q) + i \cdot c - r, d(p_i, q) + i \cdot c + r]. \qquad (2)$$

So, several *iDistance* intervals are determined, distance $d(q, x)$ is evaluated for all objects x from these intervals and the query answer set $S_A \subseteq I$ is created as $S_A = \{x | d(q, x) \leq r\}$.

The *iDistance* $\mathbf{kNN}(q, k)$ algorithm is based on repetitive range queries with growing radius. Such a policy is not very convenient for distributed environment and we propose a different one (see below).

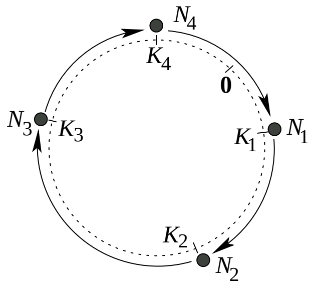

Figure 2. The Chord structure

2.3. Chord

The *Chord* [17] is a P2P protocol providing the functionality of a *Distributed Hash Table* an efficient localization of the node that stores the data item corresponding to a given search key. It is a message driven dynamic structure that is able to adapt as nodes (cooperating computers) join or leave the system.

Using *consistent hashing*, the protocol uniformly maps the domain of search keys into the *Chord* domain of keys $[0, 2^m)$. Every *Chord* node N_i is assigned a key K_i from the same domain $K_i \in [0, 2^m)$. The identifiers are ordered in an *identifier circle* modulo 2^m, $K_i < K_j \Leftrightarrow i < j$. Node N_i is responsible for all keys from interval $(K_{i-1}, K_i]$ (mod 2^m) see Figure 2 for visualization.

Notation: For every key $k \in [0, 2^m)$, let us denote $N_{\langle k \rangle}$ the node responsible for key k.

Every node stores addresses of its predecessor and successor on the identifier circle and, furthermore, it maintains a routing table called the *finger table* with addresses of up to m other nodes [17]. Due to the uniformity of the *Chord* domain distribution, the protocol preserves, with high probability, that:

- in an n-node system, the node responsible for a given key is located via $\mathcal{O}(\log n)$ number of messages to other nodes (number of hops);
- the storage load of the nodes is balanced.

3. M-Chord

In this section, we describe basic ideas and architecture of *M-Chord* the proposed distributed data structure for similarity searching in general metric spaces. The *M-Chord* approach can be summarized as follows:

- generalize the idea of *iDistance* to metric spaces and map the dataset to a $[0, 2^m)$ domain of keys;
- divide the $[0, 2^m)$ domain into intervals and let every node store data with keys from one interval;
- use the *Chord* routing mechanism for navigation;
- design the $\mathbf{Range}$ and $\mathbf{kNN}$ search algorithms;
- provide an additional filtering mechanism to reduce the computational cost of the query processing.

The rest of this section analyzes this approach in detail.

3.1. Basic principles

The *iDistance* algorithm partitions the data space and selects a reference point in every partition. This approach is applicable in vector spaces where the coordinate system can be used for partitioning and the reference points may be selected from the whole domain. In a general metric space (without any knowledge about the specific dataset), the only way to specify reference objects is by choosing them from a given set of objects and only then the space $\mathcal{D}$ can be partitioned with respect to these objects.

In order to generalize *iDistance* to metric spaces, we first choose a set of n pivots $p_0, p_1, \ldots, p_{n-1}$ from an a priori given sample dataset $S \subseteq \mathcal{D}$. Then, the Voronoi-like partitioning [21] is used to divide the set of indexed objects I into clusters $C_0, C_1, \ldots, C_{n-1}$:

$$C_i = \{x \in I | d(p_i, x) \leq d(p_j, x), 0 \leq j < n\}.$$

This modification has no impact on *iDistance* functionality.

While partitioning the data space, the distances $d(p_0, x), \ldots, d(p_{n-1}, x)$ are computed for every object $x \in I$. These values can be stored together with the object x and can be used at query time for further filtering. The triangle inequality property of function d implies the standard *filtering criterion* for $\mathbf{Range}(q, r)$ query [21]: Every object $x \in I$ may be excluded without evaluating $d(q, x)$ if

$$\exists i (0 \leq i < n) : |d(p_i, x) - d(p_i, q)| > r. \qquad (3)$$

This criterion is applied to reduce number of distance computations at query-time.

Pivot selection Let us discuss the way in which the pivots are selected. The proposed selection algorithm is general and applicable to any metric dataset. Exploiting some additional knowledge about the particular dataset, various data-tailored methods can be developed.

Because the pivots are used for filtering, their selection influences the performance of the search algorithm. So, the main objective of the pivoting technique is to increase the efficiency of the filtering criterion (3). We follow the approach described by Chavez et al. [7] which corresponds with our requirements.

