A Scalable Nearest Neighbor Search in P2P Systems

Michal Batko[1], Claudio Gennaro[2], and Pavel Zezula[1]

[1] Masaryk University
Brno, Czech Republic
{xbatko,zezula}@fi.muni.cz
[2] ISTI-CNR
Pisa, Italy
gennaro@isti.cnr.it

Abstract. Similarity search in metric spaces represents an important paradigm for content-based retrieval of many applications. Existing centralized search structures can speed-up retrieval, but they do not scale up to large volume of data because the response time is linearly increasing with the size of the searched file. In this article, we study the problem of executing the nearest neighbor(s) queries in a distributed metric structure, which is based on the P2P communication paradigm and the generalized hyperplane partitioning. By exploiting parallelism in a dynamic network of computers, the query execution scales up very well considering both the number of distance computations and the hop count between the peers. Results are verified by experiments on real-life data sets.

1 Introduction

Peer-to-peer (P2P) communication has become a prospective concept for publishing and finding information on the ubiquitous computer networks today. Most P2P systems so far support only simple lookup queries, i.e. queries that retrieve all objects with a particular key value, for example [1] and [2]. Some recent work has extended this functionality to support *range queries* over a single attribute [3]. However, an increasing amount of data today can only be effectively searched through specific (relative) measures of similarity.

For example, consider a P2P photo-sharing application where each user publishes photographs tagged with color histograms as its metadata. A typical query in such a system would contain similarity predicates asking for photographs with color histograms which are not very different from the color histogram of the query photo sample.

The problem of retrieving elements from a set of objects that are close to a given query reference (using specific similarity criterion), has a lot of applications ranging from the pattern recognition to the textual and multimedia information retrieval. The most general abstraction of the similarity concept, which is still indexable, use the mathematical notion of the *metric space*.

The advantage of the metric space approach to the data searching is its "extensibility", because in this way, we are able to perform the *exact match*, *range*, and *similarity* queries on any collection of metric objects. Since any *vector space* is covered by a metric space with a proper distance function (for example the Euclidean distance), even the n-dimensional vector spaces are easily handled. Furthermore, there are numerous metric functions able to quantify similarity between complex objects, such as free text or multi-media object features, which are very difficult to manage otherwise. For example, consider the *edit distance* defined for sequences and trees, the *Hausdorff distance* applied for comparing shapes, or the *Jacard coefficient*, which is often used to asses similarity of sets. The problem has recently attracted a lot of researchers to develop techniques able to structure collections of metric objects in such a way so that the search requests are performed efficiently – see the recent surveys [4] and [5].

Though metric indexes on single computers are able to speedup query execution, the processing time is not negligible and it grows linearly with the size of the searched collection. Such property has recently been confirmed by numerous experiments in [6]. The evaluation of metric distance functions can also take a considerable amount of computational time. To search the partitioned space, we usually have to compute distances (using the metric function) between many objects. The time to compute a distance can also depend on the size of compared objects. For example, the *edit distance* of two strings has the computational complexity $O(n \cdot m)$, where n and m represent the number of characters in the compared strings.

The distributed computer environment of present days is a suitable framework for the parallel execution of queries. With such infrastructure, parallel distance computations would enhance the search response time considerably. Modern computer networks have a large enough bandwidth, so it is becoming more expensive for an application to access a local disk than to access the RAM of another computer on the network. In this paper, we try to apply current approaches to the distributed data processing – *Scalable and Distributed Data Structures*, SDDS, and *Peer to Peer*, P2P, communication – to the metric space indexing. The motivation and the basic concepts of our proposal have been published in [7] considering only the *similarity range* queries. In this paper we show how such idea can be extended to the important case of similarity predicates, specifically the *nearest neighbors* queries.

The rest of the paper is organized as follows. In Section 2, we summarize the principles of metric queries, while in Section 3 we introduce a formal definition of the distributed metric index. Section 4 describes the strategies for the nearest neighbor search, which are experimentally evaluated in Section 5. The paper concludes in Section 6.

