Scalable Similarity Search in Metric Spaces

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

[1] Masaryk University, Brno, Czech Republic
{zezula, xbatko}@fi.muni.cz
[2] ISTI-CNR, Pisa, Italy
{claudio.gennaro, pasquale.savino}@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. The proposed GHT* index is a scalable and distributed structure. By exploiting parallelism in a dynamic network of computers, the GHT* achieves practically constant search time for similarity range queries in data-sets of arbitrary size. The amount of replicated routing information on each server increases logarithmically. At the same time, the potential for interquery parallelism is increasing with the growing data-sets because the relative number of servers utilized by individual queries is decreasing. All these properties are verified by experiments on a prototype system using real-life data-sets.

1 Introduction

The search operation has traditionally been applied to structured (attribute-type) data. Complex data types – such as images, videos, time series, text documents, DNA sequences, etc. – are becoming increasingly important in modern digital libraries. Searching in such data requires a gradual rather than the exact relevance, so it is called the *similarity* retrieval. Given a query object q, this process involves finding objects in the database D that are similar to q. It has become customary to assume the similarity measure as a distance metric d. The primary challenge in performing similarity search is to structure the database D in such a way so that the search can be performed fast.

Though many metric index structures have been proposed, see the recent surveys [2] and [6], most of them are only main memory structures and thus not suitable for a large volume of data. The scalability of two disk oriented metric indexes (the M-tree [3] and the D-index [5]) have recently been studied in [4]. The results demonstrate significant speed-up (both in terms of distance computations and disk-page reads) in comparison with the sequential search. Unfortunately, the search costs are also linearly increasing with the size of the data-set.

On the other hand, it is estimated that 93% of data now produced is in a digital form. The amount of data added each year exceeds exabyte (i.e. 10^{18}

bytes) and it is estimated to grow exponentially. In order to manage similarity search in multimedia data types such as plain text, music, images, and video, this trend calls for putting equally scalable infrastructures in motion. In this respect, the Grid infrastructures and the Peer-to-Peer (P2P) communication paradigm are quickly gaining in popularity due to their scalability and self-organizing nature, forming bases for building large-scale similarity search indexes at low costs. However, most of the numerous P2P search techniques proposed in the recent years have focused on the single-key retrieval. See for example the *Content Addressable Network* (CAN) [9], which is a distributed hash table abstraction over the Cartesian space.

Our objective is to develop a distributed storage structure for similarity search in metric spaces that would scale up with (nearly) constant search time. In this respect, our proposal can be seen as a *Scalable and Distributed Data Structure* (SDDS), which uses the P2P paradigm for the communication in a Grid-like computing infrastructure. We achieve the desired effects in a given (arbitrary) metric by linearly increasing the number of network nodes (whole computers), where each of them can act as a client and some of them can also be servers. Clients insert metric objects and issue queries, but there is no specific (centralized) node to be accessed for all (insertion or search) operations. At the same time, insertion of an object, even the one causing a node split, does not require immediate update propagation to all network nodes. A certain data replication is tolerable. Each server provides some storage space for objects and servers also have the capacity to compute distances between pairs of objects. A server can send objects to other peer servers and can also allocate a new server.

The rest of the paper is organized as follows. In Section 2, we summarize the necessary background information. Section 3 presents the GHT* distributed structure and its functionality. Section 4 reports some results of our performance evaluation experiments. Section 5 concludes the paper and outlines directions for future work.

2 Preliminaries

Probably the most famous scalable and distributed data structure is the LH* [8], which is an extension of the *linear hashing* (LH) for a dynamic network of computers enabling the *exact-match* queries. The paper also clearly defines the desirable properties of SDDSs in terms of *scalability*, *no hot-spots*, and *update independence*.

2.1 Metric Space Searching Methods

The effectiveness of metric search structures [2,6] consists in their considerable *extensibility*, i.e. the degree of ability to support execution of a variety of queries. Metric structures can support not only the exact-match and range queries on sortable domains, but they are also able to perform similarity queries in the

generic metric space, considering the important *Euclidean* vector spaces as a special case.

