Hybrid Index for Metric Space Databases

Mauricio Marin[1], Veronica Gil-Costa[2], and Roberto Uribe[3]

[1] Yahoo! Research, Santiago, Chile
[2] DCC, Universidad Nacional de San Luis, Argentina
[3] DCC, Universidad de Magallanes, Chile
mmarin@yahoo-inc.com

Abstract. We present an index data structure for metric-space databases. The proposed method has the advantage of allowing an efficient use of secondary memory. In the case of index entirely loaded in main memory our strategy achieves competitive performance. Our experimental study shows that the proposed index outperforms other strategies known to be efficient in practice. A valuable feature of the proposal is that the index can be dynamically updated once constructed.

1 Introduction

Searching in metric spaces is a very active research field since it offers efficient methods for indexing and searching by similarity in non-structured domains. For example, multimedia databases manage objects without any kind of structure like images, audio clips or fingerprints. Retrieving the most similar fingerprint to a given one is a typical example of similarity search. The problem of text retrieval is present in systems that range from a simple text editor to big search engines. In this context we can be interested in retrieving words similar to a given one to correct edition errors, or documents similar to a given query. We can find more examples in areas such as computational biology (retrieval of DNA or protein sequences) or pattern recognition (where a pattern can be classified from other previously classified patterns).

Similarity search can be trivially implemented comparing the query with all the objects of the collection. However, the high computational cost of the distance function, and the high number of times it has to be evaluated, makes similarity search very inefficient with this approach. This has motivated the development of indexing and search methods in metric spaces that make this operation more efficient trying to reduce the number of evaluations of the distance function. This can be achieved storing in the index information that, given a query, can be used to discard a significant amount of objects from the data collection without comparing them with the query.

Although reducing the number of evaluations of the distance function is the main goal of indexing algorithms, there are other important features. Some methods can only work with discrete distance functions while others admit continuous distances too. Some methods are static, since the data collection cannot grow

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 327–336, 2008.

once the index has been built. Dynamic methods support insertions in an initially empty collection. Another important factor is the possibility of efficiently storing these structures in secondary memory.

Search methods in metric spaces can be grouped in two classes [2]: pivot-based and clustering-based search methods. A pivot-based strategy selects some objects as *pivots* from the collection and then computes the distance between the pivots and the objects of the database and use this information to group related objects. This method selects a subset of objects from the collection as pivots, and the index is built computing and storing the distances from each of them to the objects of the database. During the search, this information is used to discard objects from the result without comparing them with the query. Clustering techniques partition the collection of data into groups called *clusters* such that similar entries fall into the same group. Thus, the space is divided into zones as compact as possible, usually in a recursive fashion, and this technique stores a representative point ("center") for each zone plus a few extra data that permit quickly discarding the zone at query time. In the search, complete regions are discarded from the result based on the distance from their center to the query.

In this paper we propose a combination of two existing methods (Sec. 2). The first method is used as it is proposed by their authors whereas the second one has been highly optimized by us to deal with secondary memory efficiently and very importantly to reduce the running time by increasing the ability of the strategy to quickly discard objects that cannot be part of the solution to a given query (Sec. 3). We present a complete evaluation of the performance of the proposed strategy in Sec. 4 which shows that our strategy consistently outperforms all others in practice. Sec. 5 presents concluding remarks.

2 Metric Spaces and Indexing Strategies

A *metric space* $(\mathbb{X}, d)$ is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects. The goal is, given a set of objects and a query, to retrieve all objects close enough to the query. This function holds several properties: strictly positiveness ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq d(x, y)+d(y, z)$). The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called the database and represents the collection of objects.

A k-dimensional vector space is a particular case of metric space in which every object is represented by a vector of k real coordinates. The definition of the distance function depends on the type of the objects we are managing. In a vector space, d could be a distance function of the family $L_s(x, y) = \sum_{i<=i<=k}(|x_i - y_i|^s)^{1/s}$. For example $s = 2$ yields Euclidean distance, that is the number of insertions, deletions or modifications to make two words equal.