First, let us deduce a formula for comparing the quality of two sets of pivots. For $\mathbf{Range}(q, r)$ query and a set of pivots $\{p_0, \ldots, p_{n-1}\}$, let us denote

$$D_{\{p_0,\ldots,p_{n-1}\}}(q, x) = \max_{0 \leq i < n} |d(p_i, x) - d(p_i, q)|.$$

The filtering condition (3) may be reformulated as

$$D_{\{p_0,\ldots,p_{n-1}\}}(q, x) > r \qquad (4)$$

and the efficiency of the filtering grows with the growing probability of (4). One way to increase this probability is to find a set of pivots that maximizes the mean of distribution of $D(x, y)$, $x, y \in \mathcal{D}$. Let us denote the mean value μ_D. So, we say that $\{p_0, \ldots, p_{n-1}\}$ is a better set of pivots than $\{p'_0, \ldots, p'_{n-1}\}$ when:

$$\mu_{D_{\{p_0,\ldots,p_{n-1}\}}} > \mu_{D_{\{p'_0,\ldots,p'_{n-1}\}}}. \qquad (5)$$

The value of μ_D can be estimated on a sample set $S \subseteq \mathcal{D}$. Having the comparison criterion (5), several actual techniques for pivot selection can be defined. The proposed system adopts the *incremental selection technique* [7].

If the dataset to be indexed $I \subseteq \mathcal{D}$ is known a priori, the pivot selection can be tailored directly for it. Otherwise, we try to maximize the filtering efficiency on the whole domain $\mathcal{D}$. This is possible when designing a specific real-life application with a specific domain $\mathcal{D}$. If the distribution of the sample S strongly differed from the distribution of I, the pivot filtering efficiency would be damaged.

M-Chord domain Having the pivots selected and the data space partitioned, we can use *iDistance* to map the dataset into a one-dimensional domain and join this domain with the *Chord* protocol. Since the *Chord* presumes a key space of size 2^m, the range of the *iDistance* domain is normalized by an order-preserving function h to a $[0, 2^m)$ interval. The *iDistance* formula (1) for an object $x \in C_i$, $0 \leq i < n$ becomes an *M-Chord* key-assignment formula:

$$mchord(x) = h(d(p_i, x) + i \cdot c). \qquad (6)$$

The transformation h has another important purpose. In our approach, every node takes over responsibility for a subinterval of the *M-Chord* domain. Such a partitioning should follow the domain distribution to preserve balanced load of the nodes. Further, the *Chord* protocol guarantees its efficiency while nodes are distributed *uniformly* on the domain circle. Therefore, the ideal transformation h would map the original $iDist$ domain on the $[0, 2^m)$ interval *uniformly*.

Generally, the task is to find an order-preserving uniform transformation $h : [0, A] \longrightarrow [0, B]$ (knowing the distribution of $[0, A]$ on a set S). This is a very well studied topic and can be solved, for instance, by a piecewise-linear transformation [13]. This method fixes the h-values in several selected points and the overall transformation is the linear interpolation of these values.

The described construction works flawlessly only when distribution of the set S copies distribution of the indexed data I. It is impossible to reach this criterion fully in a real-life application. But note that the imperfection of the *M-Chord* domain uniformity can only slightly degrade the distribution of the keys assigned to nodes and, thus, the routing efficiency (hop count). The storage load balance of the nodes is addressed in other way (see Section 3.2) and is not influenced by the domain distribution.

3.2. Data structure

The *M-Chord* system constitutes a logical overlay over any network of directly addressable nodes. The structure consists of autonomous nodes that store data, can insert objects into the system, and retrieve them by executing similarity queries. The nodes communicate via messages.

The logical topology of the network corresponds to the structure of *Chord*. As explained above, the data objects are assigned keys from the *M-Chord* domain according to (6). The nodes N_i are assigned keys K_i from the same domain in order to divide the data among the nodes. So, every node N_i contains:

- *Chord* routing information key K_i, links to predecessor and successor nodes and the finger table;
- B$^+$-tree storage for the $(K_{i-1}, K_i]$ (mod 2^m) interval.

Initialization phase The initialization proceeds on the first node started and the determined settings are then used in all nodes of the system. The initialization algorithm has the following parameters:

- sample set $S \subseteq \mathcal{D}$;
- number of pivots n.

First, the pivots $p_0, p_1, \ldots, p_{n-1}$ are selected from S using the algorithm described in Section 3.1. Then, the *iDistance* formula (1) is applied to determine the distribution of $iDist$ function on S and this information is used to find the transformation h. Both, the pivots and the function h, are necessary for evaluation of the $mchord$ key-assignment function (6).

When other than first node joins the system, it receives this configuration from an already running node we presume that the joining node learns about an existing node through some external mechanism.

Nodes activation The first node of the system is automatically assigned key $2^m - 1$ and covers the whole *M-Chord* domain $[0, 2^m)$. Other nodes are not assigned *M-Chord* keys on the startup they become *non-active* nodes at first. There are several ways to maintain the pool of non-active nodes, e.g., separated distributed peer-to-peer layer, central (replicated) register, broadcast messages in LAN, etc. The choice of a suitable solution is left to a specific implementation environment.