The mathematical *metric space* is a pair (D, d), where D is the *domain* of objects and d is the *distance function* able to compute distances between any pair of objects from D. It is assumed that the smaller the distance, the closer or more *similar* the objects are. For any distinct objects $x, y, z \in D$, the distance must satisfy the following properties of *reflexivity*, $d(x, x) = 0$, *strict positiveness*, $d(x, y) > 0$, *symmetry*, $d(x, y) = d(y, x)$, and *triangle inequality*, $d(x, y) \leq d(x, z) + d(z, y)$.

Though several sophisticated metric search structures have been proposed in literature, the two fundamental principles are called the *ball* and the *generalized hyperplane* partitioning [10]. In this article, we have concentrated on the recursive application of the second principle, which can be seen as the *Generalized Hyperplane Tree* (GHT). For the sake of clarity of the further discussion, we describe the main features of the GHT in the following.

Generalized Hyperplane Tree – GHT The GHT is a binary tree with metric objects kept in leaf nodes (buckets) of fixed capacity. The internal nodes contain two pointers to descendant nodes (sub-trees) represented by a pair of objects called the *pivots*. In the leaf nodes of the left sub-tree are objects closer to the first pivot, objects closer to the second pivot are in the leaf nodes of the right sub-tree.

The creation of the GHT structure starts with the bucket B_0. When the bucket B_0 is full, we create a new empty bucket, say B_1, and move some objects from B_0 to B_1. This idea of split is implemented by choosing a pair of pivots P_1 and P_2 ($P_1 \neq P_2$) from B_0 and by moving all the objects O, which are closer to P_2 than to P_1, into the bucket B_1. The pivots P_1 and P_2 are then placed into a new root node and the tree grows by one more level. In general, given an internal node i of the GHT structure with the pivots $P_1(i)$ and $P_2(i)$, the objects that meet Condition (1) are stored in the right sub-tree. Otherwise, they are found in the left sub-tree.

$$d(P_1(i), O) > d(P_2(i), O). \tag{1}$$

To **Insert** a new object O, we first traverse the GHT to find the correct storage bucket. In each inner node i, we test Condition (1): if it is true, we follow the right branch. Otherwise, we follow the left one. This is repeated until a leaf node is found and the object O is inserted – the split is applied if necessary.

In order to perform a similarity **Range Search** for the query object Q and the search radius r, we recursively traverse the GHT following the left child of each inner node if $d(P_1(i), Q) - r \leqslant d(P_2(i), Q) + r$ is satisfied and the right child if $d(P_1(i), Q) + r > d(P_2(i), Q) - r$ is true. Observe that, depending on the size of the radius r, both the conditions can be met simultaneously, which implies necessity of searching the left and the right subtrees at the same time.

3 GHT*

In general, the scalable and distributed data structure GHT* consists of a network of nodes that can insert, store, and retrieve objects using similarity queries. The nodes with all these functions are called *servers* and the nodes with only the insertion and query formulation functions are called *clients*. The GHT* architecture assumes that network nodes communicate through the *message passing* paradigm. For consistency reasons, each *request message* expects a confirmation by a proper *reply message*. Each node of the network has a unique *Network Node IDentifier* (NNID). Each server maintains data objects in a set of *buckets*. Within a server, 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), which is a structure similar to the GHT. The AST is used to actually determine the necessary (distributed) buckets when inserting and retrieving objects.

3.1 The Address Search Tree

Contrary to the GHT containing data objects in leaves, every leaf of the AST includes exactly one pointer to either a bucket (using BID) or a server (using NNID) holding the data. Specifically, NNIDs are used if the data are on a remote server. BIDs are used if the data are in a bucket on the local server. Since the clients do not maintain data buckets, their ASTs contain only the NNID pointers in leaf nodes.