There are three main queries of interest,

- *range search:* that retrieves all the objects $u \in \mathbb{U}$ within a radius r of the query q, that is: $(q, r)_d = \{u \in \mathbb{U}/d(q, u) \leq r\}$;

- *nearest neighbor search:* that retrieves the most similar object to the query q, that is $NN(q) = \{u \in \mathbb{U} / \forall v \in \mathbb{U}, d(q,u) \leq d(q,v)\}$;
- *k-nearest neighbors search:* a generalization of the nearest neighbor search, retrieving the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q)$, $v \in \mathbb{U} - kNN(q), d(q,u) \leq d(q,v)$.

We focus on range queries since nearest neighbor queries can be rewritten as range queries in an optimal way [2]. In the following we describe the data structures we combine to produce our metric-space index.

2.1 List of Clusters (LC)

This strategy [1] builds the index by choosing a set of centers $c \in \mathbb{U}$ with radius r_c where each center maintains a bucket that keep all objects that are within the extension of the ball (c, r_c). Each bucket contains the k objects that are the closet ones to the respective center c. Thus the radius r_c is the maximum distance between the center c and the k-nearest neighbor.

The buckets are filled as the centers are created and thereby a given element a located in the intersection of two or more center balls is assigned to the first center. The first center is randomly chosen from the set of objects. The next are selected so that they maximize the sum of the distances to all previous centers.

A range query q with radius r is solved by scanning in order of creation the centers. At each center we compute $d(q,c)$ and in the case that $d(q,c) \leq r_c$ all objects in the bucket associated with c are compared against the query. Also if the query ball (q,r) is totally contained in the center ball (c, r_c), there is no need to consider others centers.

2.2 Sparse Spatial Selection (SSS)

During construction, this pivot-based strategy selects some objects as pivots from the collection and then computes the distance between the pivots and the objects of the database [4]. The result is a table of distances where columns are the pivots and rows the objects. Each cell in the table contains the distance between the object and the respective pivot. These distances are used to solve queries as follows. For a range query (q,r) the distances between the query and all pivots are computed. The objects x from the collection that do not hold the condition $|d(p_i,x) - d(p_i,q)| \leq r$ for *all* pivots p_i can be immediately discarded due to the triangle inequality. The objects that pass this test are considered as potential members of the final set of objects that form part of the solution for the query and therefore they are directly compared against the query by applying the condition $d(x,q) \leq r$. The gain in performance comes from the fact that it is much cheaper to effect the calculations for discarding objects using the table than computing the distance between the candidate objects and the query.

A key issue for efficiency is the method employed to calculate the pivots, which must be effective enough to drastically reduce total number of distance computations between the objects and the query. To select the pivots set, let

$(\mathbb{X}, d)$ be a metric space, $U \subset \mathbb{X}$ an object collection, and M the maximum distance between any pair of objects, $M = \max\{d(x, y)/x, y \in \mathbb{X}\}$. The set of pivots contains initially only the first object of the collection. Then, for each element $x_i \in \mathbb{U}, x_i$ is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than $\alpha\, M$, being α a constant parameter. Therefore, an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots.

2.3 LC-SSS Combination (Hybrid)

We propose a combination between the List of Clusters (LC) and Sparse Spatial Selection (SSS) indexing strategies. In this case we both compute the LC centers and SSS pivots independently. We form the clusters of LC and within each cluster we build a SSS table using the global pivots and organization of columns and rows described above. We emphasize on *global* SSS pivots because intuition tells that in each cluster of LC one should calculate pivots with the objects located in the respective cluster. However, we have found that the quality of SSS pivots degrades significantly when they are restricted to a subset of the database, and also the total number of them tends to be unnecessarily large. We call this strategy *hybrid*.

