Scalable and Distributed Similarity Search in Metric Spaces

Michal Batko, Claudio Gennaro and Pavel Zezula

Abstract

In this paper we propose a new access structure, called GHT*, based on generalized hyperplane tree (GHT) and distributed dynamic hashing (DDH) techniques. GHT* is a distributed structure which allows to perform range search in a metric space according to a distance function d. The structure does not require a central directory and it is able to gracefully scale through splits of one bucket at a time.

1 Introduction

The main function of computers is processing information and we now use computers almost everywhere. The information can be of any type and it can have any structure. Multimedia is one type of often processed data, however we cannot use common searching methods for them. Fortunately another approach is available: the metric spaces.

Multimedia data are usually huge files and the distance computations (necessary to search in metric space) are time consuming operations. The scalability and performance of a single site multimedia indexing system is thus limited. But networks are very fast today and computers usually have plenty of memory, so the use of the distributed architectures would be a great enhancement.

Some methods for effective indexing in metric space have been proposed in the past and distributed scalable structures are available too. However, present metric space indexing techniques are mainly focused on single site algorithms and almost every distributed structure is based on linear order of primary key. The aim of this research is to introduce a new structure, which combines those two approaches.

2 Previous work

2.1 Scalable and Distributed Data Structures

The first scalable and distributed data structure (SDDS) proposed was LH* [3], which uses linear hashing to split file among network sites. In this article were also defined the most important properties that every SDDS should satisfy:

scalability – data expand to new servers gracefully, and only when servers already used are efficiently loaded.

no hot-spot – there is no master site that must be accessed for resolving the address of the searched objects, e.g., there is no centralized directory.

independence – the file access and maintenance primitives, e.g., search, insertion, split, etc., never require atomic updates to multiple clients.

Another hash-based algorithm is DDH [2], based on dynamic hashing. The advantage of DDH over LH* is that it can immediately split any bucket (LH* according to LH can split only the bucket with token).

2.2 Metric space searching methods

Metric space is pair compound of a set of objects (A) and a so-called metric function (d), which computes "distance" between any pair of objects from the set (i.e. the smaller distance, the closer or more similar are the objects). The distance is a positive real number and the function d has following properties:

$\forall x \in \mathcal{A} : d(x,x) = 0$	*reflexivity*
$\forall x,y \in \mathcal{A} : d(x,y) > 0$	*strict positiveness*
$\forall x,y \in \mathcal{A} : d(x,y) = d(y,x)$	*symmetry*
$\forall x,y,z \in \mathcal{A} : d(x,y) \leq d(x,z) + d(z,y)$	*triangle inequality*

We usually look for similar objects when searching in metric space, i.e. we look for the *nearest neighbor* object. Another type of query is the *range* or *proximity* search – with the goal of retrieving all objects which have distance from query object at most the specified range.

In this research we have exploited the Generalized Hyperplane Tree approach presented in [4], because it can be easily combined with locally autonomous splitting of DDH.

3 GHT* structure

Our scalable distributed algorithm for indexing metric space data is based on DDH (the distributed part) and GHT (the metric space indexing). Objects are stored using the metric function d into distributed buckets. The structure satisfies the properties of SDDS and it allows efficient similarity queries on stored data.

3.1 Generalized Hyperplane Tree – GHT

GHT is a binary tree that allows to store metric space objects and quickly perform similarity search on them. The objects are kept in buckets of fixed capacity, that are pointed from leaf nodes of the tree. Inner tree nodes contain pointers to a selected pair of objects (pivots). Elements closer to the first pivot go into left subtree and the objects nearest to the second one, into the right subtree.

We start by building a tree with only one root node pointing to a bucket B_0. When the bucket B_0 is full we must split it: we create a new empty bucket B_1 and move some objects (exactly one half if possible) into it to obtain space in B_0 (see Figure 1a). We can do that by choosing a pair of pivots P_1 and P_2 ($P_1 \neq P_2$) from B_0 and moving into bucket B_1 all objects O that satisfy the following condition.

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

Pivots P_1 and P_2 will be associated with a new root node and thus the tree will grow one level up. This split algorithm can be applied on any node and it is an autonomous operation (no other tree nodes need to be modified).