A form of the AST structure is present in every network node, which naturally implies some replication. Due to the autonomous update policy, the AST structures in individual network nodes may not be identical – with respect to the complete tree view, some sub-trees may be missing. As we shall see in the next section, the GHT* provides a mechanism for updating the AST automatically during the insertion or search operations. Figure 1a illustrates the AST structure in a network of one client and two servers. The dashed arrows indicate the NNID pointers while the solid arrows represent the BID pointers.

Insert Insertion of an object starts in the node asking for insertion by traversing its AST from the root to a leaf using Condition (1). 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 server where the insertion continues recursively until an AST leaf with the BID pointer is reached.

In order to avoid repeated distance computations when searching the AST on the new server, a once-determined path specification in the original AST is also forwarded. The path sent to the server is encoded as a bit-string designated BPATH, where each node is represented by one bit – "0" for the left branch, and "1" for the right branch. Due to the construction of the GHT*, it is guaranteed that the forwarded path always exists on the target server.

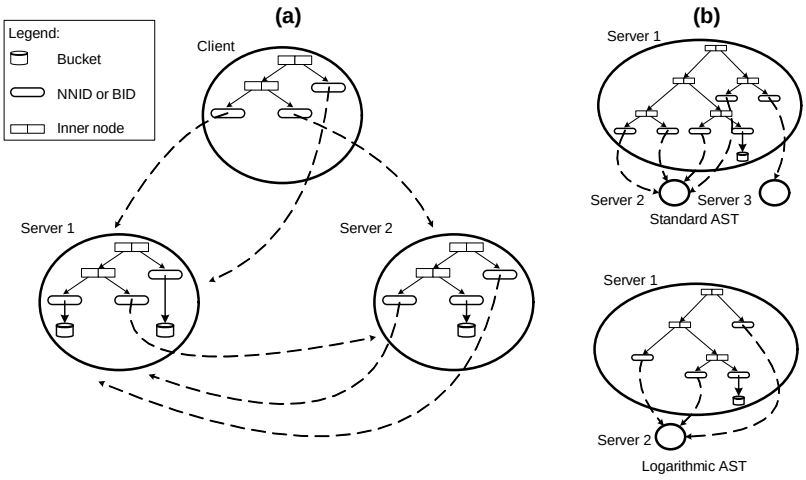

Fig. 1. AST and the GHT* network (a); example of logarithmic AST (b)

Range Search The range search also starts by traversing its local AST, but multiple paths can qualify. For all qualifying paths having a NNID pointer in their leaves, the query request with known BPATH is recursively forwarded to identified servers until a BID pointer occurs in every leaf. If multiple paths point to the same server, the request is sent only once but with multiple BPATH attachments. The range search condition is evaluated by the servers in every bucket determined by the BID pointers.

3.2 Image Adjustment

During insertion, servers split buckets without informing the other nodes of the network. Consequently, the network nodes need not have their ASTs up to date with respect to the data, but the advantage is that the network is not flooded with multiple messages at every split. The updates of the ASTs are thus postponed and actually done when respective insertion or range search operations are executed.

The inconsistency in the ASTs is recognized on a server that receives an operation request with corresponding BPATH from another client or server. In fact, if the BPATH derived from the AST of the current server is longer than the received BPATH, this indicates that the sending server (client) has an out-of-date version of the AST and must be updated. The current server easily determines a sub-tree that is missing on the sending server (client) because the root of this sub-tree is the last element of the received BPATH. Such a sub-tree is sent back to the server (client) through the *Image Adjustment Message*, IAM.

If multiple BPATHs are received by the current server, more sub-trees are sent back through one IAM (provided inconsistencies are found). Naturally, the IAM process can also involve more pairs of servers. This is a recursive procedure

which guarantees that, for an insert or a search operation, ASTs of every involved server (client) are correctly updated.

3.3 Logarithmic Replication Strategy

Using the described IAM mechanism, the GHT* structure maintains the ASTs practically equal on all servers. However, since every inner node of the AST contains two pivots, the number of replicated pivots increases linearly with the number of servers used. In order to reduce the replication, we have also implemented a much more economical strategy which achieves logarithmic replication on servers at the cost of moderately increased number of forwarded requests.