3 Optimizing Running Time and Secondary Memory

Our contribution to increasing the performance of the SSS index is as follows. During construction of the table of distances we compute the cumulative sum of the distances among all objects and the respective pivots. We then sort the pivots by these values in increasing order and define the final order of pivots as follows. Assume that the sorted sequence of pivots is p_1, p_2, ..., p_n. Our first pivot is p_1, the second is p_n, the third p_2, the fourth p_{n-1} and so on. We also keep the rows in the table sorted by the values of the first pivot so that upon reception of a range query q with radius r we can quickly (binary search) determine between what rows are located the objects that can be selected as candidates to be part of the answer. This because objects o_i being part of the answer can only be located between the rows that satisfies $d(p_1, o_i) \geq d(q, p_1) - r$ and $d(p_1, o_i) \leq d(q, p_1) + r$.

In practice, during query processing and after the two binary searches on the first column of the table, we can take advantage of the column x rows organization of the table of distances by first performing a few, say v, vertical wise applications of the triangular inequality on the objects located in the rows delimited by the results of the binary searches, followed by horizontal wise applications of the triangular inequality to discard as soon as possible all objects that are not potential candidates to be part of the query answer. See Fig. 1 which shows the case of two queries being processed concurrently.

For secondary memory the combination of these strategies have the advantage of increasing the locality of accesses to disk and the processor can keep in main

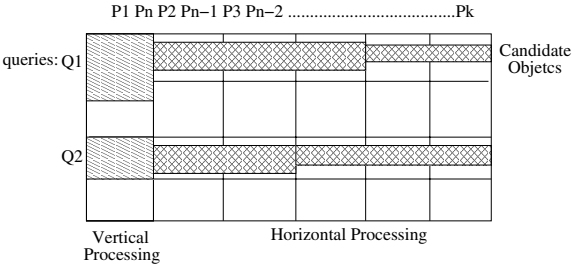

Fig. 1. Optimization to the SSS distance table

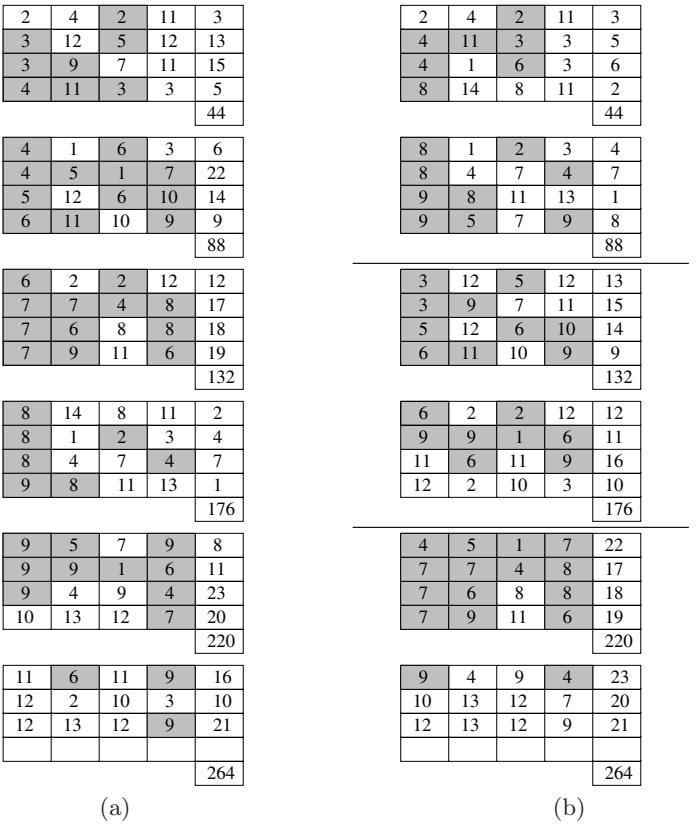

Fig. 2. Storing the distance table in blocks composed of a fixed number of disk pages

memory the first v columns of the table. In the experiments performed in this paper we observed that with $v = n/4$ we achieved competitive running times.