2 Metric Space and Similarity Queries

The mathematical metric space is a pair $(\mathcal{D}, d)$, where $\mathcal{D}$ is the *domain* of objects and d is the *distance function* able to compute distances between any pair of

objects from $\mathcal{D}$. It is typically assumed that the smaller the distance, the closer or more *similar* are the objects. For any distinct objects $x, y, z \in \mathcal{D}$, the distance must satisfy the following properties:

$$\begin{array}{ll} d(x,x) = 0 & \textit{reflexivity} \\ d(x,y) > 0 & \textit{strict positiveness} \\ d(x,y) = d(y,x) & \textit{symmetry} \\ d(x,y) \leq d(x,z) + d(z,y) & \textit{triangle inequality} \end{array}$$

Let $\mathcal{F} \subseteq \mathcal{D}$ be the data-set. There are two basic types of similarity queries. The **range** query retrieves all objects which have a distance from the query object $q \in \mathcal{D}$ at most the specified threshold (range or radius) ρ.

$$\{x \in \mathcal{F} \mid d(q,x) \leq \rho\}$$

The **nearest neighbor** query returns the object that is the nearest (having the shortest distance) to the query object q. We can extend this type of query to return k nearest objects that form a set $\mathcal{K} \subset \mathcal{F}$ defined as follows:

$$|\mathcal{K}| = k \wedge \forall x \in \mathcal{K}, y \in \mathcal{F} - \mathcal{K} : d(q,x) \leq d(q,y)$$

Other forms of similarity queries concern the **similarity joins** and the **reverse nearest neighbor** queries. In this article, we concentrate on the most frequent (the most natural) form of similarity queries that is the nearest neighbors queries.

3 Principles of GHT*

The GHT* structure was proposed in [7] as a distributed metric index for similarity range queries. In this paper, we provide a new formalization of this approach and extend the search capability of the GHT* to processing the nearest neighbors queries.

3.1 Architecture of GHT*

In general, the scalable and distributed data structure GHT* consists of network nodes, peers, that can insert, store, and retrieve objects using similarity queries. The GHT* architecture assumes that:

- Peers communicate through the *message passing* paradigm. For consistency reasons, each *request message* expects a confirmation by a proper *acknowledgment message*.
- Each peer participating in the network has a unique *Network Node IDentifier* (NNID).
- Each peer maintains data objects in a set of *buckets*. Within a peer, the *Bucket IDentifier* (BID) is used to address a bucket.
- Each object is stored exactly in one bucket.

An essential part of the GHT* structure is the *Address Search Tree* (AST). In principle, it is a structure based on the Generalized Hyperplane Tree (GHT) [8], which is one of the centralized metric space indexing structures. In the GHT*, the AST is used to actually navigate to the (distributed) buckets when data objects are stored and retrieved.

3.2 Address Search Tree

The AST is a binary search tree, where its inner nodes hold routing information and the leaf nodes represent pointers to the data. Specifically, the inner nodes always store a pair of pivots – these are some representative metric objects from the data-set – and respective pointers to the left and the right subtrees. An example of AST can be seen in Figure 1b. When searching for a place where to store a new object, we start in the root and compute distances between the inserted object and the pivots. If the distance to the first pivot is smaller than the distance to the second pivot, we navigate to the left subtree of that inner node, otherwise, the right subtree is considered. This process is recursively repeated until a leaf node is reached.

The data objects are stored in buckets that are held either locally (thus we can address the bucket by its BID) or on another peer, which can be identified by a proper NNID. Therefore, the AST has always one of those two types of pointers in leaf nodes. Whenever the navigation procedure reaches the leaf node of the AST, the inserted object is stored either locally in the respective bucket (if a BID identifier is found) or on a remote peer (if an NNID identifier is encountered).

In order to avoid hot-spots caused by the existence of a centralized node accessed by every request, a form of the AST structure is present in every peer. Due to the autonomous update policy, the AST structures in individual peers may not be identical – with respect to the complete tree view, some sub-trees may be missing. However, the GHT* provides a mechanism for updating the AST automatically during the insertion or search operations.

