A Search Engine Index for Multimedia Content

Mauricio Marin[1,2], Veronica Gil-Costa[3], and Carolina Bonacic[4]

[1] Yahoo! Research, Santiago, Chile
[2] University of Santiago of Chile
[3] DCC, University of San Luis, Argentina
[4] ArTeCS, Complutense University of Madrid, Spain

Abstract. We present a distributed index data structure and algorithms devised to support parallel query processing of multimedia content in search engines. We present a comparative study with a number of data structures used as indexes for metric space databases. Our optimization criteria are based on requirements for high-performance search engines. The main advantages of our proposal are efficient performance with respect to other approaches (sequentially and in parallel), suitable treatment of secondary memory, and support for OpenMP multithreading. We presents experiments for the asynchronous (MPI) and bulk-synchronous (BSP) message passing models of parallel computing showing that in both models our approach outperforms others consistently.

1 Introduction

Dealing efficiently with multiple user queries, each potentially at a different stage of execution at any given instant of time, is a central issue in large-scale plain-text based search engines. Here the use of suitable parallel computing techniques devised to grant, among other optimizations, all queries an even share of the computational resources is crucial to reduce response time and avoid unstable behavior caused by dynamic variations of the query traffic. At the core of a plain-text search engine is a data structure used as an index that allows fast solution of queries.

New applications demand the use of data more complex than plain text. As such, it is reasonable to expect that in the near future search engines will be compelled to include facilities to handle metric space databases. Metric spaces are useful to model complex data objects such as images or audio. In this case queries are represented by an object of the same type to those in the database wherein, for example, one is interested in retrieving the top-R objects which are most similar to the query. The degree of similarity between two objects is calculated by an application-dependent function called the *distance function*.

A number of data structures and algorithms for metric spaces have been proposed so far [4] and papers on parallelization of some of these strategies have been presented in [5, 8, 9]. In this paper we propose a new strategy which satisfies demanding requirements from high-performance search engines. In the following we describe the two main principles we have identified as the ones leading to

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 866–875, 2008.

efficient performance in plain-text based search engines which we apply in the context of this paper.

Sync/Async search engines. This is about the specific way we organize the parallel processing of queries [7]. For plain text, we have observed that for low query traffic it is efficient to use the standard asynchronous message passing approach to parallel computing whereas for high traffic we can take advantage of bulk-synchronous parallel processing of queries. In this paper we show that this also holds for the metric-space context. Our proposal is openMP friendly in the sense that it can allow in-core threads to efficiently cooperate in the solution of queries. For the Sync and Async modes of operation the communication among nodes is performed by using MPI and BSPonMPI respectively whereas within nodes openMP is used to speed-up the processing of queries.

Round-Robin query processing. This assigns every query a similar share of key resources such as processors time and disk and network bandwidth [6]. In plain-text query processing we can decompose query solution in K-sized quanta of CPU, disk and network traffic where R is a fraction of K such as $1/2$. In this paper we show how to apply this strategy in the metric-space context. This requires careful consideration of the most costly parts of the solution to queries in very large distributed metric-space databases which are secondary memory management and load balance of distance calculations across processors.

The remainder of this paper is organized as follows. In section 2 we describe two basic data structures for metric-space databases which we combine and refine to propose our index data structure in section 3. Section 4 gives details on the parallel realization of the proposed index. We made similar implementations on top of other data structures for comparison purposes. Section 5 presents a comparative evaluation of our proposal including results from sequential and parallel executions. Section 6 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* $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ 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: strict 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. There are three main queries,

- *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 $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q,u) \leq d(q,v)$.

In the following we describe two indexing strategies which we combine into a single one to build up our search engine index.

2.1 List of Clusters (LC)

This strategy [3] 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 up as the centers are created and thereby a given element i 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 ones 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 $d(q,c) \leq r$ all objects in the bucket associated with c are compared against the query. This can end up at the first center found to hold $d(q,c) + r < c_r$, mining that the query ball (q,r) is totally contained in the center ball (c, r_c), or when all centers have been considered.