In the following we describe two feasible physical organizations of the index on disk pages. The description is illustrated in Fig. 2 which presents two cases

for the distribution of a distance table with 23 objects and 4 pivots. The table is partitioned in 5 blocks. The first 4 columns contains the distances from objects to the 4 pivots and the last column contains the respective object ID associated with each row. The cell located at the bottom-right indicates the physical address of the disk page containing the next table block. Each block is stored in contiguous disk pages. We assume that the main memory is large enough to store two blocks. Fig. 2.a represents a case in which all objects 1 ... 23 are available at construction time and Fig. 2.b a case in which objects are arriving one by one to the index and every time a block is filled up a new one is started. The first case requires a external memory sorting by the first pivot. In the latter case the first column is kept sorted every two blocks since we are assuming that they both fit into main memory. Thus external sorting is not required. In the next section we show that both strategies achieve a very similar performance which indicates that the scheme supports efficiently further updates once the index has been constructed from an initial set of objects.

In Fig. 2.a and 2.b, the grey cells represent the cases in which the triangular inequality gives a positive match for a range query q with $d(q, p_i) = \{6, 8, 3, 7\}$ for pivots p_i and radius $r = 3$. We assume that the query is solved by performing one vertical operation followed a horizontal operation for each row selected for the first pivot. In fact, as the first pivot is sorted by distance it is only necessary to perform two binary searches to detect the first row with value $d(q, p_1) - r = 3$ and the last row with value $d(q, p_1) + r = 9$. Then the sequence of horizontal applications of the triangular inequality determines that the objects 22, 17 and 11 are candidates which must be directly compared against the query object. Notice that a second vertical operation would have reduced significantly the number of horizontal operations (which is a tradeoff that depends on the application).

4 Experiments

The performance of Hybrid Index was tested with several collections of data. First, we used a collection of $100,000$ vectors of dimension 10, synthetically generated with Gaussian distribution. The Euclidean distance was used as the distance function when working with this collection. We also worked with a collection of $86,061$ words taken from the Spanish dictionary, and using the edit distance as the distance function. The algorithm was compared with other well-known clustering-based indexing methods: M-Tree [6], GNAT [8], EGNAT [7], Spatial Approximation Trees (SAT) [3]. We also included in the comparison the LC [1] and SSS [4] strategies, and a recent version of the SSS called the SSSTree [5] which uses a tree structure in which the SSS pivots are used to recursively divide the space.

4.1 Cost of Secondary Memory Access

In the left part of table 1 we show for the Spanish dictionary data set the total number of blocks and objects per block for cases in which we limit the total

number of pivots to 4, 8, 12, 16 and 20. The first three columns show the disk activity when constructing the index with the 90% of the data set by using the strategy despicted in Fig. 2.a. The last column shows the case when the same data is indexed on-line by using the strategy of Fig. 2.b. In this case no reads of blocks are effected and blocks are written to disk as soon as they become full during the insertion of objects. In the first case reads and seeks have to be performed in order to perform the sorting by the first column and move whole rows among blocks. However, the actual difference in running time between the two alternatives is negligible, presumebly because of disk-cache effects.

Table 1. Disk activity for index construction

Pivots	Blocks	Objects	Writes	Seeks	Reads	Writes
4	378	204	399	761	780	380
8	686	113	721	1373	1408	688
12	994	78	1030	1989	2025	996
16	1291	60	1342	2583	2634	1293
20	1614	48	1676	3229	3291	1616

The next 10% of the data set is used to perform range queries with radio 1, 2, 3 and 4. The Fig. 3.a and 3.b show the total number of block reads performed during the processing of queries for the two methods of index construction. The differences in disk ativity are irrelavant showing that both approaches achieve similar performance. However, for large radious 4 the on-line creation of the index tends to generate more activity because large radio tend to generate a large number of candidate objects which are expected to be evenly distributed onto all blocks.