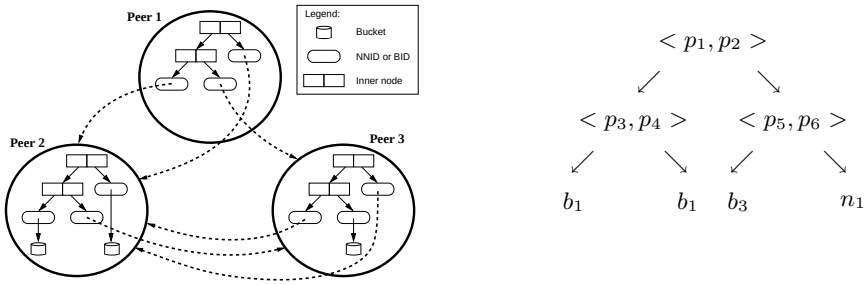

Fig. 1. The GHT* network (a) and an example of Address Search Tree (b)

Figure 1a illustrates the AST structure in a network of three peers. The dashed arrows from the leaves indicate the NNID pointers while the solid arrows

represent the BID pointers. In case of identical ASTs on all peers, each peer knows all its neighbors having data and the proper routing is done just in one step. However, the growth of replication is linear. The GTH* is also able to use the so called *logarithmic replication scheme* where the replicated data grows in a logarithmic way. In this case, each peer knows only few neighbors by keeping complete AST paths only to its local buckets. With such scheme, the routing is logarithmic in the worst case. In the following, we introduce a notation and operators, which will help us to define the insertion and search algorithms more precisely.

Address Search Tree Notation

Definition 1. *Suppose that $\mathbb{L}_{BID}$ is the set of all possible BIDs and $\mathbb{L}_{NNID}$ is the set of all possible NNIDs. The set of all possible ASTs $\mathbb{T}_{AST}$ of the data-set $\mathcal{F}$ is formed by the following rules:*

- *$\mathbb{L}_{BID} \subset \mathbb{T}_{AST}$; i.e., every BID pointer is a legal AST.*
- *$\mathbb{L}_{NNID} \subset \mathbb{T}_{AST}$; i.e., every NNID pointer is a legal AST.*
- *Let $\langle p_1, p_2 \rangle \in \mathcal{F} \times \mathcal{F}$. Let $T_l \in \mathbb{T}_{AST}$ and $T_r \in \mathbb{T}_{AST}$. Then the triple $\langle \langle p_1, p_2 \rangle, T_l, T_r \rangle \in \mathbb{T}_{AST}$.*
- *The set $\mathbb{T}_{AST}$ contains nothing else.*

Observe that every $T \in \mathbb{T}_{AST}$ is a rooted binary tree, where *leaf nodes* are elements of $\mathbb{L}_{BID}$ or $\mathbb{L}_{NNID}$ and the *inner nodes* are the pairs $\langle p_1, p_2 \rangle$. Every inner node contains two pointers: one to the *left subtree* (T_l) and one to the *right subtree* (T_r). For example, given the metric objects $p_1, p_2, p_3, p_4, p_5, p_6$, the BIDs b_1, b_2, b_3, and the NNID n_1, a possible address search tree $T \in \mathbb{T}_{AST}$ of three levels could be: $\langle \langle p_1, p_2 \rangle, \langle \langle p_3, p_4 \rangle, b_1, b_2 \rangle, \langle \langle p_5, p_6 \rangle, b_3, n_1 \rangle \rangle$, as Figure 1b illustrates.

Definition 2. *Let $r(T)$ be the function that returns the root node of the tree $T \in \mathbb{T}_{AST}$. In particular, the node returned by $r(T)$ is a pair $\langle p_1, p_2 \rangle$, a BID pointer, or a NNID pointer.*

Definition 3. *We represent a path of a generic tree $T \in \mathbb{T}_{AST}$, called* BPATH*, with a string of n binary elements $\{0, 1\}$: $p = (b_1, b_2, \ldots, b_n)$. Given a BPATH p, the function $S(T, p)$ returns the subtree reached by p on the tree T, as in the following*

$$S(T, ()) = T$$

$$S(\langle \langle p_1, p_2 \rangle, T_l, T_r \rangle, (b_1, \ldots, b_n)) = \begin{cases} S(T_l, (b_2, \ldots, b_n)) & \text{if } b_1 = 0 \\ S(T_r, (b_2, \ldots, b_n)) & \text{if } b_1 = 1 \end{cases}$$

Let $p = (p_1, \ldots, p_n)$ and $s = (s_1, \ldots, s_m)$ be two BPATHs, the concatenation operator $+$ is defined as $p+s = (p_1, p_2, \ldots, p_n, s_1, s_2, \ldots, s_m)$. The concatenation operator can be easily extended for sets of BPATHs. Let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of BPATHs, then $p + Q = \{p + q_1, p + q_2, \ldots, p + q_n\}$.

