M-Grid: Similarity Searching in Grids

Michal Batko
Faculty of Informatics
Masaryk University
Botanická 68a
Brno, Czech Republic
xbatko@fi.muni.cz

Vlastislav Dohnal
Faculty of Informatics
Masaryk University
Botanická 68a
Brno, Czech Republic
dohnal@fi.muni.cz

Pavel Zezula
Faculty of Informatics
Masaryk University
Botanická 68a
Brno, Czech Republic
zezula@fi.muni.cz

ABSTRACT

The problem of similarity searching is nowadays attracting a lot of attention, because upcoming applications process complex data and the traditional exact match searching is not sufficient. There are efficient solutions, but they are tailored for the needs of specific data domains. General solutions, based on the metric space abstraction, are extensible, but they are designed to operate on a single computer only. Therefore, their scalability is limited and they cannot adapt to different performance requirements. In this paper, we propose a distributed access structure which is fully dynamic and exploits a Grid infrastructure. We study properties of this structure in numerous experiments. Besides, the performance tuning is analyzed with respect to user-specific requirements which include the maximum response time and the number of queries executed concurrently.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; H.2.4 [**Database Management - Systems**]: Query processing, Multimedia databases; H.3.4 [**Information Storage and Retrieval - Systems and Software**]: Distributed systems

General Terms

Algorithms, Performance

Keywords

similarity searching, metric space, M-Grid, D-index, performance analysis

1. INTRODUCTION

In traditional databases, the problem of searching for objects that exactly meet specific criteria is very well defined and there are plenty of efficient solutions. However, as new digital-age applications have emerged lately, this concept fails to meet the new requirements. For example, searching for time series with similar development or groups of customers with common buying patterns in respective data collections cannot use the traditional approach. Instead, a new searching paradigm is introduced: the similarity searching. In this respect, not only objects that perfectly satisfy a posed query in all aspects (which probably do not exist anyway) but also objects that meet the criteria closely enough are retrieved from the database.

Methods for efficient evaluation of similarity queries over a data collection exist (see recent surveys [21, 7]), but they are usually designed for centralized systems where all data are stored and the processing is done on one computer. However, the problem of these solutions, as shown in [10], is their scalability. Since the response time of these structures grows linearly with the size of the indexed dataset, sooner or later, the response time of the system becomes unacceptable for online processing. Moreover, the centralized systems cannot easily adapt to performance requirements such as the maximum query processing time or the number of users accessing the data simultaneously.

Therefore, attempts to use distributed infrastructures to parallelize computations have arisen recently. Since the distributed environment is not restricted to one computer only and can employ virtually unlimited resources, the scalability challenge is shifted to another level. Besides, the performance can be tuned, since the load of individual nodes can be divided among additional nodes or the overloaded ones can be replicated.

A very recent paper [3] studies the scalability of similarity searching in a peer-to-peer computing environment and results show that providing enough resources, the response time is practically unaffected by the volume of indexed data. However, the peer-to-peer paradigm expects the participating computer nodes (which provide the necessary pool of resources) to be independent and equal in functionality. Moreover, the computers are usually not dedicated for a specific application. Thus, we need a more reliable distributed environment where the properties can be guaranteed and the performance tuned as necessary. From this point of view, a Grid infrastructure [13] seems to satisfy our requirements. The parallel resources, both the computational and storage, can be requested on demand as necessary and once assigned they are dedicated to the application.

In this paper, we provide a similarity searching structure designed for Grid infrastructures. Our technique is based on the D-index [11] structure, which provides a hashing-like centralized similarity index. Since an addressing schema

P2PIR'06, November 11, 2006, Arlington, Virginia, USA.

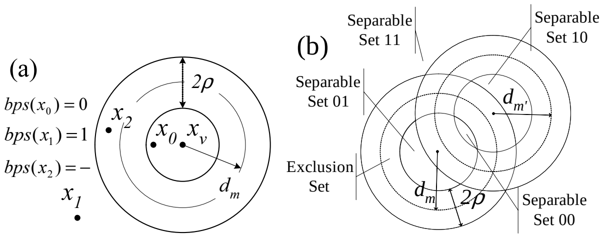

Figure 1: The *bps* function (a) and the combination of two *bps* functions (b).

of this method is static and must be provided by the user in advance, we discuss three dynamic variants allowing the addressing to adjust automatically as new data items are stored. Next, we study its performance properties by defining grouping and replication techniques. They help the structure distribute the computational load equally. Finally, we deal with the issue of performance tuning.

2. SIMILARITY SEARCHING

A convenient way to assess similarity between two objects is to apply metric functions to decide the closeness of objects as a distance, that is the objects' dissimilarity. A *metric space* $\mathcal{M} = (\mathcal{D}, d)$ is defined by a domain of objects (elements, points) $\mathcal{D}$ and a total (distance) function d – a non-negative ($d(x, y) \geq 0$ with $d(x, y) = 0$ iff $x = y$) and symmetric ($d(x, y) = d(y, x)$) function that satisfies the triangle inequality ($d(x, y) \leq d(x, z) + d(z, y), \forall x, y, z \in \mathcal{D}$).