2.2 Sparse Spatial Selection (SSS)

During construction, this pivot-based strategy [2] selects some objects as *pivots* from the collection and then computes the distance between the pivots and the objects of the database. 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. An object x from the collection can be discarded if there exists a pivot p_i for which the condition $|d(p_i,x) - d(p_i,q)| > r$ does not hold. 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.

3 LC-SSS Combination and Refinements

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*.

We increase the performance of the SSS index 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 $\quad d(p_1, o_i) \geq d(q, p_1) - r \quad$ and $\quad d(p_1, o_i) \leq d(q, p_1) + r$. We have observed that this re-organization of pivots produces a SSS which is between 5 to 10 times faster than the original proposal in [2].

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 $\times$ 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 figure 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 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.

This scheme of a table of distances per cluster can have two possible organizations based on a set of blocks stored in several contiguous disk pages. The first

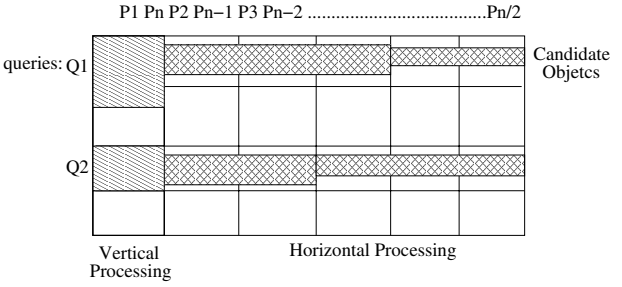


Fig. 1. Optimization to the SSS distance table

one is for the case in which we start with an existing collection of objects and requires sorting of the first pivot (column) across several blocks. In the second one we can insert new objects in an on-line manner. In this case blocks can contain objects as they were inserted with first columns sorted locally. Here sorting is spread across a few blocks, a number given by the amount of blocks that can be hold in main memory. In section 5 we show that both strategies are efficient in terms of total number accesses to disk.

4 Parallelism

In this section we describe the parallel algorithms we have devised to build the different index data structures and to process queries considering the two efficiency principles we enumerated in the introduction section of this paper, namely Sync/Async and Round-Robin parallel query processing.

For the Sync mode of operation we use the bulk-synchronous model of parallel computing (BSP) [11]. In BSP the parallel computer is seen as composed of a set of P processor local-memory components which communicate with each other through messages. The computation is organized as a sequence of supersteps. During a superstep, the processors may perform sequential computations on local data and/or send message to others processors. The messages are available for processing at their destination by the next superstep, and each superstep is ended with the barrier synchronization of processors. In our experiments we use a realization of BSP built on top of the MPI communication library. For the Async mode of operation we use the standard asynchronous message passing model of parallel computing implemented using the same MPI communication library.

The switching between the two modes of operation is effected in accordance with the observed query traffic. Our results show that in situations of low traffic it is more efficient to operate in the Async mode because the barrier synchronization of processors performed by the Sync mode under the same low traffic becomes too detrimental to performance as load balance degrades significantly. On the other hand, when query traffic is high we have a situation in which the

Sync mode can profit from economy of scale by performing optimizations such as bulk sending of messages among processors and proper load balancing of bulk query processing.

Queries are assumed to be received by a broker machine which in turn routes queries to processors. In our case, as we explain below, all queries are sent to all processors (broadcast). The broker can measure traffic to decide in which mode of operation the current queries can be processed. The arrival time of queries is unpredictable and the departure time of queries is also unpredictable along time. Thus the broker needs to estimate the average number of queries being processed during a fixed period of time and use this information to decide the mode of operation for the next period of time. The average number of queries can be determined as we propose in [7] which basically models the system as a $G/G/\infty$ queuing model where service time is given by the response time to queries.