Pruning mechanism In order to search in an AST T, we need an algorithm able to execute a query in the tree. For this purpose we define the traversing operator Ψ. In principle, the operator examines every inner node (i.e. nodes with pairs of two pivots $\langle p_1, p_2 \rangle$) and it decides which subtree to follow. Such pruning is based on the well known generalized hyperplane principles from [8].

Definition 4. *Given an AST T, a metric object q, and a non negative real number ρ, the traversing operator $\Psi(T, q, \rho)$ returns a set of BPATHs as follows:*

$$\begin{aligned}
\Psi(l_{bid}, q, \rho) &= \{()\} \\
\Psi(l_{nnid}, q, \rho) &= \{()\} \\
\Psi(\langle\langle p_1, p_2\rangle, T_l, T_r\rangle, q, \rho) &= \begin{cases}
(0) + \Psi(T_l, q, \rho) & \text{if } d(p_1, q) - \rho \leq d(p_2, q) + \rho \\
(1) + \Psi(T_r, q, \rho) & \text{if } d(p_1, q) + \rho > d(p_2, q) - \rho \\
(0) + \Psi(T_l, q, \rho) & \\
\quad \cup & \text{if both conditions } qualify \\
(1) + \Psi(T_r, q, \rho) &
\end{cases}
\end{aligned}$$

The algorithm Ψ works as follows. If T is only composed of a node n (the first two conditions of the definition), $\Psi(T, q, \rho)$ corresponds to a single empty BPATH (). For the other cases, the algorithm recursively traverses T, on the basis of the query range (q, ρ), as in the search algorithm of the GHT. If for the root node $\langle p_1, p_2 \rangle$ the condition $d(p_1, q) - \rho \leq d(p_2, q) + \rho$ holds, we concatenate all the BPATHs of the $\Psi(T_l, q, \rho)$ to the simple BPATH (0). Whenever $d(p_1, q) + \rho > d(p_2, q) - \rho$, we concatenate all the BPATHs of the $\Psi(T_r, q, \rho)$ to the BPATH (1). Note that, the conditions can be met simultaneously. When the radius $\rho = 0$, which corresponds either to the exact match query or to the process of insertion, a single BPATH is returned.

3.3 Insert and Range Search Algorithms

Insertion of an object starts at the peer asking for insertion by traversing its AST from the root to a leaf using the function Ψ with $\rho = 0$. If a BID pointer is found, the inserted object is stored in this bucket. Otherwise, the found NNID pointer is applied to forward the request to the proper peer where the insertion continues recursively until an AST leaf with the BID pointer is reached.

Algorithm 1
procedure *Insert(x, p)*
    $S_p = S(T, p)$;
    $\{p_1\} = \Psi(S_p, x, 0)$;
    $n = r(S(S_p, p_1))$;
    if $n \in \mathbb{L}_{NNID}$ ***then***
        send a request for Insert(x,p_1) to peer with NNID n;
    if $n \in \mathbb{L}_{BID}$ ***then***
        insert x in local bucket with BID n;

Algorithm 1 formalizes this insertion procedure, where x is the inserted object and p represents the path in the AST traversed so far (using the BPATH notation), which is initially empty, i.e. $p = ()$. If the search is forwarded to another peer, the p parameter contains the BPATH already traversed by the sending peer.