In general, the problem of indexing in metric spaces can be defined as follows: *given a set $X \subseteq \mathcal{D}$ in the metric space $\mathcal{M}$, preprocess or structure the elements of X so that similarity queries can be answered efficiently.* For a query object $q \in \mathcal{D}$, two fundamental similarity queries can be defined. A *range query* retrieves $\mathcal{R}(q, r)$ all elements within distance r to q, that is, the set $\{x \in X, d(q, x) \leq r\}$. A *nearest neighbor* query retrieves $NN_k(q)$ the k closest elements to q, that is a set $R \subseteq X$ such that $|R| = k$ and $\forall x \in R$, $y \in X - R, d(q, x) \leq d(q, y)$.

Due to space constraints of this article, we consider only similarity range search operations here. In the following, we briefly outline the D-index, on ideas of which we build our approach.

2.1 D-index

The Distance Index (D-index) is designed as a centralized solution for similarity searching. It consists of two parts: an *addressing structure* and a *storage*. The addressing structure forms a multi-level structure of ρ-split functions. It addresses a set of *search-separable* buckets which are organized in levels and which comprise the storage.

The D-index supports easy insertion and bounded search costs because at most one bucket needs to be accessed at each level for range queries up to a predefined value of search radius ρ. At the same time, the applied *pivot-based filtering* significantly reduces the number of distance computations in accessed buckets. In the following, we provide a brief overview of the D-index, more details can be found in [14] and the full specification, as well as performance evaluations, are available in [11].

The partitioning principles of the D-index are based on a multiple definition of a mapping function, called the ρ-split function. Figure 1a shows a possible implementation of a ρ-split function, called the *ball partitioning split* (*bps*), originally proposed in [20]. This function uses one pivot x_v and the *medium distance* d_m to partition a data set into three subsets. The result of the following *bps* function gives a unique identification of the set to which the object x belongs:

$$bps(x) = \begin{cases} 0 & \text{if } d(x, x_v) \leq d_m - \rho \\ 1 & \text{if } d(x, x_v) > d_m + \rho \\ - & \text{otherwise} \end{cases}$$

The subset of objects characterized by the symbol '$-$' is called the *exclusion set*, while the subsets of objects characterized by the symbols 0 and 1 are the *separable sets*, because any range query with radius not larger than ρ cannot find qualifying objects in both the subsets.

More separable sets can be obtained as a combination of several *bps* functions, where the resulting exclusion set is the union of the exclusion sets of the original ρ-split functions. Furthermore, new separable sets are obtained as the intersection of all possible pairs of the separable sets of original functions. Figure 1b gives an illustration of this idea for the case of two ρ-split functions. The separable sets and the exclusion set form separable buckets and an exclusion bucket of one level of the D-index structure, respectively.

Naturally, the more separable buckets we have, the larger the exclusion bucket is. For a large exclusion bucket, the D-index allows an additional level of splitting by applying a new set of ρ-split functions on the exclusion bucket. The exclusion bucket of the last level forms the exclusion bucket of the whole structure. The ρ-split functions of individual levels should be different but they must use the same ρ. Moreover, by using a different number of ρ-split functions (generally decreasing with the level), the D-index's addressing structure can produce a different number of buckets at individual levels.

2.2 Search Algorithms

Before giving a sketch of search algorithm, we assume that we have an h-level distance searching index $D\text{–}index(h, s_1^{m_1,\rho}, \ldots, s_h^{m_h,\rho})$ which is determined by h independent ρ-split functions $s_i^{m_i,\rho}$, where m_i specifies the number of *bps* functions combined together.

Given a range query $\mathcal{Q} = \mathcal{R}(q, r)$, we define a simple search algorithm. This algorithm, however, evaluates only limited queries with $r \leq \rho$.

Algorithm 1. *Search*

```
for $i = 1$ to $h$
    return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,0}(q)\rangle}$;
end for
return all objects $x$ such that $x \in \mathcal{Q} \cap E_h$;
```

During the elaboration of the algorithm we manipulate the value of parameter ρ of ρ-split functions. When we use $\rho = 0$ the function $\langle s_i^{m_i,0}(q)\rangle$ always returns an identification id of a separable set and cannot evaluate to '$-$', i.e., the exclusion set. Consequently, one separable bucket $B_{i,id}$ on each level i is determined. Finally, the algorithm also accesses the exclusion bucket E_h of the whole structure. The algorithm requires $h+1$ bucket accesses, which forms the upper bound on the search. Such a behavior is correct since due to the mathematical properties of the ρ-split functions, precisely defined in [11], an arbitrary object belonging to a separable bucket is at distance at least 2ρ from any object of another separable bucket of the same level.

With additional computational effort, a query with $r > \rho$ can also be executed using a generalized range search algorithm [11]. By manipulating the parameter ρ of a ρ-split function again, this algorithm detects situations when the query intersects more separable buckets on the same level and accesses them. However, the number of accesses can still be reduced: i) if the query region is entirely contained in the exclusion set of a level i, no object qualifying the query can be found in the separable buckets of this level and the next level is considered directly, ii) if the query region is completely contained in a separable set of a level i, the following levels, as well as, the global exclusion bucket need not be accessed and the search is terminated. The D-index also supports nearest neighbor(s) queries.