If there is an available non-active node N_{new} then any active node can invoke a *split* request. Generally, every node may define its own criteria forcing split these could be reaching a storage capacity limit, heavy computational load (either because of frequent *M-Chord* claims, demands of other processes, or weak technical parameters), etc. The splitting of node N_i, responsible for interval $(K_{i-1}, K_i]$, proceeds as follows:

- determine a key K_{new}: $K_{i-1} < K_{\text{new}} < K_i$ and assign it to a non-active node N_{new};
- move data from the $(K_{i-1}, K_{\text{new}}]$ interval to N_{new} and follow the standard *Chord* join mechanism for N_{new}.

Note that, due to the separation constant c in the $mchord(x)$ formula (6), values $h(i \cdot c)$, $0 \leq i \leq n$ form the boundaries of particular clusters within the $[0, 2^m)$ domain. If the interval $(K_{i-1}, K_i]$ covers more than one cluster (z clusters, $z > 1$) then K_{new} is selected as a cluster boundary so that interval $(K_{i-1}, K_{\text{new}}]$ covers $\lfloor \frac{z}{2} \rfloor$ clusters. If interval $(K_{i-1}, K_i]$ covers only one cluster (or its part) then K_{new} is set in order to split the storage of N_i equally.

Insert Any node N_{ins} can initiate insert operation for an object $x \in \mathcal{D}$. First, node N_{ins} applies Formula 6 to calculate $mchord(x)$ key. Values $d(p_0, x), \ldots, d(p_{n-1}, x)$ are obtained as a by-product and are carried along with x from now on.

If N_{ins} is non-active then the request is forwarded to the node known from the startup. If N_{ins} is active then the *Chord* protocol is followed to forward the request to $N_{\langle mchord(x) \rangle}$ (node responsible for $mchord(x)$). This node stores x into the B$^+$-tree storage according to the $mchord(x)$ key. See an example in Figure 3(a).

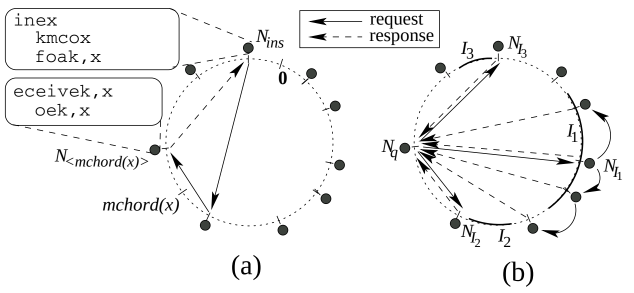

Figure 3. The insert (a) and range search (b)

3.3. Range search

The $\mathbf{Range}(q, r)$ query algorithm is usually the base algorithm for more sophisticated similarity queries. The *M-Chord* structure has been designed in such a way that the **Range** algorithm may follow the *iDistance* pruning idea. The node that initiates the query, N_q, executes the RANGESEARCH(q, r) algorithm with the following schema:

- for each cluster C_i, $0 \leq i < n$ determine interval I_i of keys to be scanned:

  $$I_i = [h(d(p_i, q) + i \cdot c - r), h(d(p_i, q) + i \cdot c + r)];$$

- $\forall i : 0 \leq i < n$ send an INTERVALSEARCH(I_i, q, r) request to node N_{I_i} responsible for the midpoint of interval I_i: $N_{I_i} = N_{\langle h(d(p_i,q)+i \cdot c) \rangle}$;
- wait for all responses and create the final answer set.

The INTERVALSEARCH(I_i, q, r) algorithm, executed on the N_{I_i} nodes, $0 \leq i < n$, has the following schema:

- if N_{I_i} is not responsible for the whole interval I_i, forward the INTERVALSEARCH(I_i, q, r) request to the predecessor and/or successor;
- examine the locally stored objects and create the local answer set $S_A = \{x | mchord(x) \in I_i, d(q, x) \leq r\}$; use the precomputed values $d(p_i, x)$ and the filtering formula (3) while evaluating $d(q, x) \leq r$;
- return S_A to node N_q and eventually notify N_q to wait for responses from the predecessor and/or successor nodes.

Figure 3(b) shows an example of a **Range** query flow through the system. Both the algorithm description and the picture simplify the query forwarding to nodes N_{I_i} the *Chord* protocol must be employed to reach these nodes. So, n queries are spread simultaneously from node N_q following the *Chord* protocol. According to this protocol, a lot of these messages travel partially along the same path. In order to decrease flooding of the network, the requests are sent as one message along the common path parts.

As described in Section 2.2, the *iDistance* search algorithm may exclude cluster C_i from the search if $d(q, p_i) - r >$ max-dist$_i$, where max-dist$_i$ is the radius of C_i. This is not possible in the distributed algorithm because values max-dist$_i$ are not known to all nodes. Thus, some (most of them) interval search requests return immediately after reaching N_{I_i} node because there is nothing to be searched within the I_i interval. This issue and its possible solutions are candidates for future studies.