Algorithm 2
procedure *RangeSearch(q,ρ,p)*
    $S_p = S(T, p)$*;*
    $P = \Psi(S_p, q, \rho)$*;*
    for each $p_i \in P$
        $n = r(S(S_p, p_i))$*;*
        if $n \in \mathbb{L}_{NNID}$ ***then***
            send a request for RangeSearch(q,ρ,p_i*) to peer with NNID n;*
        if $n \in \mathbb{L}_{BID}$ ***then***
            search (q, ρ) *in local bucket with BID n;*
    end for each

By analogy to insertion, the *range search* also starts by traversing the local AST of the querying peer. The AST is traversed by using the operator Ψ with the query object q and the search radius ρ. As already explained, the function Ψ can assign both the sub-trees as qualifying. For all qualifying paths having a NNID pointer in their leaves, the query request is recursively forwarded (including its known BPATHs) to the respective peers until a BID pointer occurs in every leaf.

4 Searching for Nearest Neighbors

Whenever we want to search for similar objects using the range search, we must specify the maximal distance of objects that qualify. However, it can be very difficult to specify the radius without some knowledge about the data and the used metric space. For example, the range $\rho = 3$ of the edit distance metric represents less than four edit operations between the two strings, which has a clear semantic meaning. However, a distance of two color-histogram vectors of images is a real number, which cannot be so easily quantified. When a too small query radius is specified, the result set can even be empty and a new search with a larger radius is needed to get a result. On the other hand, queries with too large query radii might be computationally expensive, and the response sets might contain many not significant objects.

An alternative way to search for similar objects is to use the nearest neighbors queries. Such queries guarantee the retrieval of k most relevant objects, that is the set of k objects with the shortest distances to the query object q. Though the problem of executing k nearest neighbors (kNN) queries is not new and many algorithms have been proposed in the literature, see for example [9] for many references and additional readings, the distributed kNN query processing have not been systematically studied.

4.1 kNN Search in GHT*

In principle, there are two basic strategies how the kNN queries can be evaluated. The first strategy starts with a very large query radius, covering all the data in a given data-set, to identify a degree to which specific regions might contain searched neighbors. Such information is stored in a priority stack (queue) so that the most promising regions are accessed first. As objects are found, the search radius is reduced and the stack adjusted accordingly. Though this strategy never accesses regions not intersecting the query region that is bound by the distance to the k-th nearest neighbor of the query, the processing of regions is strictly serial. On a single computer, the approach is optimum [9], but not convenient for distributed environments aiming at exploiting parallelism. The second strategy starts with the zero radius to locate the first region to explore and then extends the radius to locate other candidate regions, if the result set is still not correct. In this article, we adopt the second approach.

4.2 kNN Algorithm

In our algorithm, we first search for the bucket with a high probability of the occurrence of the nearest neighbors. In particular, we access the bucket in which the query object would be stored using the insert operation. In the accessed bucket, we sort its objects according to their distances with respect to q. Assume that there are at least k objects in the bucket, so the first k objects, i.e. the objects with the shortest distances to q, are the candidates for the result. However, there may be other objects in different buckets that are closer to the query than some of the candidates. In order to control, we issue a similarity range search with the radius equal to the distance of the k-th candidate. In this way, we get a set of objects of the cardinality always greater than or equal to k. If we sort all the retrieved objects and retain the first k with the shortest distances, we get the exact answer to the query.

If less than k objects are found in the first accessed bucket, other strategy must be applied because we do not know the upper bound on the distance to the k-th nearest neighbor. We again execute the range search operation, but we have to estimate the radius. If we get enough objects from the range query (at least k), we are done – the result is again the first k objects from the sorted result of the range search. Otherwise, we have to expand the radius and try again (i.e. iterate) until enough objects are received. There are two possible strategies how to estimate the radius.

Optimistic strategy The objective is to minimize the costs, i.e. the number of accessed buckets and distance computations, by using a not very large radius, at the risk that more iterations are needed if not enough objects are found. In the first iteration we use the bounding radius of the candidates, i.e. the distance to the last candidate, despite of the fact that we have less than k candidates. The optimistic strategy hopes that there are enough objects in the other buckets within this radius. Let x be the number of objects returned from the last range

query. If $x \geq k$, we are done, otherwise, we expand the radius to $\rho + \rho\frac{k-x}{k}$ and iterate again.