2.3 Critical Study

A query evaluation in the D-index can be separated into two stages. First, the *addressing* step identifies buckets to be examined by using the addressing structure. Second, the *execution* step is responsible for retrieving qualifying data objects in the buckets identified during the first stage. In buckets, a naive sequential scan with the pivot-based filtering is used [21]. The execution step can be easily parallelized since the evaluation in individual buckets is totally independent. The Grid computational environment is an ideal for such a structure because the addressing stage can be done on a dedicated Grid node and the execution step can be spread over other nodes.

The D-index is a dynamic structure but the dynamicity is ensured using elastic buckets which have unlimited capacity. The first disadvantage of the D-index is its fixed addressing structure that must be manually tuned by the user before data are loaded. From the data scalability point of view, when volume of data to be organized is doubled, the average occupation of buckets is doubled as well, so there is a strictly-linear correlation between the indexed volume and the execution costs in buckets whereas the addressing costs stay constant. This is inconvenient because the constant addressing structure leads to a constant number of buckets, i.e., Grid nodes. In order to guarantee the same response time, the power of individual Grid nodes must therefore be increased and adding further nodes will not help.

The experiments conducted on both synthetic and real-life datasets [11] revealed that the D-Index is very efficient and outperforms the M-tree[1] [8] nearly in all situations. However, these performance trials also exemplify the second disadvantage of the D-index: The partitioning principles are not very optimal and are prone to produce unbalanced buckets [9]. The unbalanced buckets would lead to a non-uniform load of Grid nodes. The static addressing structure is also unable to adapt to a radical change of distribution of data being inserted.

A fully adaptive structure and a balanced partitioning would allow expanding or shrinking to the desired number of Grid nodes while guaranteeing the response time. In particular, we concern the problem of selecting reference objects (pivots). Next, we deal with the issue of combining several rho-split functions.

The problem of choosing pivots is important for any search technique in general metric spaces, because all such algorithms need, directly or indirectly, some "anchors" for partitioning and search pruning. It is well known that the way in which pivots are selected affects the performance of search algorithms. This has been recognized and demonstrated by several researchers [19, 4]. Recently, the problem was systematically studied in [6], and several strategies for selecting pivots have been proposed and tested. The generic conclusion is that good pivots are i) far away from the remaining objects of the metric space and ii) far away from each other.

In the following, we analyze possible design strategies of rho-split functions.

3. METRIC GRID

The Metric Grid (M-Grid) is a distributed index structure for similarity searching that exploits a Grid infrastructure. The idea behind this structure is straightforward and has already been slightly depicted in the previous section. The D-index's schema is applied in the M-Grid. In particular, we define a special Grid node named the *master node* where the addressing structure is stored. This node is responsible for contacting buckets to be searched. The storage is distributed over all other Grid nodes.

Distributing the storage over Grid nodes requires a special mapping that assigns buckets to nodes. It is called the *bucket-to-node mapping*. This translation of bucket identifications to node identifications allows to have more buckets stored on the same node. It also permits to replicate buckets, i.e., the same bucket can be kept on multiple nodes. We use a simple load-balancer of replicas which is introduced in Section 4.3.

When a user poses a similarity query, the master node identifies the buckets that might contain relevant data. The bucket-to-node mapping is consulted to get addresses of Grid nodes. The master node initiates the execution phase by forwarding the query to the respective Grid nodes where the objects qualifying the query get retrieved. The individual sub-answers are then sent back to the master node which merges them to form the final answer to the query. A schema of M-Grid and of the querying process is depicted in Figure 2.

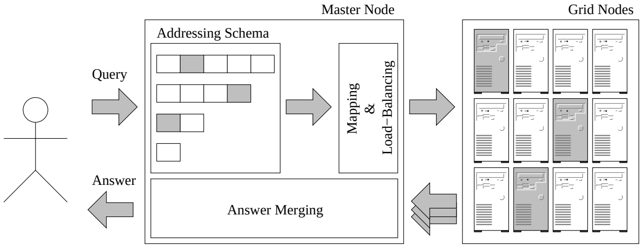

Figure 2: The schema of M-Grid and the process of querying.