3.4. Nearest-neighbors search

The *iDistance* approach to **kNN** query processing a sequence of $\mathbf{Range}(q, r)$ queries with growing radius r does not seem to be suitable for distributed environment because multiple **Range** iterations would result in an unpleasant number of successive message transmissions increasing the overall response time. Our approach, similar to the approach adopted, e.g., in *GHT** [4], has two phases:

1. Employ a low-cost heuristic to find k objects that are near q. Measure the distance ϱ_k to the k^{th} nearest object found. Value ϱ_k is an upper bound of the distance to the actual k^{th} nearest neighbor of q.
2. Run the $\mathbf{Range}(q, \varrho_k)$ query and return the k nearest objects from the query result (skip the space searched during the first phase).

In the first phase, node $N_{\langle mchord(q) \rangle}$ searches the cluster C_i into which q belongs:

- localize the B$^+$-tree leaf covering key $mchord(q)$;
- walk through the B$^+$-tree leaves alternately left and right, add the first k objects to the answer set S_A, and initialize the ϱ_k value;
- continue walking left and right examining objects x while keys $mchord(x) \in I_i$

  $$I_i = [h(d(p_i, q) + i \cdot c - \varrho_k), h(d(p_i, q) + i \cdot c + \varrho_k)];$$

- if $d(q, x) < \varrho_k$ then remove the k^{th} object from S_A, add x into S_A, and update ϱ_k interval I_i shrinks;
- finish searching when either the whole interval I_i or the cluster C_i in $N_{\langle mchord(q) \rangle}$ has been searched.

Note that the described algorithm presumes that at least k objects are stored in cluster C_i on node $N_{\langle mchord(q)\rangle}$. If so, the second phase is applied otherwise we adopt the *optimistic strategy* by iterating over the following steps:

- run a range query with radius ϱ distance to the farthest object found so far;
- if $k' < k$ objects are found, increment ϱ by $\varrho\frac{k-k'}{k}$ and search again skipping the space already searched.

Finish searching when k objects are found.

4. Performance evaluation

In this section, we present and analyze results of experiments conducted on the prototype implementation of *M-Chord*. The experiments focus mainly on the system performance for $\mathbf{Range}$ and $\mathbf{kNN}$ queries processing and on various aspects of scalability of the system.

4.1. Experiments settings

The experiments have been performed on up to 300 workstation nodes connected by a high-speed local-area network. The nodes communicate via TCP protocol. The following real-life datasets have been selected to conduct the experiments on:

VEC 45-dimensional vectors of color image features compared by a *quadratic-form* distance function [21]. Distribution of the distances between pairs of objects is practically normal and such a high-dimensional space is extremely sparse.

TTL titles and subtitles of books and periodicals from several academic libraries. The *edit distance* function [16] on the level of individual characters is used to compare these *strings* of lengths from 3 to 200 characters. The distance distribution of this dataset is skewed.

Observe that neither of these datasets can be efficiently indexed and searched by a standard vector data structures **VEC** uses a specific distance function and **TTL** does not form a vector space at all. Both distance functions are highly computationally intensive.

When testing the system scalability with respect to dataset size we use the whole sets (1,000,000 objects for **VEC** and 800,000 objects for **TTL**) otherwise the size of the stored data is fixed to 200,000. The sample set S used in the initialization phase (Section 3.2) consists of 5,000 objects randomly chosen from the dataset. The number of pivots n is discussed in Section 4.3. The separation constant c, used in Equation 6, is determined by a simple heuristic the double of the maximal distance between an objects in S and its *farthest* pivot p_i. Because the *M-Chord* domain is normalized and redistributed by the h function, the value of the constant c has no serious impact. Size of the *M-Chord* domain is 2^{20}. For simplicity and transparency, we define only one criterion forcing nodes' splits reaching storage capacity limit of 5,000 objects.

4.2. What to measure – how to measure

Since the technical resources used for testing were not dedicated, our experimental environment was very fluctuating the computational load of the workstations caused by other running processes and load of the network was unpredictable and difficult to measure. Therefore, we do not present the exact response times values or duration of computations. Generally, the query response times on our implementation were below one second for smaller radii and about two seconds for bigger radii regardless of the dataset size.

The CPU costs are measured as the number of evaluations of the distance function d. This is a standard way for evaluation of metric-based structures other operatio ns (and usually I/O cost as well) are practically negligible compared to the distance evaluation demands. We measure the *total cost* on all participating nodes and the *parallel cost* maximal number of distance computations performed in a sequential manner. The total costs correspond to costs on a centralized version of the structure.

The communication costs are measured by the *total number of messages* (both requests and responses) and by the maximal number of messages sent in a serial way the *maximal hop count*. We also present the number of nodes influenced by the query processing. All presented measurement results are taken as an average over queries on fifty query objects randomly selected from the dataset.

4.3. Number of pivots

In the first set of experiments, we measured the influence of the number of pivots (clusters) n on the performance of the search algorithms. For all tested numbers of pivots (1 70) and for 200,000 stored objects, there were about sixty active nodes in the system so the average storage load ratio was approx. 66%.