Inspired by the *lazy updates* strategy from [7], our logarithmic AST on a specific server stores only the nodes containing pointers to the local buckets (i.e. leaf nodes with BID pointers) and all their ancestors. So that the resulting AST is still a binary tree, all the sub-trees leading to leaf nodes with the NNID pointers are substituted by the leftmost leaf node of this sub-tree. The reason for choosing the leftmost leaf node is connected with our split strategy which always keeps the left node and adds the right one. Figure 1b illustrates this principle. In a way, the logarithmic AST can be seen as the minimum sub-tree of the fully updated AST. Furthermore, the image adjustment is only required when a split allocates a new bucket on a different server.

3.4 Storage Management

As we have already explained, the atomic storage unit of the GHT* is the bucket. The number of buckets and their capacity on a server are bounded by specified constant numbers, which can be different for different servers. To achieve scalability, the GHT* must be able to split buckets and allocate new storage and network resources.

Bucket Splitting The bucket splitting operation is performed in the following three steps: (1) a new bucket is allocated. If there is a capacity on the current server, the bucket is activated there. Otherwise, the bucket is allocated either on another existing server with free capacity or a new server is used; (2) a pair of pivots is chosen from the objects of the overflowing bucket. (3) objects from the overflowing bucket that are closer to the second pivot than to the first one are moved to the new bucket.

New Server Allocation In our prototype implementation, we use a pool of available servers which is known to every active server. We do not use a centralized registering service. Instead, we exploit the *broadcast messaging* to notify the active servers. When a new network node becomes available, the following actions occur: (1) the new node with its NNID sends a broadcast message saying "I am here" (this message is received by each active server in the network); (2) the receiving servers add the announced NNID to theirs local pool of available

servers. When an additional server is required, the active server picks up one item from the pool of available servers. An activation message is sent to the chosen server. With another broadcast message, the chosen server announces: "I am being used now" so that other active servers can remove its NNID from their pools of available servers. The chosen server initializes its own pool of available servers, creates a copy of the AST, and sends to the caller the "Ready to serve" reply message.

Choosing Pivots We use an incremental pivot selection algorithm from [1], which tries to find a pair of distant objects. At the beginning, the first two objects inserted into an empty bucket become the candidates for pivots. Then, we compute distances to the current candidates for every additionally inserted object. If at least one of these distances is greater than the distance between the current candidates, the new object replaces one of the candidates so that the distance between the new pair of candidates grows. When the bucket overflows, the candidates become pivots and the split is executed.

4 Performance Evaluation

In this section, we present results of performance experiments that assess different aspects of our GHT* prototype implemented in Java. The system was executed on a cluster of 100 Linux workstations connected with a 100Mbps network using the standard TCP/IP protocol.

We conducted our experiments on 45-dimensional vectors (VEC) of color image features with the L_2 (Euclidian) metric distance function and on sentences of the Czech national corpus with the *edit distance* as the metric (TXT). See [4] for more details about these data sets.

In the following, we designate the maximal number of buckets per server as n_b, and the maximal number of objects in a bucket as b_s. Due to the space limitations, we only report results concerning the range search performance and the parallel aspects of query execution.

4.1 Range Search Performance

For the range search, we have analyzed the performance with respect to different sizes of query radii. We have measured the search costs in terms of: (1) the number of servers involved in the execution of a query, (2) the number of buckets accessed, (3) the number of distance computations in the AST and in all the buckets accessed. We have not used the query execution time as the relevant criterion because we could not ensure the same environment for all participating workstations.

Experiments were executed on the GHT* structure with configuration $n_b = 10, b_s = 1,000$, which was filled with 100,000 objects either from the VEC or the TXT data-set. The average load of buckets was about 50 percent of their capacity. We have used the logarithmic replication strategy. Each point of every

graph was obtained by averaging results of 50 range queries with the same radius and a different (randomly chosen) query object.