In the previous section, we have pointed out a disadvantage of the D-index's – its static addressing structure. The M-Grid exploits principles of linear hashing to make the addressing structure adaptable. The design of addressing structure requires a specification of several ρ-split functions which are usually combinations of *bps* functions. In general, the idealized split function should produce balanced buckets each containing nearly the same number of objects and

[1] the de facto standard for similarity searching in metric spaces

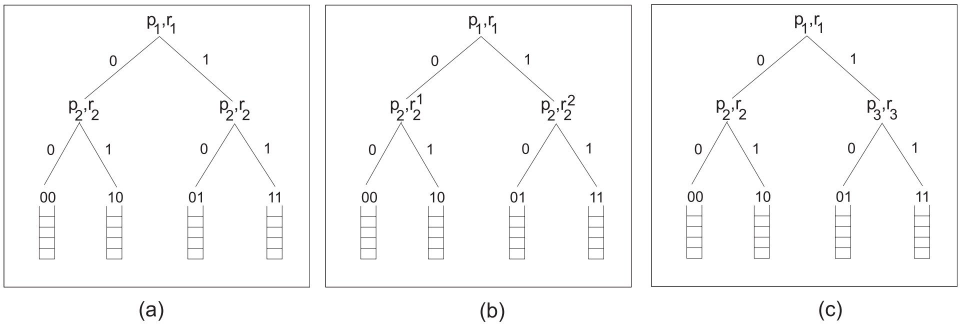

Figure 3: Different strategies of combining two *bps* split functions.

minimize the size of the exclusion bucket. Figure 3 presents possible strategies to combine two *bps* functions.

The first technique, depicted in (a), is utilized in the D-Index (also refer to Figure 1b) and uses the pivot p_1 and $d_m = r_1$ to divide the space into two separable sets. Next, these two sets are repartitioned using a different *bps* function which applies a different pivot p_2 and $d_m = r_2$, however, the same for both the sets. As a result, we obtain four separable buckets. We refer to this method as the *strict strategy* from here and on.

The second strategy in (b) differs from the first one in one aspect. It makes use of the pivot p_2 in the second function as well, but two different values of d_m (r_2^1 and r_2^2) are applied for the left and the right set, respectively. The hypothesis behind is that by manipulating d_m, better-balanced buckets are achieved, empty buckets are diminished, and the occupation of exclusion sets is decreased. We refer to this as the *variable strategy.*

To complete the list of split policies, the third strategy modifies also the pivot and is denoted as the *loose strategy*, see Figure 3c. In details, this technique can be viewed as the application of three independent *bps* functions instead of two in the previous strategies. However, the major disadvantage of this approach are higher memory requirements, e.g., for obtaining 128 separable buckets we need 127 different pivots, which is in a sharp contrast with the former strategies that require only $\log(128) = 7$ pivots to define the same partitioning. The memory requirements are not the only issue: the more pivots we have the more distance computations we must evaluate during search operations. Since we optimize also the costs in terms of distance computations, such behavior is not desired.

The experiments in [9] shows that the variable strategy with respect to the strict one is able to diminish all empty (or under-occupied) buckets and to reduce the number of pivots needed at the same time. In this respect, this leads to a more compact structure and a higher average bucket occupation. For this reason, we will not deal with the strict strategy in the following. We compare only the variable and loose strategies. The global observation is that the query execution costs with the loose strategy are by 20% to 50% less than the query execution costs with the variable strategy depending on the query radius. The reason is quite obvious. Following the example above, in the case of the loose strategy the pivot-based filtering in buckets can make profit of using all 127 pivots, thus it is very effective. It is in a sharp contrast with seven pivots in the case of the variable (or strict) strategies. In our Grid scenario, the loose strategy however yields very high load on the master node. More details are given in the experimental evaluation below.

Updates in M-Grid are handled in the following way. If no change in the addressing schema is required, an object is directly inserted to or deleted from the bucket that have been identified by the addressing schema. In the case of a bucket split, the master node updates its addressing schema and contacts the corresponding node that maintains the bucket with the necessary information about the new bucket. The contacted node then reallocates a portion of its data. The master node can also request to create a new replica. To keep all replicas synchronized, similar algorithms as used in distributed databases [17] can be applied. For space constraints, we do not analyze this phenomenon here.

4. PERFORMANCE TRIALS

To study properties of M-Grid, we have run several groups of experiments. First, we focused on different building strategies, namely the variable and loose strategies. Next, we tackled the problem of storing several buckets on a single node and the issue of replication. Finally, we provide the reader with a performance tuning analysis.

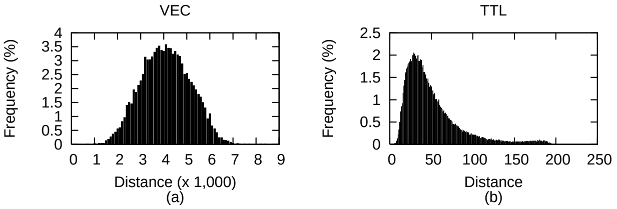

Figure 4: Distance densities for the VEC and TTL datasets.

We selected the following significantly different real-life datasets to conduct the experiments on:

VEC 45-dimensional *vectors* of extracted color image features. The similarity function d for comparing the vectors is a *quadratic-form distance* [18]. The distance distribution of the dataset is quite uniform and such a high-dimensional data space is extremely sparse, see Figure 4a.