Pessimistic strategy The estimated radius is chosen rather large so that the probability of next iteration is minimum, risking excessive (though parallel) bucket accesses and distance computations. To estimate the radius, we use the distance between pivots of the inner nodes, because the pivot selection algorithm (described in [7]) chooses pivots as the most distant pair of objects available. More specifically, the pessimistic strategy traverses the AST from the leaf with the pointer to the bucket up to the tree root and applies the distance between pivots as the range radius. Every iteration climbs one level up in the AST until the search terminates or the root is encountered. If there are still not enough retrieved objects, the maximum distance of the metric is used and all the objects in the structure are evaluated.

Algorithm 3

procedure *kNN(q,k,p)*
    $S_p = S(T, p)$*;*
    $\{p_1\} = \Psi(S_p, x, 0)$*;*
    $n = r(S(S_p, p_1))$*;*
    if $n \in \mathbb{L}_{NNID}$ ***then***
        send a request for kNN(q,p_1*) to peer with NNID n;*
    if $n \in \mathbb{L}_{BID}$ ***then***
        compute distances to all objects in local bucket with BID n;
        A = sort object using the distances, smallest first;
        do
            ρ = EstimateRadius(); // Using some strategy
            O = RangeSearch(q, ρ, ());
            insert sort objects O into A using distances computed by the range search;
        repeat until $|A| < k$
    end if

4.3 Implementation issues

As an extension of the algorithms above, several optimization strategies have been implemented. To avoid multiple accesses to the same buckets, the so called BPATH sets are used during the range searches in the kNN iterations. In particular, if the distances to all objects in a bucket are evaluated during a range search, this bucket is never accessed again in the following iterations. Naturally, the first bucket (the one with candidate objects) is never searched twice.

If objects are sent during the query evaluation, the computed distances are always appended. Therefore the peer, which is sorting the result set, never repeats distance computations and only performs a rather quick merge sort of the distances. Unless necessary, we send between peers object identities and not the whole objects, which can be large. Recall that a range search can return more than k objects. In this way, the peer which initiated the query, only receives the matching objects and not all the intermediate results.

5 Performance Evaluation

In this section, we present results of performance experiments that assess different aspects of our GHT* prototype implemented in Java. We have conducted our experiments on two real-life data-sets. First, we have used a data-set of 45-dimensional vectors of color image features with the L_2 (Euclidian) metric distance function (VEC). This data-set have a normal distribution of distances and every object has the same size (45 numbers). The second data-set is formed by sentences of the Czech national corpus with the *edit distance* function as the metric (TXT). The distribution of distances in this data-set is rather skewed – most of the distances are within a very small range of values. Moreover, the size of sentences varies significantly. There are sentences with only a few words, but also quite long sentences with tenths of words.

For the experimental purposes, we have used 100 independent computers (peers) connected by a high-speed network. Essentially, every peer provides some computation capacity, which is used during the object insertion and the query evaluation. We have used some of the peers to insert the data and the others to execute queries. The number of peers storing the data was automatically determined by the size of the data-set, because the number and capacity of buckets on individual peers was constant.

Notice that our prototype uses the simplest bucket structure – a linked list of objects – which needs to examine every object in a bucket in order to solve the query. By applying a metric index on individual peers, the performance would significantly improve.

5.1 Performance Characteristics

In our experiments, we have measured different performance related characteristics of the query evaluation. In order to quantify the CPU costs, we have counted the number of distance computations necessary to evaluate a nearest neighbor query. The number is the sum of the computations incurred during the navigation (i.e. while searching in the AST) and the computations necessary to evaluate the query in the accessed buckets. The total number of distance computations corresponds to the number of distance computations that would be needed in a centralized environment. The parallel cost is the maximal number of distance computations evaluated on a peer or a set of peers accessed serially. As we have already explained, the evaluation algorithm for a kNN query consists of sequential steps. At the beginning we have to find the first bucket and examine its objects (see Section 4.2), then, we iterate using the range search with the estimated radii. Naturally, the distance computations evaluated during these steps must be considered serial.

We also measured the number of accessed buckets, which are of a limited capacity. In our experiments, we have used a maximum of 2,000 objects per bucket and maximally 5 buckets per a peer. The average bucket occupation was about 50%. We have measured the total number of buckets accessed during a query and the number of buckets accessed per a peer.