Figure 4 shows the total number of distance computations for $\mathbf{Range}(q, r)$ queries with respect to the number of pivots n. Generally, the filtering formula (3) reduces the number of distance computations less effectively for **TTL** because two strings of similar length often have similar *edit distance* from any pivot.

Figure 5 shows the parallel number of distance computations for the same experiment. We can observe that the parallel costs do not decrease so significantly for larger radii

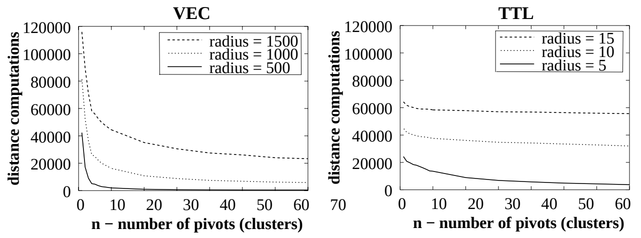

Figure 4. Number of pivots   total costs

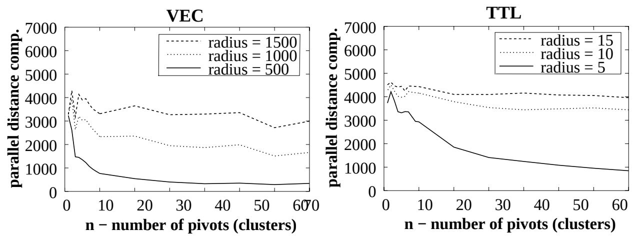

Figure 5. Number of pivots   parallel costs

because close objects are usually stored on a single node which is searched sequentially.

Figure 6 presents the maximal hop count (a) for **VEC** and the total number of messages (b) for **TTL**. The hop count is bigger for small number of clusters because the intervals of *M-Chord* domain to be visited by the query are longer. The hop count remains stable for higher values of n. Though the INTERVALSEARCH() request is sent for every cluster (see Section 3.3), the mentioned message passing optimization reduces the total number of messages and it does not grow as fast as the number of clusters.

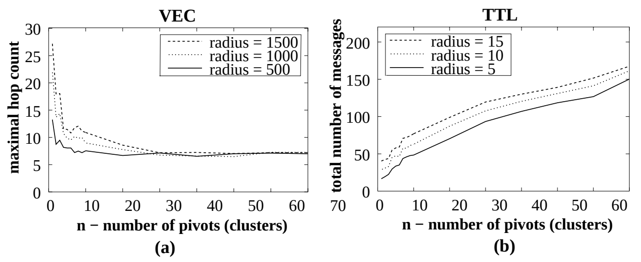

Figure 6. Number of pivots   maximal hop count (a) and total (b) number of messages

In summary, there is an obvious trade-off between the CPU costs and the messaging while increasing the number of pivots. In the following experiments, we fix the number of pivots to *forty* for both datasets.

4.4. Range queries

In this section, we analyze the performance of the $\mathbf{Range}(q, r)$ search algorithm more deeply. First, we monitor the algorithm while the query radius r grows, then we concern with the interquery parallelism of the query execution, and, finally, we study the scalability with respect to growing dataset.

Size of the query The dotted line in Figure 7 shows the number of retrieved objects while increasing the query radius r. Observe that, e.g., radius 2,000 for **VEC** retrieves about 8,000 objects on average and radius 20 for **TTL** 12,000 objects (6% of the whole database). Such queries are usually not reasonable for application (strings with edit distance 20 differ significantly) but we study the system performance even in these cases.

The solid line in Figure 7 depicts the parallel number of distance computations. The $\mathbf{Range}$ algorithm is designed so that the query replicas are always forwarded before the local storage is searched and any metric distance evaluated. Therefore, the parallel costs are upper bounded by the number of objects stored in a single peer (5,000 in our setting).

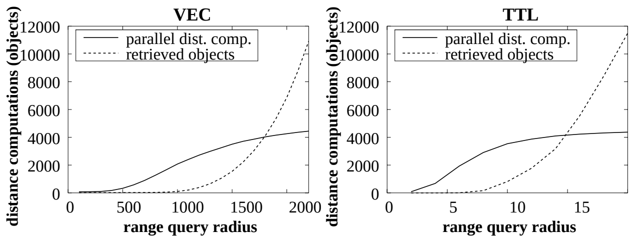

Figure 7. Size of the query   parallel costs

The parallel distance computations together with the maximal hop count (presented in Figure 8) can be considered as the characterization of the actual response time of the query. For example, running a query $\mathbf{Range}(q, 15)$ for **TTL**, we retrieve approx. 4,000 objects on average and the longest branch of the query execution maximally consists of nine forwardings of the query replicas and 4,000 distance computations.