TTL titles and subtitles of Czech books and periodicals collected from several academic libraries. These *strings* are of lengths from 3 to 200 characters and are compared by the *edit distance* [15] on the level of individual

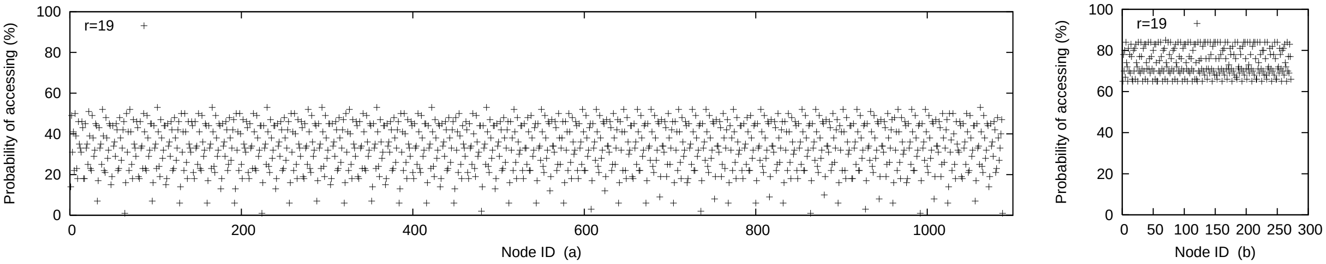

Figure 5: Probability of accessing a node for TTL dataset: (a) managing single bucket and (b) managing four buckets.

characters. The distance distribution of this dataset is skewed, refer to Figure 4b.

The stored data volume was 1,000,000 objects. The capacity of a bucket was set to 2,048 objects and was fixed for all the tested structures and both the datasets.

The building phase of M-Grid required 1,017 and 1,087 data reallocations, i.e., bucket splits, for **VEC** and **TTL** datasets, respectively. The amount of transferred objects was 1,873,767 and 1,520,359, which means that after the initial insertion, 873,767 and 520,359 objects were reallocated to another node, for respective datasets. We have not implemented any bulk-loading procedure which might further reduce the data transfer, so these numbers represent the upper bound.

All the presented performance characteristics of query processing have been taken as a sum over 100 range queries with randomly chosen query objects, if not stated otherwise. The query radii were 200, 1,000 and 2,000 for **VEC** dataset each returning on average 4, 19 and 20,000 objects, respectively. For **TTL** we used 7, 10 and 19 each returning on average 4, 40 and 20,000 objects, respectively.

4.1 Strategy Comparison

The parameter ρ is the only parameter of the addressing structure left to be user specified. We studied its influence mainly on the query costs. The value of ρ also affects the building costs. In general, the higher the value of ρ, the more objects fall into the exclusion set of a ρ-split function. This leads to more levels of addressing structure, so more distance computations are evaluated to insert an object. The reader might think the best value is zero, but it is not always true because no index structure is built without the aim of searching.

From the query costs point of view, the situation is not very clear. Table 1 presents query costs in terms of distance computations for the variable strategy. During the addressing stage, the number of distance computations is increasing as ρ is getting larger, so it supports the assertion stated above. The costs in the execution stage represent the total number of distance computations made on all nodes. These numbers are mean values for one query. This characteristic has a descending tendency. It is more noticeable for greater query radii ($r = 2,000$ in the table) than for smaller ones ($r = 200$ in the table). Such a trend is obviously caused by the improved effectiveness of pivot-based filtering since more pivots are used in the addressing structure for larger values of ρ.

var	$r = 200$		$r = 2,000$	
VEC	**Stage**		**Stage**	
ρ	**Addr.**	**Exec.**	**Addr.**	**Exec.**
0	10	157	10	404,911
50	59	97	69	374,099
100	88	119	116	345,489

Table 1: Distance computations for different values of ρ with the variable strategy and VEC dataset.

loose	$r = 200$		$r = 2,000$	
VEC	**Stage**		**Stage**	
ρ	**Addr.**	**Exec.**	**Addr.**	**Exec.**
0	26	55	632	321,335
50	53	50	751	329,362
100	75	47	931	314,505

Table 2: Distance computations for different values of ρ with the loose strategy and VEC dataset.

Table 2 shows the same features but in the case of loose strategy. In overall, the trends here are the same, however, the effectiveness of pivot-based filtering does not influence the results very much. For many pivots the filtering effectiveness is becoming quite stable and is not improved even if the number of pivots is doubled. In particular, the loose strategy used 1,000, 1,100 and 1,500 pivots in total for $\rho = 0$, 50 and 100, respectively. The significantly increased number of pivots in the loose strategy with respect to the variable strategy leads to more efficient search during execution stage.

For M-Grid, the best setting is $\rho = 0$ which implies the lowest costs in the addressing stage. Recall that these costs cannot be parallelized and form the total costs on the master node. In this respect, the favorable strategy for the addressing structure is the variable one because of its negligible requirements in terms of distance computations even for large query radii. The worse query costs of the variable strategy in the execution stage are not critical owing to parallelism.

4.2 Creating Groups

In this section, we focus on the probability of accessing buckets during query elaboration. Figure 5a depicts this for the **TTL** dataset and queries with radii $r = 19$ which return about 20,000 objects. The observation is that buckets are not visited very uniformly[2], which leads to an unbalanced load of Grid nodes. However, there is no long sequence of

[2] The probability of visiting varies from 1% to 53%.