In the first experiment, we have focused on relationships between query radius sizes and the number of buckets (servers) accessed. Figure 2 reports the results of these experiments together with the number of objects retrieved. If the radius increases, the number of accessed servers grows practically linearly, but the number of accessed buckets grows a bit faster. However, the number of retrieved objects satisfying the query grows even exponentially. This is in accordance with the behavior of centralized metric indexes such as the M-tree or the D-index on the global (not distributed) scale. The main advantages of the GHT* structure are demonstrated in Section 4.2 when the parallel execution costs are considered.

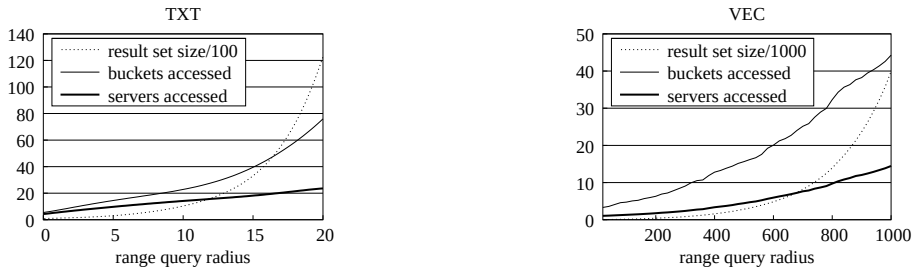

Fig. 2. Average number of buckets, servers, and retrieved objects (divided by 100 and 1,000) as a function of the radius.

Another important aspect of a distributed structure is the number of messages exchanged during search operations. Figure 3 presents the average number of messages sent by a client and the number of forwarded messages initiated on servers during a query evaluation. In fact, if we sum the number of messages sent by a client and by the servers we get the total number of servers involved in a query execution, see Figure 2 for verification. Observe that even with the logarithmic replication strategy the number of forwardings is reasonably low.

In Figure 4 we show the average number of distance computations performed by a client and the necessary servers during a query execution. We only report distance computations needed during the traversal of the AST (two distance computations must be evaluated per each inner node traversed). We do not show the number of distance computations inside the accessed buckets, because they depend on the bucket implementation strategy. In our current implementation, the buckets are organized in a dynamic list so the number of distance computations per bucket is simply given by the number of objects stored in the bucket. We are planning to use more sophisticated strategies in the future.

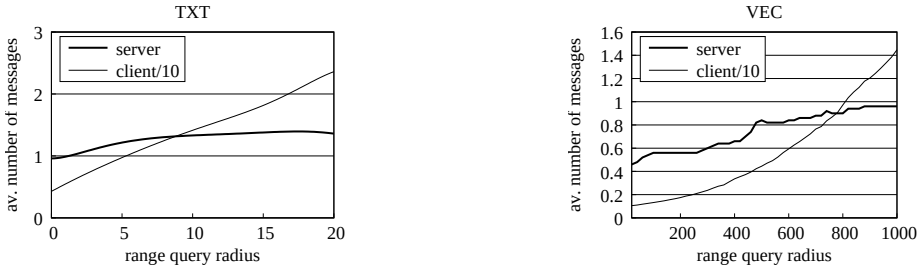

Fig. 3. Average number of messages sent by a client and server as a function of the radius.

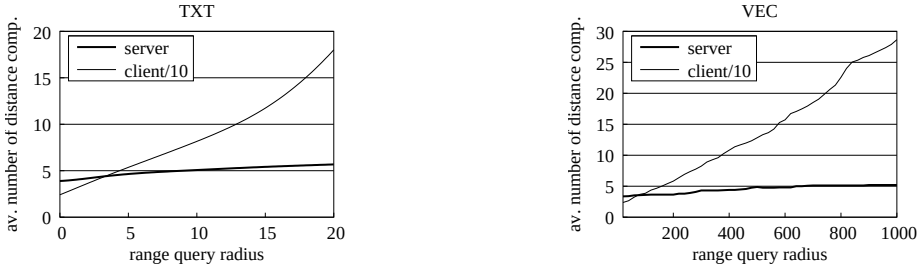

Fig. 4. Average number of distance computations in the AST for clients (divided by 10) and servers as a function of the radius.