When we **Insert** a new object O we first traverse GHT to find the correct bucket. At each inner node we test the condition (1): if it is true we take the right branch; otherwise the left branch is taken. This is done until a leaf node is found. Then we insert O into the pointed bucket and we split it, if necessary.

In order to perform a **Range Search** with query object Q and radius r, we recursively traverse GHT following the left child of each inner node if condition (2) is satisfied and right child if condition (3) is true.

$$d(P1, Q) - r \leqslant d(P2, Q) + r \tag{2}$$

$$d(P1, Q) + r > d(P2, Q) - r \tag{3}$$

Note that, since both the above conditions can be simultaneously satisfied (i.e., both the left and right subtree can be traversed) eventually more than one bucket can contain qualifying objects.

The **Nearest Neighbor** search is not described in this paper, due to space limitations, however it uses range search as was specified in [5].

3.2 Architecture of GHT*

The distributed environment comprises network nodes which can be divided into two groups. *Clients*, which insert new objects and perform similarity queries on stored data. *Servers*, which hold the objects and respond to queries.

A server is uniquely identified by its *SID* (Server ID). Every server holds a bunch of buckets, which are used to physically store the objects. Each bucket has an unique identifier per server called *BID*. Each client and server have a GHT like structure that we will call *address search tree* (*AST*). It is used to actually address correct server for storing or retrieving data.

3.3 Address search tree

AST is a slightly modified GHT (defined in section 3.1). First, we have a different bucket association and the buckets are distributed among the servers. Moreover, the tree leaf nodes have either a *SID*, if the bucket is on another server, or a *BID* if the bucket is on the current server. Second, since it may

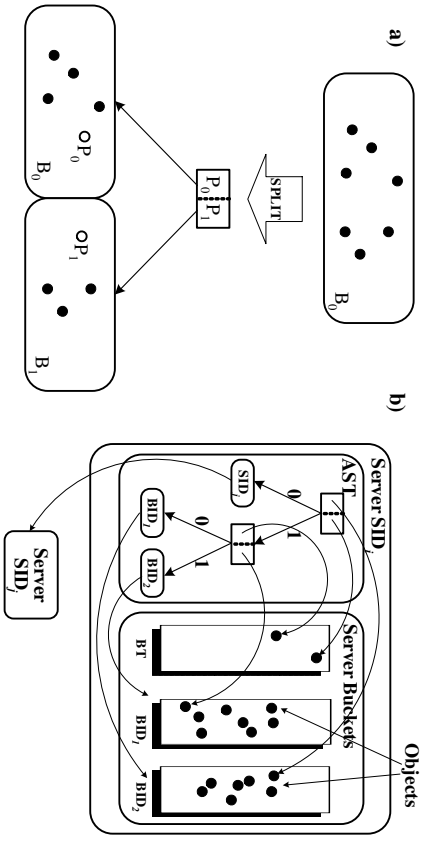

Figure 1: Split of the bucket and modification of the tree (a), address tree with pivots stored in server buckets (b).

happen that a distance computation needs a pivot stored in another server, we need to locally replicate this pivot onto a special bucket called Tree Bucket (TB). Actually only the features essential for metric function computations are copied. Client's AST has only SID pointers in leaf nodes, because it has no data buckets. Figure 1b illustrates the server structure.

Each server (or client) has its own "view" of the distributed structure represented by its AST. The view may be incomplete (in a way that some branches of the tree are missing), because any server can split a bucket any time. This may cause clients to send requests to incorrect servers, but this is not a problem. Every server has a mechanism, which allows to forward a request to the correct server and to inform a client about necessary updates to its AST, as explained in section 3.5.

When a client has to **insert** a new object, it first consults its local AST, and it finds a leaf node containing a SID, where request has to be sent. The receiving server will traverse its own AST. If again the leaf node contains a SID pointer, the request has to be forwarded to that server. Otherwise, a BID pointer is found and the server has to store the object into the bucket BID, which must be split when an overflow occurs.