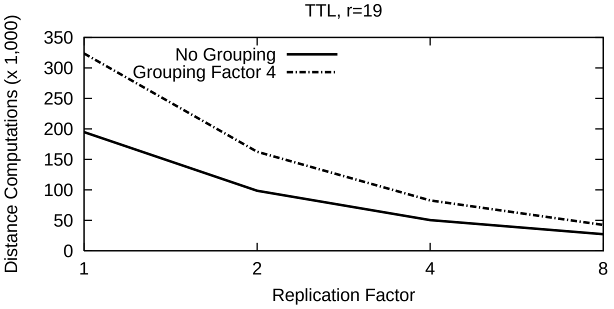

Figure 6: Response time in distance computations while changing the replication factor.

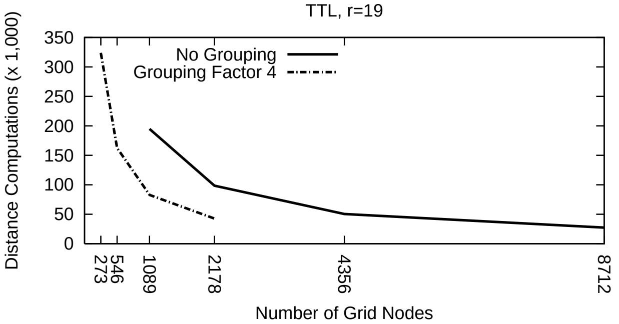

Figure 7: Response time in distance computations while changing the number of replicas.

buckets that would be contacted very often or very rarely. In our scenario, we assumed every bucket is managed by a separate Grid node. From the storage occupation point of view, one bucket per computer is probably not an overwhelming volume. The idea to make the probabilities more uniform is to group several buckets together and store them on the same Grid node.

Figure 5b presents the probabilities when groups of four buckets have been created – the *grouping factor* is four. It is noticeable that a more uniform load of Grid nodes is achieved[3]. The negative fact about grouping is the increased probability of contacting a node. For larger grouping factors, the probability is obviously 100% and all nodes get contacted for any query. As for CPU costs, the most loaded node evaluated 194,806 distance computations to complete 100 queries with $r = 19$ when grouping was disabled. The most loaded node organizing a group of four buckets computed the distance function 323,841 times, which is by 66% more. On the other hand, we need four times fewer computers in this case.

In general, this technique is applicable to any dataset. Nonetheless, the application of this technique results in increased demands on a Grid node in terms of storage and CPU costs. If CPU costs are crucial the following technique – replication – can be applied.

4.3 Replication

The replication is a well-known technique in distributed database systems that increases the availability of resources [17, 5]. The other implication of replication is the ability of balancing the load among more nodes that manage identical data, which is the objective of this section.

In our environment, the contribution of replication is twofold. First, it allows a query to be completed earlier. However, this holds only if grouping is applied. In spite of it a system with replication cannot answer the query more efficiently than the configuration with every bucket assigned to a separate Grid node. Second, more important one, additional queries can be completed at the same time.

Figure 6 presents the number of distance computations to answer 100 queries issued in parallel while changing the replication factor from one replica to eight copies. The trends are linear, i.e., doubling the replication factor leads to the half number of distance computations. The master node is responsible for distributing computations over the replicas. We have implemented a simple load-balancer which selects the least loaded node among other replicated nodes and forwards the incoming request to it. By the load of a node we mean the number of distance computations since it forms the main part of all costs.

Referring back to Figure 6, it seems that the grouping technique only deteriorates the performance because the configuration with the grouping factor of 4 is systematically slower. However, notice that the grouping factor of 4 implies four times fewer Grid nodes. The same results but rescaled to the real number of nodes are presented in Figure 7. It reveals that the configuration with the grouping factor of 4 and two replicas (546 nodes) outperforms the simple setting (one bucket to one node, i.e., 1089 nodes). Moreover, it operates on a half of Grid nodes only. On the same number of nodes, the configuration with the grouping factor of 4 and four replicas results in only 40% distance computations of the simple setting. These observations are, of course, a consequence of using grouping and replication techniques which redistribute the load more uniformly. Such results require a deeper analysis which we call performance tuning and which is tackled in the following section.

4.4 Performance Tuning

Performance tuning of M-Grid is very important because it has many parameters that must be properly set in advance. Except ρ and the bucket capacity, we have introduced two new parameters: the grouping and replication factors.

The user or administrator of a system usually has a clear notion about the maximum response time of a query and the total number of queries that will be issued simultaneously. They also suggest maximum memory requirements that are available on individual Grid nodes and the maximum number of available Grid nodes. Our experimental setting was at maximum 64 buckets per node and in total 4,100 Grid nodes available. We used the **VEC** dataset here and queries with radius r =2,000. Notice that such queries are quite demanding because they return about 20,000 objects. The dataset was organized in 1,022 buckets.