The Round-Robin principle is achieved by assigning to each query being processed a similar amount of the resources. We explain it in the context of BSP. In each superstep, each query is granted a fixed number of distance calculations and in the case of SSS a fixed number of computations on the distance table. This also fixes the amount of communication effected at the end of the superstep and the number of disk accesses. Thus a given query can require several supersteps to be completed.

The database objects are uniformly distributed at random on the set of P processors. This marks the starting point to the design of our index construction and query processing algorithms. Query processing is effected by broadcasting each query to all processors and then each processor works on the partial solution of the query. Then a selected processor is in charge of collecting the partial solutions to integrate them and return the top-R results to the broker. In this case each processor sends its best R results. As we can have several queries being processed and the integrator processor for each query is chosen circularly among all processors, we can achieve a high degree of parallelism during query processing. Notice that both centers/pivots are the same at each processor so we can avoid distance recalculations among the queries and centers/pivots.

Constructing the LC-SSS index is effected as follows. For the List of Clusters (LC) strategy each processor selects its candidate centers using its local objects. Then, these lists of candidates are broadcast to all processors. After receiving the candidate list each processor computes the distance among the local centers and selects the ones maximizing the sum of distance. From this point no communication is required, and each processor can build its local index using the same global centers to organize into buckets its local objects.

In the Sparse Spatial Selection (SSS) strategy we do the same. Namely each processor selects the pivots candidates from the local object collection and broadcast them to all processors. Each processor receives the local pivots computed previously and then they refine these set of pivots selecting only the ones that satisfy the condition $d(p_i, p_j) \geq \alpha\, M$, $\forall i \neq j$. After that, each processor has the same set of pivots and can build the local distance table for each bucket.

5 Experimental Results

We performed experiments using a 32-processors cluster and different data sets and queries. We use two multimedia data sets. The first one is a collection of 47,000 images extracted from the NASA photo and video archives, each of them transformed into a 20-dimensional vector. The Euclidean distance is the distance function used in this data set. The second one is a large set of 900,000 words taken from documents crawled from the Chilean Web. In this case the number of characters required to make identical two words is the distance function. Using these collections of data we can study the behavior of the algorithm in spaces of different intrinsic dimensionality.

Figures 2 below have been drawn with the following convention. The Y-axis shows the total running time. The X-axis shows different cases for the query traffic, ranging from A (low traffic) to D (high traffic). The curves are divided into two areas, results for the Sync (BSP) and Async (MPI) modes of parallel computation. In each area we show results for the different strategies. For each strategy we show results for P= 4, 8, 16, and 32 processors. Notice that running time increases as we increase the number of processors. This is because we keep constant the total number of queries processed by each processor. That is, every time we double the number of processors we also double the overall number of queries that are processed. The total running time cannot be constant under this scenario since the communication hardware has at least $\log P$ latency.

Figures 2.a and 2.b show results for two MPI realizations of the parallel strategies explained in this paper. The first one is based on the asynchronous message passing (Async) approach and the second one is a bulk-synchronous MPI (Sync) realization. Figure 2.a shows results for the NASA collection whereas figure 2.b shows results for words space data set. In both cases, the results shows that the Hybrid algorithm (combining the LC and the improved SSS strategies) outperforms the others parallel strategies. The curves for SSS and LC alone were obtained with our modified SSS and LC running as a single strategy respectively.