In order to perform a **range search** we use a similar method. The only difference is that in general we obtain a set of SIDs, this because, as explained, range query may follow both the subtrees of a node. In this case, the query message is sent to all servers from the set.

In order to save time, avoiding some distance computations, the message sender creates a binary traversal path ($BPATH$), by writing 0 if it takes a left branch and 1 for a right branch and sent it to receiving server. This representation is then used to quickly traverse AST (that was already computed on client) at receiving server.

3.4 Bucket splitting

Since the server bucket capacity is limited, it must be split when an overflow occurs. Suppose a bucket in the server with $SID = j$ overflows, the splitting is performed in the three following steps. First, we create a new bucket on either the server j or another server. In general, if the server j have enough space for the new bucket it must preferred. Otherwise, the bucket is created on a different less loaded server. Second, for the created bucket we choose a pair of pivots and we update the AST. Finally, we move some objects from current bucket to the new one according to the new pivots.

The pivot decision algorithm is very important, because it can directly affect the performance of the structure. In addition, the selection of pivots is time-consuming operation – the algorithm usually must perform many distance computations to choose good pivots. This could lead to latencies in distributed environment, so we decided to use incremental pivot decision algorithm. It is based on efficiency criterion that was proposed in [1] and it allows preselecting two objects for each bucket that are used as pivots when bucket splits. All necessary computations are performed on the insertion of new object. If the bucket is empty, the first two inserted objects are selected. Then for every inserted object, we compute distance to both preselected pivots and use efficiency criterion to decide, if the new object should replace

one of the preselected pivots. When split occurs, we simply use the current two preselected objects as pivots.

3.5 Messages and forwarding

Each client and server is able to send a message to another server using its *SID*. This is done through some underlying framework incorporated in every server or client. The framework also allows to create new servers and to assign them unique *SID*s.

When a client request occurs, it first consults its own AST and discovers the proper *SID* and the representation of the search ($BPATH$). Then the client sends a request message containing the necessary objects to the server *SID*.

The server which receives the message, first decides if the current request was not already processed and committed. In this way the same request is not processed twice – this is important mainly for range queries. The server then quickly traverses its own AST using *BPATH*. If a leaf node is reached, the target bucket is processed because it is on the current server. Otherwise the server performs additional search from this node extending $BPATH$ until a leaf node is found. Two possibilities may occur – the leaf node contains a pointer to a local bucket or to another server. In both cases, by processing either local or remote bucket, the results and the update messages are sent back to client.

Moreover, if the request is a range query, we can have a set of nodes and more *BPATH*s instead of one. The search then continues on every branch specified by the $BPATH$ and all matching leaf nodes are forwarded or processed.

3.6 Update message

The update message mentioned above is a special message, which allows clients and servers to update the "view" represented by AST. It is usually called IAM (Image Adjustment Message). The update message is sent only if forwarding or additional search is performed at server side, i.e. the client has an improper AST.

The update message contains a subtree of the AST node marked from client as the target (and reconstructed at server from $BPATH$). Server sends also all necessary pivot objects. The client stores pivot objects into its TB bucket and updates pointers accordingly to the changed address subtree.

4 Experimental results

In this section we present a preliminary experimental evaluation based on a Java implementation of the proposed GHT* structure. For the evaluation and testing purposes, we have used a data–set of 1000 2D vectors and the Euclidean distance function has been used. The vectors are uniformly distributed in the square of size $[-1000, 1000] \times [-1000, 1000]$.