In order to study the performance of M-Grid, we have temporarily fixed the grouping and replication factors. Figure 8 presents costs to answer a batch of 10 or 100 queries for different M-Grid configurations but limited to bounds stated above. Different graphics distinguishes individual grouping factors. The horizontal line in each graph constitutes the limit on the query response time. Only the configurations

[3] The probability of visiting varies from 65% to 84%.

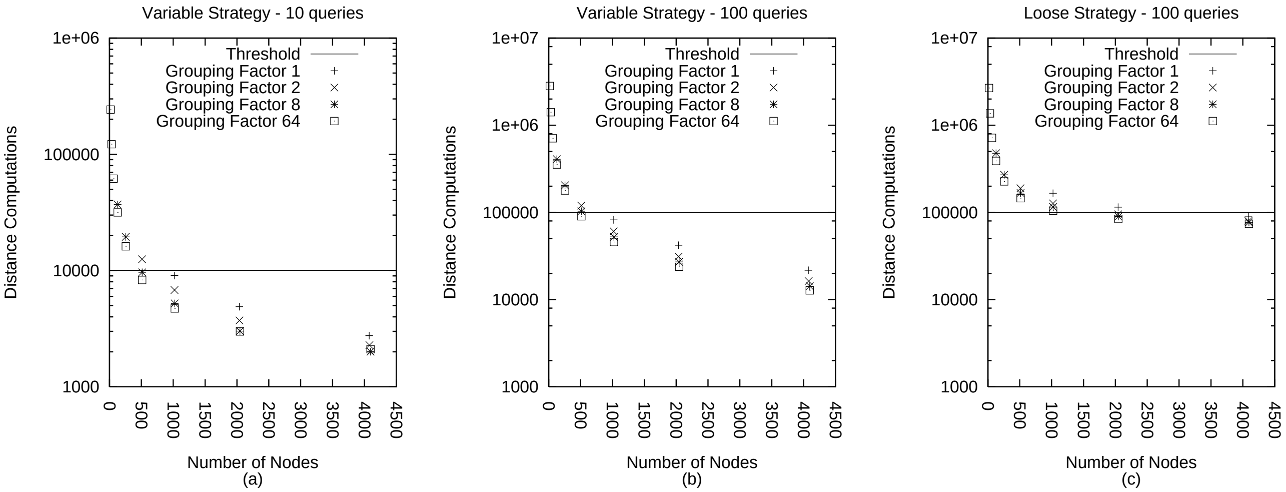

Figure 8: Response times in distance computations for different configurations of M-Grid. (a) 10 simultaneous queries and (b) 100 simultaneous queries for the variable strategy, and (c) 100 simultaneous queries for the loose strategy.

under it satisfy all user restrictions.

From the figures, we can observe that the most efficient configurations are ones with the grouping factor of 64. The M-Grid with the grouping factor greater than 64 for both **VEC** and **TTL** datasets has the probability of accessing a node equal to 100%. Therefore, all nodes are almost equally loaded and used optimally in this respect. In Figure 8a, the user requires ten simultaneous queries to be completed in 10,000 distance computations. Only configurations running on at least 512 nodes can meet these requirements. The most economical setting uses the grouping factor of 64 and 512 nodes, which leads to 32 replicas[4]. The other configuration almost equivalent in power uses 512 nodes as well, but the grouping factor is eight, which results in four replicas.

Figure 8b depicts the same but the user specified the threshold of 100,000 distance evaluations and 100 simultaneous queries. The results for various configurations are the same but shifted up. The two configurations emphasized above both are the most efficient solutions again. The latter configuration, however, needed 102,123 distance computations.

The loose strategy has a disadvantage of increased costs during addressing. These costs form a sequential step in processing queries, so the results of the same configurations are worse by 62,202 distance computations than the results for the variable strategy. Due to this fact, the desired configurations operate on at least 2,048 nodes, as shown in Figure 8c. The best configuration has the replication factor of 128 and the grouping factor of 64. The second best one uses the grouping factor of 8, but the replication factor is now 16.

The trends in all these graphs have also shown that doubling the number of Grid nodes improves the distributed power of the system twice, i.e., a query is answered in half distance computations. From the memory consumption point of view, configurations with smaller grouping factors are preferable if nodes cannot maintain too much data for some reason.

The advantage of M-Grid's addressing schema is that it is based on principles of linear hashing. Hence, new buckets are allocated as the data volume is increasing. By monitoring performance indicators, the M-Grid can increase the grouping factor or the replication factor gracefully to adjust the performance automatically.

5. CONCLUSIONS

We have proposed a dynamic distributed similarity searching structure suitable for Grid infrastructures. This technique can adapt its addressing schema as new data arrive and, more importantly, as specific performance requirements are updated. In performance trials, we have studied several design and performance tuning issues of the M-Grid.