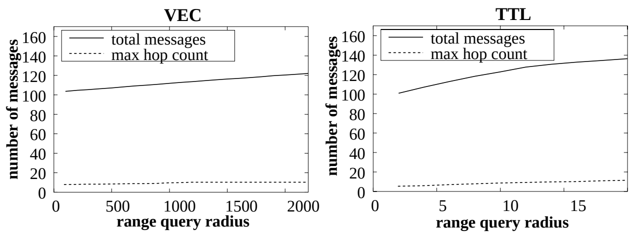

Figure 8. Size of the query   messages sent

Interquery parallelism In the previous experiment, we focused on the *intraquery* parallelism, i.e. parallel processing of a single query. The *interquery* parallelism refers to the ability of the system to accept multiple queries at the same time. In the following experiment, we executed l queries simultaneously each from a different node. We measured the *overall parallel costs* of the set of the l queries as the maximal number of distance computations performed on a single node of the system (*inter-cost*(l)).

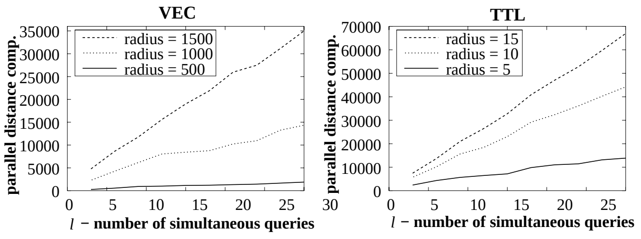

Figure 9. Overall parallel costs

Figure 9 presents the *inter-cost*(l) with respect to number of simultaneous queries l. The baseline of this experiment (a system with zero level of interquery parallelism) is *inter-cost*$(l) \approx l \times$ *parallel-cost* of one query (the parallel cost of l queries executed one by one).

For example, considering the **VEC** results for radius $r = 1{,}000$, *inter-cost*$(30) \approx 15{,}000$ while the parallel cost of one-by-one execution is approx. $30 \times 2{,}200 = 66{,}000$ distance computations (Figure 7). The *inter-cost* for **TTL** is higher because the **TTL** *total costs* are higher (Figure 4).

The following example brings another point of view on the way in which the computations are spread over the nodes. The sum of the total costs of thirty $\mathbf{Range}(q, 10)$ queries is approx. $30 \times 38{,}000 = 1{,}140{,}000$ (Figure 4 for **TTL** and forty pivots). The *average* number of distance computations on the sixty active nodes is, therefore, $1{,}140{,}000/60 = 19{,}000$. The *maximal* number of computations per node is 40,000 (*inter-cost*(30), $r = 10$) which indicates quite balanced computational load of the nodes.

Size of the dataset The following set of experiments concerns the $\mathbf{Range}$ search scalability with respect to growing dataset size up to 1,000,000 objects for **VEC** and 800,000 objects for **TTL**. The number of active nodes grows from approx. thirty for 100,000 objects to approx. 300 for 1,000,000 objects so the average storage load of 66% remains stable. Figure 10 shows the percentage number of nodes actively visited by the search algorithm. This value decreases with growing data size because the indexed space becomes more dense and, therefore, nodes cover smaller regions of the data space $\mathcal{D}$. During the query execution, some nodes of the structure are used only for routing and these are not considered actively visited.

Figure 11 shows the parallel number of distance computations the main factor influencing the query response time. Observe that it remains quite stable with the upper

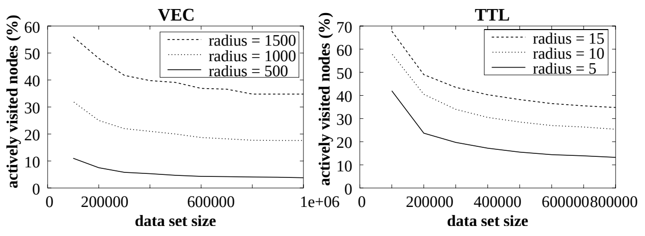

Figure 10. Size of the dataset visited nodes

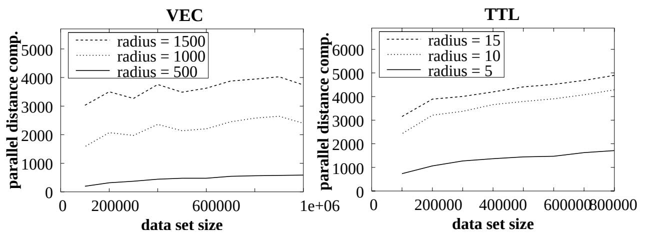

Figure 11. Size of the dataset parallel costs

bound of the size of the data stored by individual nodes (5,000 in our setting). The visible fluctuations are caused by the nodes' splitting.

The storage capacity limit for individual nodes is one of the parameters that can be utilized for tuning of the system performance. By decreasing this limit, the parallel number of distance computations can be lowered if there is sufficient number of available nodes to spread the data over.

The price to be paid is the increase of the number of messages. Figure 12 shows the maximal hop count while increasing the dataset size. The value slowly grows because, while size of the particular clusters grows, the sequential walk within the cluster interval gets longer. This is a weak point of the *M-Chord* approach but, at the moment, we are implementing an improved message passing algorithm which should make the hop count stable without increasing the total number of messages sent.