4.2 Parallel Performance Scalability

The most important advantage of the GHT* with respect to the single-site access structures concerns its scalability through parallelism. As the size of a data-set grows, new server nodes are plugged in and their storage as well as the computing capacity is exploited. By assuming the same number of buckets of equal capacity on each server, the number of used servers grows linearly with the data-set size. This was also experimentally confirmed.

We focused on experimental studies of the intrinsic aspects of the *intraquery* and *interquery* parallelism to show how they actually contribute to the GHT* scalability. First, we consider the scalability from the perspective of a growing data-set and a fixed set of queries, i.e. the *data volume scalability*. Then, we consider a constant data-set and study how the structure scales up with the growing search radii, i.e. the *query volume scalability*.

We quantify the intraquery parallelism as the parallel response time of a range query. It is determined as the maximum of the costs incurred on servers involved in the query evaluation plus the serial search costs of the ASTs. For the evaluation purposes, we use the number of distance computations (both in the ASTs and in all the accessed buckets) as the computational costs of a query execution. In our experiments, we have neglected the communication time

because the distance computations were more time consuming than sending a message to a network node.

The interquery parallelism is more difficult to quantify. To simplify the analysis, we characterize the interquery parallelism as the ratio of the number of servers involved in a range query to the total number of servers – the lower the ratio, the higher the chances for other queries to be executed in parallel. Under the above assumptions, the intraquery parallelism is proportional to the response time of a query and the interquery parallelism represents the relative utilization of computing resources.

To evaluate the data volume scalability, we used the GHT* configuration of $n_b = 10$ $b_s = 1,000$. The graphs in Figure 5 represent the parallel search time for two different query radii as a function of the data-set size. The results are available separately for the vector (VEC) and the sentence (TXT) data-sets. By analogy, Figure 6 shows the relative utilization of servers for two types of queries and growing data-sets. The results are again averages of costs measured for 50 range queries of constant radius and different (randomly chosen) query objects.

Our experiments show that the intraquery parallelism remains very stable and the parallel search time is practically constant, i.e. independent of the data-set size. Though the number of distance computations needed for the AST traversal grows with the size of the data-set, this contribution is not visible. The reason is that the AST grow is logarithmic, while the server expansion is linear.

At the same time, the ratio characterizing the interquery parallelism in Figure 6 is even decreasing as the data-set grows in size. This means that the number of servers needed to execute the query grows much more slowly than the number of incoming active servers, thus the percentage of servers used to evaluate the query is on a down curve.

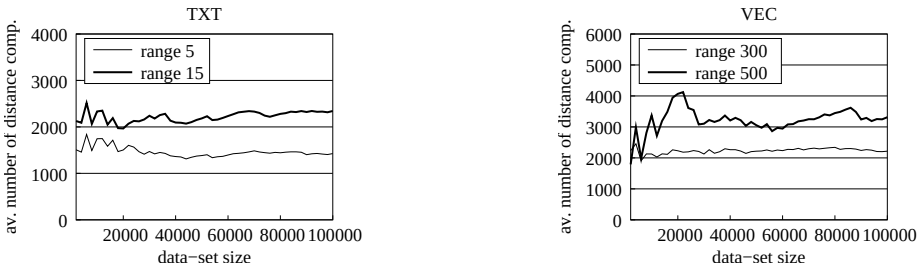

Fig. 5. Average intraquery parallelism as a function of the data-set size for two different query radii.