4.2 Calls to the Distance Evaluation Function

Computing the distance between two complex objects is known to be very expensive in terms of running time in metric-space databases. This produces an implementation independent base upon which comparing different strategies. In the following we review previous studies on comparison of a number of metric-space index and then we compare the best performers with our proposal.

Fig. 4 and 5 show results for different data structures proposed so far. The Hybrid strategy achieves the best performance in terms of this metric though very similar to the LC strategy.

4.3 Comparing Running Times

In Fig. 6 we present results for running times with the different strategies. The proposed Hybrid achieves the best performance for most cases. Notice that structures such as the SAT achieves better performance than ours for range queries with large radio. The results suggests that SAT performs significantly better for large r. However, for these radio almost all objects are part of the solution to

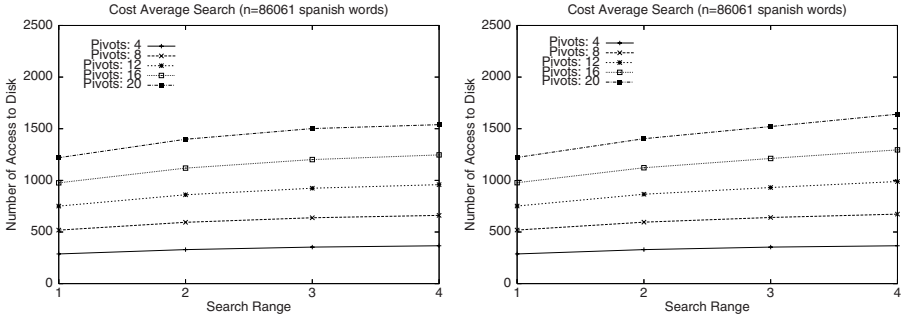

Fig. 3. Disk seeks and their respective block read for during range queries

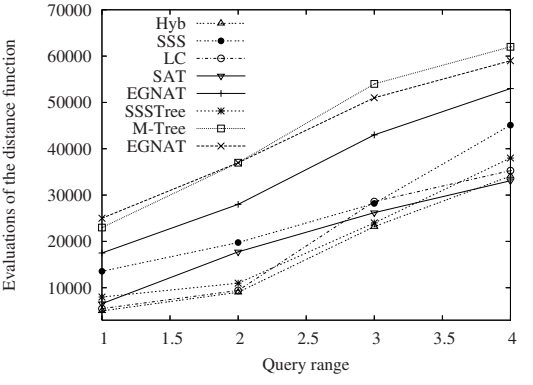

Fig. 4. Number of calls to the distance evaluation function per query for different metric-space index data structures. Results for the Spanish dictionary data set.

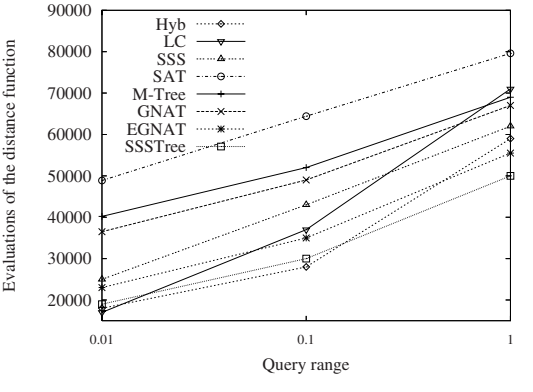

Fig. 5. Number of calls to the distance evaluation function per query for different metric-space index data structures. Results for the Gaussian vector space.