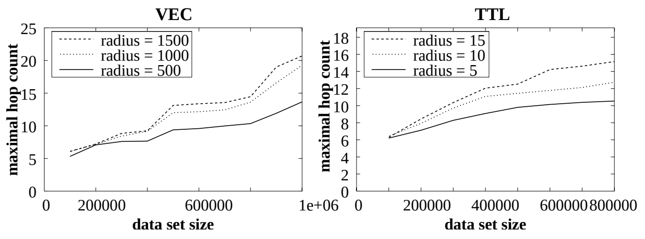

Figure 12. Size of the dataset hop count

4.5. Nearest-neighbors queries

The next set of experiments concerns with the performance of the $\mathbf{kNN}$ algorithm. The heart of the $\mathbf{kNN}(q, k)$ algorithm is running a $\mathbf{Range}(q, r)$ query for r determined in the initial phase. In the ideal case, the algorithm would estimate accurately the distance r_k to the k^{th} nearest neighbor of q and run $\mathbf{Range}(q, r_k)$. Let us denote such a query as the $\mathbf{Range}$ query *corresponding* to the $\mathbf{kNN}$ query.

Basically, there are three factors increasing the costs of a $\mathbf{kNN}$ compared to its corresponding $\mathbf{Range}$ query:

- costs of the initial phase,
- the determined radius r may be larger than r_k,
- multiple $\mathbf{Range}$ iterations could be necessary if less then k objects are retrieved in the first phase.

Figure 13 shows results of the experiment that compares the parallel costs of a $\mathbf{kNN}(q, k)$ query and of its corresponding $\mathbf{Range}$ (for growing k). The $\mathbf{kNN}$ costs are obtained as the sum of the parallel number of distance computations of the consecutive phases of the algorithm.

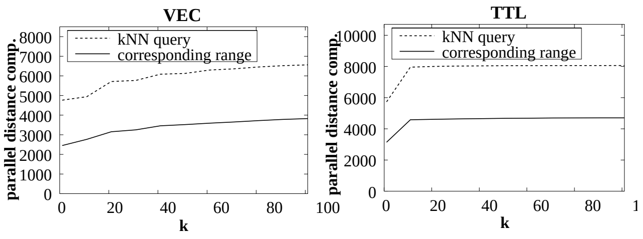

Figure 13. kNN parallel costs   increasing k

In most of the test cases, the initial phase has explored a significant part of a single node storage. The determined radius r has usually been very close to r_k and the answer set retrieved by the first phase was only slightly corrected in the second phase (the first phase result can be treated as the approximation of the precise answer). But the high costs of the initial phase influence negatively the parallel costs for $\mathbf{kNN}$ which can be observed in Figure 13.

There is a natural trade-off between the first phase cost and the demands of the second phase. Finding a suitable compromise is a matter of further testing and tuning of the system for a specific application.

Figure 14 gives comparison of the maximal hop count in the same experiment. As discussed above, this value is usually negatively influenced only by the hop-count of one *Chord* lookup operation during the $\mathbf{kNN}$ initial phase.

All trends observed in the $\mathbf{Range}$ experiments   impact of the number of pivots, interquery parallelism, scalability with respect to the dataset size   hold true for $\mathbf{kNN}$ experiments as well and we do not present these results for $\mathbf{kNN}$.

5. Related work

A number of distributed structures for interval queries in attribute-based (vector) data have been proposed. These

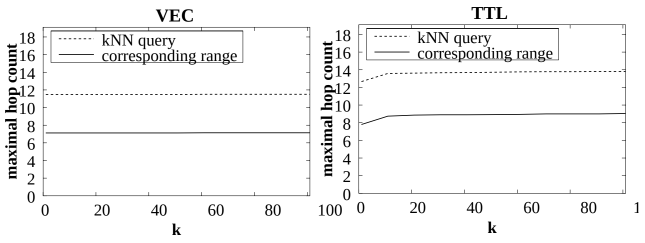

Figure 14. Maximal hop count for kNN

approaches are usually applicable and tested on data of low dimensions, they cannot manage inherently-metric data or data with non-standard distance measures, and they do not support similarity queries like nearest neighbors query.

The MAAN structure [8] extends the *Chord* protocol to support multi-attribute and range queries by means of uniform locality-preserving hashing expecting the exact knowledge of distributions of the attributes' domains. Ganesan et al. [12] use space-filling curves and the *Chord* protocol to support multi-dimensional range queries in the SCRAP structure. The authors also propose the MURK structure that adapts the kd-trees to support multi-dimensional interval queries in peer-to-peer environment. The Mercury [6] provides another protocol for routing multi-attribute range queries. This structure builds a *Chord*-like index for each attribute. The Skip Graphs [1] provide efficient routing mechanism while preserving the locality of data which enables an efficient support for range queries on one dimension.

Tanin et al. [19] introduce a P2P generalization of a *quadtree* index. The authors propose range and nearest neighbor algorithms over this structure [20]. The application of this structure is limited to *spatial domain*.