To evaluate the query volume scalability, we have fixed the data-set size at 100,000 objects and used queries with growing radii. Figure 7 shows the relationships between the size of a range query radius and the number of distance computations – the upper curve represents the total costs (in distance computations) to solve the query. Observe that this upper curve is closely related with

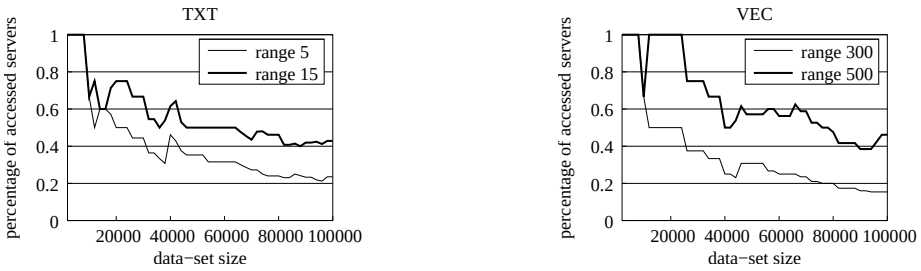

Fig. 6. Average interquery parallelism as a function of the data-set size for two different query radii.

the number of accessed buckets shown in Figure 2. The parallel cost curve represents the intraquery parallelism for the query volume scalability. Though the total computational cost of the query grows quickly as the size of the radius increases, the parallel cost remains stable after some starting phase, i.e. when retrieving very small sub-sets.

As intuitively clear, the level of the interquery parallelism for the increasing radius is actually decreasing. In fact, the larger the query radius the more servers are used from a constant set of active servers. Such property is demonstrated in Figure 2 as the linear dependence of the number of accessed servers on the radii size.

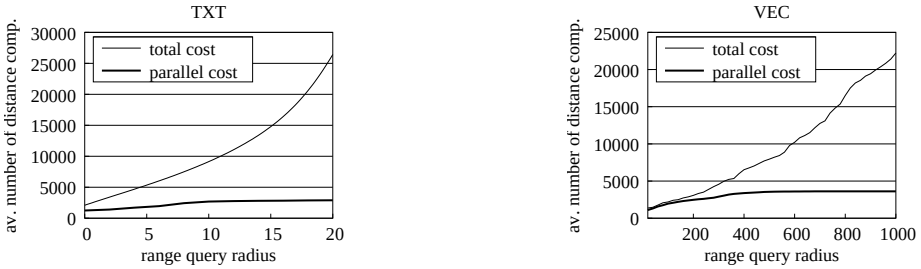

Fig. 7. Average intraquery parallelism as a function of the radius.

5 Conclusions and Future Work

To the best of our knowledge, the problem of distributed index structures for the similarity search in metric data has not been studied yet. Our structure is thus the first attempt at combining properties of scalable and distributed data structures and the principles of metric space indexing.

The GHT* structure stores and retrieves data from domains of arbitrary metric spaces and satisfies all the necessary conditions of SDDSs. It is scalable in that it distributes the structure over more and more independent servers. The parallel search time becomes practically constant for arbitrary data volume and the larger the data-set the higher the potential for the interquery parallelism. It has no hot spots – all clients and servers use as precise addressing scheme as possible and they all incrementally learn from misaddressing. Finally, updates are performed locally and a node splitting never requires sending multiple messages to many clients or servers.

The main contributions of our paper can be summarized as follows: (1) we have defined a metric scalable and distributed similarity search structure; (2) we have experimentally validated its functionality on real-life data-sets. Our future work will concentrate on strategies for updates, pre-splitting policies, and more sophisticated strategies for organizing buckets. We will also develop search algorithms for the Nearest Neighbor queries. 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. B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. In *Proc. of the XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 33–40, 2001.
2. E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
3. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of 23rd International Conference on Very Large Data Bases (VLDB)*, pages 426–435, 1997.
4. V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–13, 2003.
5. C. Gennaro, P. Savino, and P. Zezula. Similarity search in metric databases through hashing. In *Proc. of the 3rd International Workshop on Multimedia Information Retrieval*, pages 1–5, October 2001.
6. G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.
7. T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, volume 22(2), pages 337–346, 1993.
8. W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - a scalable, distributed data structure. *TODS*, 21(4):480–525, 1996.
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of ACM SIGCOMM 2001*, pages 161–172, 2001.
10. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *IPL: Information Processing Letters*, 40:175–179, 1991.