Using our prototype Java implementation, we were able to conduct numerous experiments. Even though our technique can be used in a pure distributed environment, there are specific issues such as node allocation, node migration, data replication, security and access restrictions to be solved. This is the area where a Grid infrastructure comes into account. It solves these problems effectively and transparently through middleware services for node and data management. The Grid also provides embedded authentication and authorization capabilities. Unfortunately, we have had only a limited access to large Grid systems and we had to simulate its infrastructure using a distributed framework library on a huge network of common workstations that have been available for daily use by university students. Therefore, we did not use the actual response time as a measure, because we could not dedicate these computers to our application only. Instead, we report the number of evaluations of the distance function which practically form the prevalent part of query execution costs. However, to estimate the actual response times, we have measured that 13,000 evaluations of the quadratic-form distance took approximately one second to complete on a current computer. As for communication costs, they can be neglected, because we need only one message from the master node to all the processing nodes. Thus, the costs are equal to the longest round trip time, which depends mainly on the topology of the underlying Grid. However, even for a geographically large Grid they

[4] $replicas = \frac{nodes \cdot grouping}{buckets}$, i.e., $\frac{512 \cdot 64}{1022}$

can be expected to be maximally hundreds of milliseconds.

Our experiments revealed that the M-Grid can be tuned to meet even very tight performance conditions. In the performance tuning section, we used queries with the radii r =2,000 each returning around 20,000 objects. Such a setting is rather high. In practice, smaller queries giving tens of similar objects are more likely. Also the requirement of 100 simultaneously executed queries is immoderate implying a heavy-loaded system. The response time equivalent to 100,000 distance evaluations was about 8s in the worst case, which is reasonable. On the other hand, the experiment with more relaxed requirement of 10 concurrent queries had to be completed in 10,000 distance computations (equals to 0.8s), which is real-time processing. According to the experiments with one million data items, such high expectations can be fulfilled with 512 Grid nodes. Assuming more likely requirements of radii r =1,000 and the response time less than 2.5s, the system would require only 30 nodes, which is a feasible environment for any institution.

Even though we had to simulate the Grid infrastructure, our distributed framework provided the necessary functionality in such a way that a transition to the environment of Czech national METACentrum Grid [1] should be easy to implement. We plan such activity in the near future. We also intend to compare the M-Grid with existing systems for similarity searching based on peer-to-peer paradigms. In particular, the GHT* [2], MCAN [12], and M-Chord [16] will be considered. The first one is based on a distributed tree while the others exploit transformation principles to embed a metric space to a vector space.

5.1 Acknowledgements

This research has been partially funded by the national research project 1ET100300419.

6. REFERENCES

[1] METACentrum project, August 2006. http://meta.cesnet.cz/.

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

[3] M. Batko, D. Novak, F. Falchi, and P. Zezula. On scalability of the similarity search in the world of peers. In *Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 – June 1, 2006*. ACM Press, 2006.

[4] T. Bozkaya and Z. M. Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS 1999)*, 24(3):361–404, 1999.

[5] S. Budrean, Y. Li, and B. C. Desai. High availability solutions for transactional database systems. In *7th International Database Engineering and Applications Symposium (IDEAS 2003), 16-18 July 2003, Hong Kong, China*, pages 347–357. IEEE Computer Society, 2003.

[6] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. In *Proceedings of the 21st Conference of the Chilean Computer Science Society (SCCC 2001), Punta Arenas, Chile, November 6-8, 2001*, pages 33–40. IEEE Computer Society, 2001.

[7] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys (CSUR 2001)*, 33(3):273–321, September 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 1997), Athens, Greece, August 25-29, 1997*, pages 426–435. Morgan Kaufmann, 1997.

[9] V. Dohnal. An access structure for similarity search in metric spaces. In *Current Trends in Database Technology - EDBT 2004*, volume 3268 of *LNCS*, pages 133–143. Springer, 2004.

[10] V. Dohnal. *Indexing Structures for Searching in Metric Spaces*. PhD thesis, Faculty of Informatics, Masaryk University in Brno, Czech Republic, May 2004. http://www.fi.muni.cz/~xdohnal/phd-thesis.pdf.

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

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

[13] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Amsterdam: Elsevier, 2005.

[14] C. Gennaro, P. Savino, and P. Zezula. Similarity search in metric databases through hashing. In *Proceedings of the 3rd ACM Multimedia 2001 Workshop on Multimedia Information Retrieval (MIR 2001), Ottawa, Ontario, Canada, October 5, 2001*, pages 1–5. ACM Press, 2001.

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

[16] D. Novak and P. Zezula. M-Chord: A scalable distributed similarity search structure. In *Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 - June 1*. IEEE Computer Society, 2006.

[17] M. T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys*, 28(1):125–128, 1996.

[18] T. Seidl and H.-P. Kriegel. Efficient user-adaptable similarity search in large multimedia databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 1997), Athens, Greece, August 25-29, 1997*, pages 506–515. Morgan Kaufmann, 1997.

[19] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM Symposium on Discrete Algorithms (SODA 1993), Austin, Texas, USA, January 25-27, 1993*, pages 311–321. ACM Press, 1993.

[20] P. N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *Proceedings of the 6th DIMACS Implementation Challenge: Near Neighbor*

Searches (ALENEX 1999), Baltimore, Maryland, USA, January 15-16, 1999, 1999.

[21] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*. Springer, 2005.