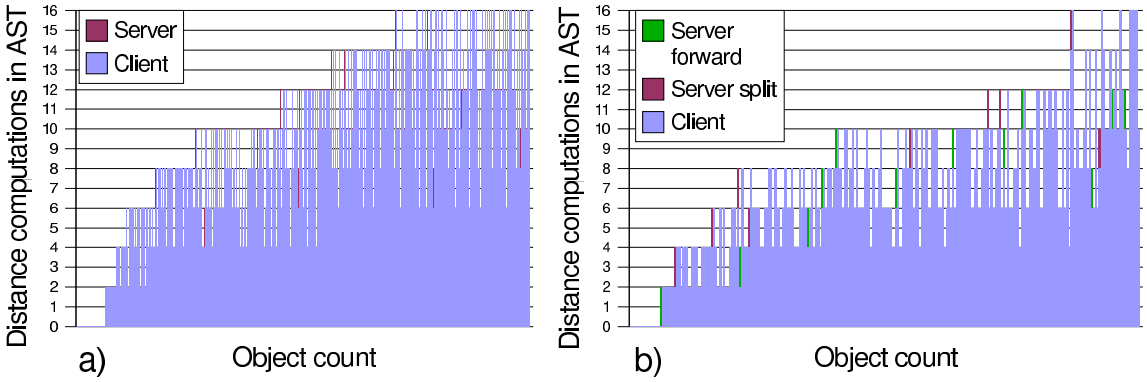

Figure 2: Distance computations per insert for single inserting client (a), Distance computations per insert for one of four simultaneously inserting clients (b).

We have chosen the following architecture of GHT* to store the objects:

- new servers are added on demand
- each server have 5 buckets at maximum
- bucket maximal capacity is of 64 objects

The first observation we have made is about the computations of the distance function. Those computations are first performed at client side and the results then sent to the distributed environment, where additional distance computations may be necessary (but they are never repeated). Another important factor is the message count and forwarding. These operations are important for the maintenance of the "correct" view of the client and servers, which are constantly improved through the updating messages. Note that, we have focused only on the distance computations that are necessary to identify the correct bucket and we have not taken in consideration the computations that are needed to actually retrieve exact objects in the bucket. This is because a specific access structure for improving the search can be used in the bucket (instead of the simple sequential scan).

4.1 Inserting

In this set of experiments, we have analyzed the number of distance computations during the insertion phase. We have inserted 1000 random objects using the same client several times. The number of distance computations at server was two at maximum (during the split) and almost a distance computation is performed at the client site for each distance computation in the AST (see Figure 2a). Figure 2b shows the case where three other clients were simultaneously inserting vectors into the structure (the number of inserted objects was reduced to 250 per client). Note that, the number of distance computations in this case is increased. This is because the clients sometimes perform an addressing error and thus servers must forward the message (distance computations due to misaddressing are marked as "Server forward").

We have also evaluated the global properties of GHT* on this simple 2D vector space. We have slightly modified the settings of GHT* structure:

- the number of inserted objects was increased to 10000
- the maximal number of buckets per server was increased to 10
- the bucket maximal capacity was increased to 250 objects

The results are shown in table 1. We have achieved nearly 65% of overall structure load, which is a good result considering the distribution of the objects of the metric space used. However, in general these results strongly depend on the type of metric space used, on the actual distribution of the objects in the space, and on the pivot technique used.

	min	max	avg
Occupied buckets	56	68	62.4
Occupied servers	7	9	8.07
Overall bucket load	58.82	71.43	64.31
Maximal AST depth	16	26	20.4

Table 1: Global properties of GHT* after inserting 1000 vectors (averaging on 30 separate experiments)

A peculiarity property of GHT* is the replication, in fact the pivots are copied to the servers into TB bucket. Each level of the AST contains two pivots, therefore the number of replicated objects is twice the number of inner nodes of AST multiplied by the number of servers. Our experiments with 2D vectors exhibited a replication factor from 3.92% (in the best case) to 5.85% (in the worst case). However, the replication factor strongly depends on the initial setting of the servers maximal capacity.

4.2 Similarity search

The similarity search is the main feature of GHT*. We have executed range searches on the entire data–set stored in GHT*. We made two different set of tests: one with a very small radius of 50, which returns about 3 object in average, and another one with a bigger radius (350) which returns approximately 100 objects in average. We have executed both the types of query with 20 random generated query objects on a fresh client (which has no knowledge of the structure, and hence has no AST at the start time). The number of distance computations (on the client and on the servers) and number of messages (sent by the client and by the servers) are shown in Figures 3 and 4.