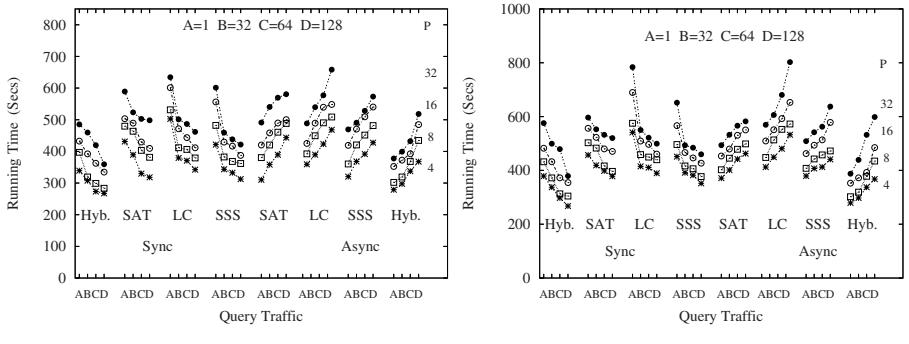


Fig. 2. Running time obtained using two data collections

These results also show that for high query traffic it is more efficient to process queries in bulk and for low traffic it is more efficient to process them individually in an asynchronous manner.

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. The load balance achieved during parallel processing is a clear indication of good use of resources. We measure load balance by using the efficiency metric which for the BSP model and for a given measure X we define as the average taken over all supersteps of the ratios $\text{avg}(X)/\max(X)$ observed in each superstep and considering all processors. In figure 3.a we show results for this metric obtained by counting the number of distance evaluations effected in each superstep and processor. The optimum indicating perfect load balance is shown by efficiency equal to 1. The results show that all strategies achieve good load balance, which is evidence that the better performance of our Hybrid index comes from factors such as reduction in the total number of distance computations and small overheads.

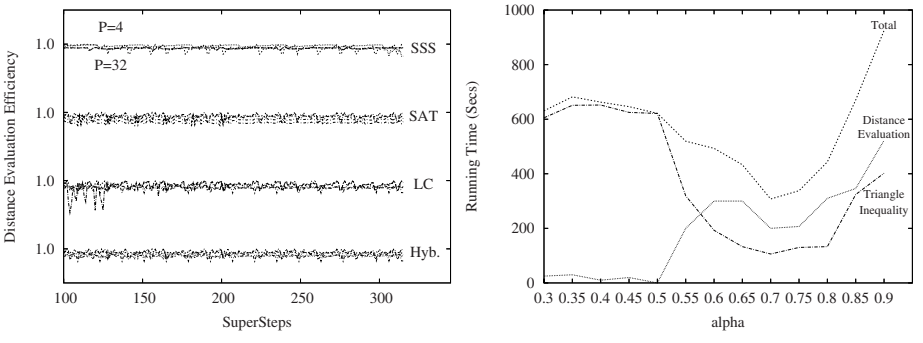
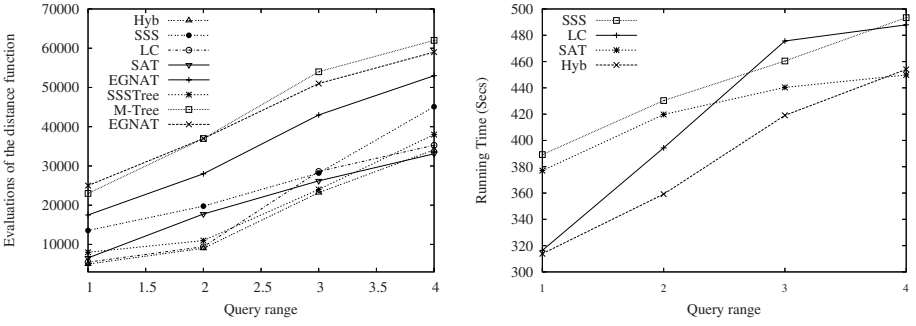


Fig. 3. Running time obtained using two data collections

In figure 3.b we show the effect of the α parameter in the total running time of the SSS strategy and thereby also in our Hybrid index. The results are for the words data set. The figure shows three curves, the one labeled *total* is the sum of the running time of the other two curves. The curve labeled *distance evaluation* is the total time spent computing distance evaluations between two objects and the third curve is the time spent computing on the distances table to reduce the number of candidate objects to be compared against the query object. Clearly for large data sets like the words one there is a value of $\alpha \approx 0.7$ that can reduce total running time significantly.