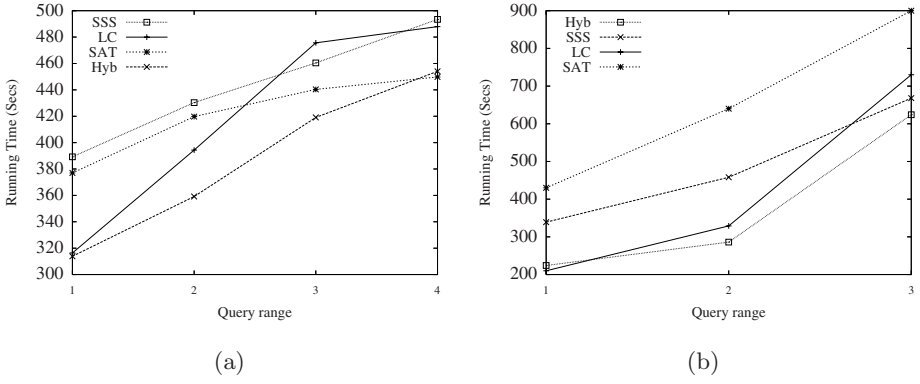

Fig. 6. Total running times for processing 10,000 queries with the Spanish dictionary (left) and a Gauss vector data set (right)

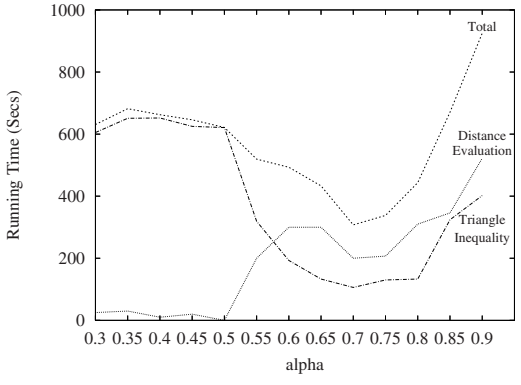

Fig. 7. Running time for the three main components in the execution of the Hybrid

the query and we do not see a practical use of queries like this ones in actual applications.

Finally Fig. 7 shows results for the cummulative running time involved in accessing the distance table and executing the distance evaluation function for different values of the parameter α, namely different number of pivots. The results show a tradeoff between both costs with optimum in $\alpha = 0.7$.

5 Conclusions

We hace presented a simple but very efficient strategy to solve queries in metric-space databases. Our strategy achieves best performance than most other strategies. However, it is not able to outperform in a significant manner to a tree

based structure called SSSTree which is in fact based on a strategy quite similar to ours. However, our strategy has clear advantages with respect to secondary management, total memory used by the index. Also the organization of the index in terms of a table with columns and rows allows it to exploit in an optimal way the parallelism available in the new computer architectures based on multi-cores devised to support multi-threading by hardware. We are currently evaluating the gain in performance in this architectures by solving queries using the standard openMP.

Acknowledgments. This work has been partially funded by FONDECYT project 1060776, UMAG PR-F1-002IC-06, and UNSL PICT 2002-11-12600.

References

1. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquyn, J.L.: Searching in metric spaces. ACM Computing Surveys 3(33), 273–321 (2001)
3. Navarro, G.: Searching in metric spaces by spatial approximation. The Very Large Databases Journal (VLDBJ) 711(1) (2002)
4. Brisaboa, N., Pedreira, O.: Spatial selection of sparse pivots for similarity search in metric spaces. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 434–445. Springer, Heidelberg (2007)
5. Brisaboa, N., Pedreira, O., Seco, D., Solar, R., Uribe, R.: Clustering-based similarity search in metric spaces with sparse spatial centers. In: Geffert, V., et al. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 186–197. Springer, Heidelberg (2008)
6. Ciaccia, P., Patella, M., Zezula, P.: 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), pp. 426–435 (1997)
7. Uribe, R., Navarro, G., Barrientos, R., Marin, M.: An index data structure for searching in metric space databases. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3991, pp. 611–617. Springer, Heidelberg (2006)
8. Brin, S.: Near neighbor search in large metric spaces. In: 21st conference on Very Large Databases (1995)