[h]

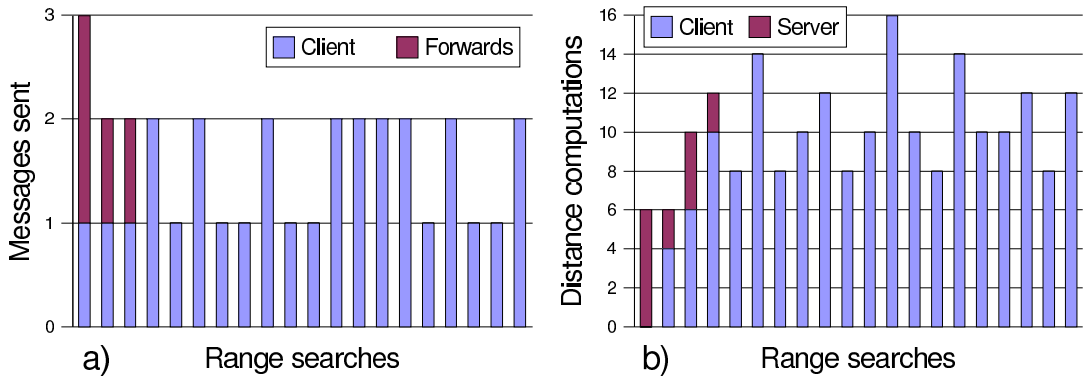

Figure 3: Distance computations per range query with radius 50 units (a), number of messages sent per range query with radius 50 (b).

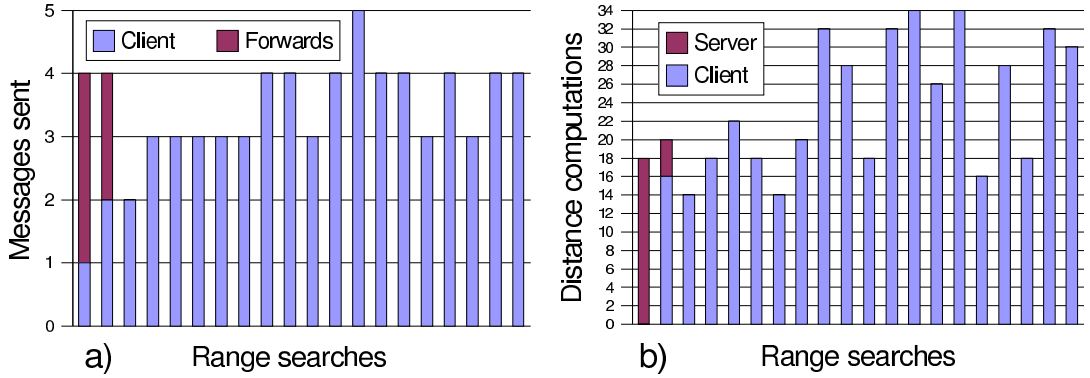

Figure 4: Distance computations per range query with radius 350 units (a), number of messages sent per range query with radius 350 (b).

5 Conclusion

As far as we know there has not been proposed metric space indexing structure, which uses distributed environment. Our structure is a first try to combine existing techniques for scalable and distributed data structures and metric space indexing.

GHT* structure allows storing data sets from any metric space and has all necessary attributes of SDDS. It is scalable – every server can anytime perform an autonomous split gracefully distributing the structure over more servers. It has no hot spot – all clients and servers use as precise addressing as possible and they all learn from misaddressing. Finally, updates are performed locally and splitting never requires sending multiple messages to many clients or servers.

Even thought our experiments of GHT* are still in preliminary phase, we can conclude that the structure matches all the desired properties and thus it can perform metric space similarity searches in distributed environment. The main advantage as opposed to single site metric indexing schemes is that the clients only perform few distance computations to address the correct server and the server then

searches through its buckets to find the qualified objects. Other important feature is the ability to divide metric space objects between servers and thus enhance the search by parallel processing on more servers.

References

[1] Benjamin Bustos, Gonzalo Navarro, and Edgar 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] Robert Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, volume 730, pages 101–114, Chicago, 1993.

[3] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH* - a scalable, distributed data structure. *TODS*, 21(4):480–525, 1996.

[4] Uhlmann. Satisfying general proximity / similarity queries with metric trees. *IPL: Information Processing Letters*, 40, 1991.

[5] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 421–430. Morgan Kaufmann, 2001.