Finally, we have measured the number of messages exchanged between peers. The total number of messages can be seen as a representation of the network load. However, most of the messages are sent in parallel, because one peer can send a message to multiple peers. In specific situations, a peer must forward a message to a more appropriate peer. The number of those forwardings is usually called the *hop count*. In our experiments, we have measured the maximal hop count to execute a query. The hop count represents the sequential part of the message passing process, i.e. its parallel cost.

We do not use the execution time of the query evaluation as the relevant measure, because there are many factors (such as the speed of peer processors, the congestion of the network, the load of peers, etc.) that can directly influence the execution time of a query.

For comparison reasons, we also provide the costs of a range search for every experiment. The radius ρ is adjusted so that it represents the minimal bounding radius of the set of objects returned by the corresponding kNN query.

Every value in the graphs represents the average obtained from execution of 50 queries with different query objects and fixed k. We only show results for the pessimistic strategy, but with 1,000 objects per bucket on average, the strategy was only applied for evaluating queries with $k > 1,000$. In such situation, nearly all the objects in the data-set had to be accessed to solve the query. Therefore, the performance of the optimistic strategy was practically the same.

5.2 kNN Search Performance

In this set of experiments, we have analyzed the performance of the kNN search with respect to different k on data-sets of $10,000$ objects. Results are summarized as graphs in Figure 2 for the VEC and Figure 3 for the TXT data-sets. Our experiments show that the parallel costs of our kNN queries remain quite stable for the increasing k while the total costs (note that graph values are divided by 10 and the x axis has a logarithmic scale) grow very quickly with the number of neighbors k. Note that, for k values greater than 100 for VEC (and values over 10 for TXT) almost all the objects had to be accessed and the distances to the query objects computed. However, this is not a general observation and it is strictly dependent on the distance distribution in the processed data. In the figures, we also show the costs of the corresponding range search, that is the range search with the radius equal to the distance of the k-th nearest neighbor. Naturally, the performance is better, but the overhead incurred by the kNN algorithm seems to be constant, not dependent on the value of k.

5.3 kNN Search Scalability

The effect of growing data-sets on the performance of queries (i.e. the scalability) is usually the worst problem with the centralized metric indices. For example, experiments with the D-Index [6] structure using the same TXT data-set have shown that a kNN query takes around 4 seconds for the data-set size of 10,000 and about 40s seconds for the size of 100,000 objects (sentences) – the increase

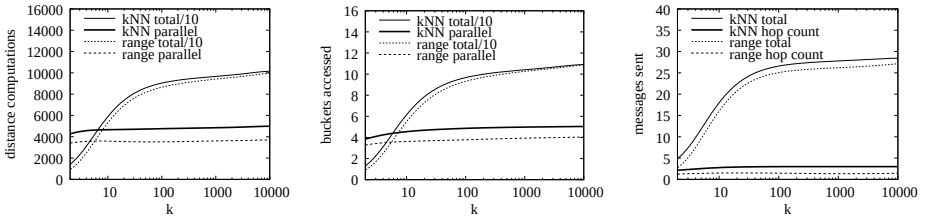

Fig. 2. Dependence of different costs on k for the VEC data-set.

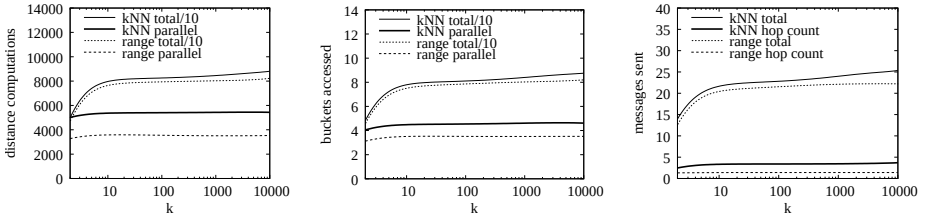

Fig. 3. Dependence of different costs on k for the TXT data-set.

of the search costs in a centralized structure is linear with respect to the size of the data-set. Our experiments with the GHT* exhibit nearly constant parallel costs even when the data-set grows. Contrary to the D-Index, we have always achieved the response time around 2 seconds.