The pSearch approach [18] uses two information retrieval techniques *vector space model* and *latent semantic indexing* to build a P2P information retrieval system for top-k similarity queries. This concept, defining similarity between a document and a query by means of their common *terms*, is suitable for a specific application area only.

Banaei-Kashani and Shahabi [2] formalize the problem of attribute-based similarity search in P2P Data Networks and propose SWAM   a family of small-world based access methods. This concept provides a general solution for the range and nearest neighbors search in the vector data.

So far, the *GHT** [4, 3] and the *MCAN* [11] are the only metric-based distributed data structures published. The *GHT** is a *native metric* P2P structure utilizing the idea of Generalized Hyperplane Tree for data partitioning and for navigation. The *MCAN*, similarly to the *M-Chord*, first transforms the metric-space searching problem and then takes advantage of an existing solution   namely the *CAN* protocol. A very detailed comparison of our approach to these structures is provided in a separate publication [5].

6. Conclusions and future work

So far, the area of distributed data structures for metric data has not been investigated sufficiently. We consider this topic very relevant for many present-day applications that manage large volumes of complex digital data. We propose *M-Chord*, a peer-to-peer data structure for similarity search in metric spaces. It maps the data space into a one-dimensional domain by means of a generalized and improved variant of a vector index method *iDistance*. The indexed data is divided among the participating nodes and protocol *Chord* defines the topology of the structure and is used for navigation. Algorithms for the range and nearest-neighbors similarity queries are proposed.

We have conducted experiments on a prototype *M-Chord* implementation and on two real-life datasets. The presented results focus mainly on performance of the query processing and on scalability of the system from various points of view. The query processing trials have proved a good level of intraquery and interquery parallelism without any hot spots. The system scales well with the size of the query and with size of the dataset managed, assuming that enough computational resources are available. The system performance can be easily tuned through the nodes split policy.

The maximal hop count during the query processing grows with the number of nodes in the system. We are implementing a new *Chord*-based messages passing algorithm that is expected to stabilize the hop count while the system grows. Furthermore, our plans for future work cover implementation of the delete and update operations and support for proper departure of nodes from the system.

Although we provide a very detailed comparison of our approach to the most relevant structures in a separate publication [5], we plan to study the system behavior on simple vector data and compare it with vector-based index structures, e.g., Mercury [6]. Finally, we would like to investigate deeply the influence and behavior of various load-balancing and replication strategies.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384 393, 2003.

[2] F. Banaei-Kashani and C. Shahabi. SWAM: A family of access methods for similarity-search in peer-to-peer data networks. In *Proceedings of CIKM 2004*, pages 304 313. ACM Press, 2004.

[3] M. Batko, C. Gennaro, and P. Zezula. A scalable nearest neighbor search in P2P systems. In *Proceedings of DBISP2P 2004, Toronto, Canada, Revised Selected Papers*, volume 3367 of *LNCS*, pages 79 92. Springer, 2004.

[4] M. Batko, C. Gennaro, and P. Zezula. Similarity grid for searching in metric spaces. In *DELOS Workshop: Digital Library Architectures*, volume 3664/2005 of *LNCS*, pages 25 44. Springer, 2005.

[5] M. Batko, D. Novak, F. Falchi, and P. Zezula. On scalability of the similarity search in the world of peers. In *Proceedings of INFOSCALE 2006, Hong Kong*. IEEE Computer Society, 2006.

[6] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353 366, 2004.

[7] B. Bustos, G. Navarro, and E. Chavez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357 2366, 2003.

[8] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A multi-attribute addressable network for grid information services. In *Proceedings of GRID 2003*, pages 184 191, Washington, DC, USA. IEEE Computer Society, 2003.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of VLDB 1997, Athens, Greece*, pages 426 435. Morgan Kaufmann, 1997.

[10] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl.*, 21(1):9 33, 2003.

[11] F. Falchi, C. Gennaro, and P. Zezula. A content-addressable network for similarity search in metric spaces. In *Proceedings of DBISP2P 2005, Trondheim, Norway*, pages 126 137, 2005.

[12] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In *Proceedings of WebDB 2004, New York, USA*, pages 19 24. ACM Press, 2004.

[13] A. K. Garg and C. C. Gotlieb. Order-preserving key transformations. *ACM Trans. Database Syst.*, 11(2):213 234, 1986.

[14] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517 580, 2003.

[15] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS 2005)*, 30(2):364 397, 2005.

[16] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8 17, 1965.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001, San Diego, USA*, pages 149 160. ACM Press, 2001.

[18] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. Technical report, HP Labs, 2002.

[19] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *ICDE*, pages 254 255. IEEE Computer Society, 2005.

[20] E. Tanin, D. Nayar, and H. Samet. An efficient nearest neighbor algorithm for p2p settings. In *Proceedings of the 2005 national conference on Digital government research*, pages 21 28. Digital Government Research Center, 2005.

[21] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer-Verlag, 2006.