In the following we review previous studies in sequential computing on comparison of a number of metric-space index data structures and then we compare the best performers against our proposal. Figure 4.a shows results for the distance evaluation metric for different data structures proposed so far. Namely



(a) (b)

Fig. 4. Sequential computing: results for a 80,000 words Spanish dictionary data set

the M-Tree [4], GNAT [1], EGNAT [9] and SAT [10]. We also included in the comparison the SSS [2] and LC [3] strategies. To make this figure we took results reported by the authors on the same data set. The Hybrid strategy proposed in this paper achieves the best performance in terms of this metric though very similar to the LC strategy. Figure 4.b shows running time results for the best performers (using our own implementation of those strategies). The results are consistent with those of figure 4.a.

Finally we show results for an openMP optimization of the Hybrid index. As suggested in figure 1 we can take advantage of the contiguous memory realization of the index data structure which makes it suitable to run on it a team of openMP threads for vertical and horizontal traversals in each superstep. The openMP threads can also be used in the sequential implementation of the Hybrid index. We made experiments using openMP as implemented in `g++` version 4.1.2 and BSPonMPI (http://bsponmpi.sourceforge.net/). In table 1 we show the gain in performance by using T openMP threads on a *Intel's Quad-Xeon* machine (2 nodes, 16 CPUs in total). The second row in the table show results for the ratio sequential running time ($T = 1$) in one CPU to running time with $T \geq 1$ threads and query traffic 64. We process batches of 64 queries using T threads, and once all threads finish we process the next batch.

Table 1. Decreasing running times per node with openMP threads

T	1	2	4	6	8	10	12	14	16
$(T = 1)/(T \geq 1)$	1.00	1.76	2.12	2.81	4.07	3.52	2.01	0.88	0.93

6 Conclusions

We have proposed a distributed index data structure devised to support the efficient processing of queries in metric-space databases. Our index is suitable for

search engines dealing with multi-media data. We have implemented it by using parallel computing techniques devised to achieve high-performance in multiple-queries processing. We performed experiments on actual data sets upon a cluster of computers.

Our results show that our Hybrid index which is a combination of the List of Clusters (LC) and Sparse Spatial Selection (SSS) indexing methods, which we have optimized and tailored to our setting, is the strategy which achieves the best performance. We have verified that this efficient performance also holds for sequential computing. An important advantage of our proposal over all other alternative indexing methods is that our particular realization of the SSS index is friendly to secondary memory and multithreading (e.g., openMP) because it contains high locality in terms of data accesses.

References

1. Brin, S.: Near neighbor search in large metric spaces. In: 21st conference on Very Large Databases (1995)
2. Brisaboa, N.R., 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)
3. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
4. Chavez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys 3(33), 273–321 (2001)
5. Costa, G.V., Marin, M.: Distributed sparse spatial selection indexes. In: PDP 2008, Toulouse, France, February 13-15 (2008)
6. Marin, M., Gil Costa, V.: High-performance distributed inverted files. In: CIKM 2007, pp. 935–938. ACM, New York (2007)
7. Marin, M., Gil-Costa, V. (Sync|Async)$^+$ MPI Search Engines. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757. Springer, Heidelberg (2007)
8. Marin, M., Reyes, N.: Efficient Parallelization of Spatial Approximation Trees. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3514, pp. 1003–1010. Springer, Heidelberg (2005)
9. Marin, M., Uribe, R., Barrientos, R.J.: Searching and updating metric space databases using the parallel EGNAT. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4487, pp. 229–236. Springer, Heidelberg (2007)
10. Navarro, G.: Searching in metric spaces by spatial approximation. The Very Large Databases Journal (VLDBJ) 711(1) (2002)
11. Valiant, L.G.: A bridging model for parallel computation. Comm. ACM 33, 103–111 (1990)