The leftmost graphs in Figures 4 and 5 confirm the scalability of the GHT* considering the number of distance computations – the cost around 4,000 computations remains stable even for the full data-set of $100,000$ objects. The middle graphs show the number of forwarded messages, which in fact represent the number of peers actually addressed. This number is increasing, because more peers are used to store the data, therefore more peers have to be accessed in order to solve the query. The hop count, shown in the last graph, is slowly rising with the growing data-set. Compared to the range search, the values for the kNN are higher, because there is always the overhead with locating the first bucket. The observed increase of the number of hops seems to be logarithmic.

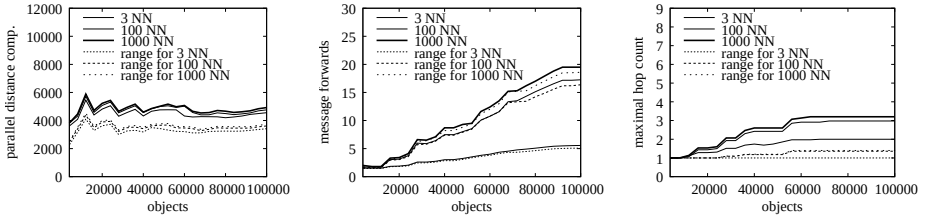

Fig. 4. The scalability of GHT* while resizing the VEC data-set

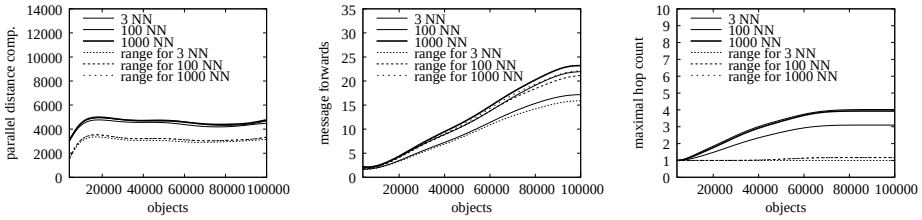

Fig. 5. The scalability of GHT* while resizing the TXT data-set

6 Conclusions

To the best of our knowledge, the problem of distributed index structures supporting the execution of the nearest neighbors queries on metric data sets has not been studied yet. The GHT* structure stores and retrieves data from domains of arbitrary metric spaces and satisfies all the necessary conditions of the scalable and distributed data structures. It is scalable in that it distributes the structure over more and more independent peer computers. The parallel search time for kNN queries becomes practically constant for arbitrary data volume, and the hop count grows logarithmically. It has no hot spots – all clients and servers use as precise addressing scheme as possible and they all incrementally learn from misaddressing during insertion or search. Finally, node splits are performed locally without sending multiple messages to many other peers.

Our future work will concentrate on strategies for updates (object deletion), pre-splitting policies, and more sophisticated strategies for organizing buckets. An interesting research challenge is to consider other metric space partitioning schemes (not only the generalized hyperplane) and study their suitability for implementation in distributed environments.

References

1. Litwin, W., Neimat, M.A., Schneider, D.A.: LH* - a scalable, distributed data structure. ACM TODS **21** (1996) 480–525
2. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proc. of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. (2001) 161–172
3. Li, X., Kim, Y.J., Govindan, R., Hong, W.: Multi-dimensional range queries in sensor networks. In: Proceedings of the First International Conference on Embedded Networked Sensor Systems. (2003) 63–75
4. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys **33** (2001) 273–321
5. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Trans. Database Syst. **28** (2003) 517–580
6. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications **21** (2003) 9–13

7. Batko, M., Gennaro, C., Savino, P., Zezula, P.: Scalable similarity search in metric spaces. In: Digital Library Architectures: Peer-to-Peer, Grid, and Service-Orientation, Pre-proceedings of the Sixth Thematic Workshop of the EU Network of Excellence DELOS, S. Margherita di Pula, Cagliari, Italy, 24-25 June, 2004, Edizioni Libreria Progetto, Padova (2004) 213–224
8. Uhlmann: Satisfying general proximity / similarity queries with metric trees. IPL: Information Processing Letters **40** (1991) 175–179
9. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24** (1999) 265–318