Dynamic Permutation Based Index for Proximity Searching *

Karina Figueroa[1] and Rodrigo Paredes[2]

[1] Facultad de Ciencias Físico-Matemáticas, Universidad Michoacana, México.
[2] Departamento de Ciencias de la Computación, Universidad de Talca, Chile.
karina@fismat.umich.mx, raparede@utalca.cl

Abstract. Proximity searching consists in retrieving objects from a dataset that are similar to a given query. This kind of tool is an elementary task in different areas, for instance pattern recognition or artificial intelligence. To solve this problem, it is usual to use a metric index. The permutation based index (PBI) is an unbeatable metric technique which needs just few bits for each object in the index. In this paper, we present a dynamic version of the PBI, which supports insertions, deletions and updates, and keeps the effectiveness of the original technique.

1 Introduction

Similarity (or Proximity) Searching consists in retrieving the most similar elements to a given query from a dataset. This makes the proximity searching an elementary task in many areas where the exact searching is not possible. Examples of these areas are machine learning, speech recognition, pattern recognition, multimedia information retrieval or computational biology, to name few. The core of such areas is precisely a searching task and the common part is a dataset and a similarity measure among its objects.

Proximity queries can be formalized using the metric space model [3, 6, 8]. Given a universe of objects $\mathbb{X}$ and nonnegative distance function defined among them $d : \mathbb{X} \times \mathbb{X} \to R^+ \cup \{0\}$, we define the metric space as a pair $(\mathbb{X}, d)$. Objects in $\mathbb{X}$ do not necessarily have coordinates (think, for instance, in strings). On the other hand, the function d provides a dissimilarity criterion to compare objects from $\mathbb{X}$. In general, the smaller the distance between two objects, the more "similar" they are. The function d satisfies the metric properties, namely: *positiveness* $d(x, y) \geq 0$, *symmetry* $d(x, y) = d(y, x)$, *reflexivity* $d(x, x) = 0$, and *triangle inequality* $d(x, z) \leq d(x, y) + d(y, z)$, for every $x, y, z \in \mathbb{X}$.

In practice, we are working with a subset of the universe, denoted as $\mathbb{U} \subset \mathbb{X}$, of size n. Later, when a new query object $q \in \mathbb{X} \setminus \mathbb{U}$ arrives, its proximity query consists in retrieving relevant objects from $\mathbb{U}$.

There are two basic queries, namely, *range* and *k-nearest neighbor* ones. The range query (q, r) retrieves all the elements in $\mathbb{U}$ within distance r to q. The k-nearest neighbor query $NN_k(q)$ retrieves the k elements in $\mathbb{U}$ that are closest to

* This work is partially funded by National Council of Science and Technology (CONACyT) of México, Universidad Michoacana de San Nicolás de Hidalgo, México, and Fondecyt grant 1131044, Chile.

q. Both queries can be trivially answered by exhaustively scanning the database, requiring n distance evaluations. However, as the distance function is assumed to be expensive to compute (think, for instance, when comparing two fingerprints), frequently the complexity of the search is defined in terms of the total number of distance evaluations performed, instead of using other indicators such as CPU or I/O time. Thus, the ultimate goal is to build an offline index that, hopefully, will accelerate the process of solving online queries.

2 Previous and Related Work

To solve this problem, a practical solution consists in building offline an index which is later used to solve online queries. Among the plethora of indices for metric space searching [3], the Permutation Based Index (PBI) [2] has shown an unbeatable performance. Let $\mathbb{P} \subset \mathbb{U}$ be a subset of permutants. Each element $u \in \mathbb{U}$ computes the distance towards all the permutants $p_1, \ldots, p_{|\mathbb{P}|} \in \mathbb{P}$. The PBI *does not store distances*. Instead, for each $u \in \mathbb{U}$, stores a sequence of permutant identifiers $\Pi_u = i_1, i_2, \ldots, i_{|\mathbb{P}|}$, called the permutation of u. Each permutation Π_u stores the identifiers in increasing order of distance, so $d(u, \mathbb{P}_{i_j}) \leq d(u, \mathbb{P}_{i_{j+1}})$. Permutants at the same distance take an arbitrary but consistent order. Thus, a simple implementation needs $n|\mathbb{P}|$ space. For the sake of saving space, we can compact several permutant identifiers in a single machine word. There are several improvements built on top of the basic PBI technique [1, 5, 7], however all of them are static indices.

The crux of the PBI is that two equal objects are associated to the same permutation, while similar objects are, hopefully, related to similar permutations. In this sense, when Π_u is similar to Π_q one expects that u is close to q. The similarity between the permutations can be measured by Kendall Tau K_τ, Spearman Footrule S_F, or Spearman Rho S_ρ metric [4], among others. As these three distances have similar retrieval performance [2], for simplicity we use S_F, defined as $S_F(\Pi_u, \Pi_q) = \sum_{j=[1,|\mathbb{P}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)|$, where $\Pi_u^{-1}(i_j)$ denotes the position of permutant p_{i_j} in the permutation Π_u. For example, if we have two permutations $\Pi_u = (42153)$ and $\Pi_v = (32154)$, then $S_F(\Pi_u, \Pi_v) = 8$.

Finally, at query time, we compute Π_q and compare it with all the permutations stored in the PBI. Next, $\mathbb{U}$ is traversed in increasing permutation dissimilarity. If we limit the number of distance computations, we obtain a probabilistic search algorithm that is able to find the right answer with some probability.

3 Our Approach

We propose a dynamic scheme for the PBI. That is, we grant the PBI the capability of inserting or deleting objects in the index while preserving the searching performance. Bestowing dynamism on the PBI allows us to manage real-world applications, where the whole dataset is unknown beforehand and objects are inserted or deleted as the retrieval system evolves. At the rest of the paper, we show how to do that.

3.1 Dynamic Permutants

A dynamic permutant based index has to support object insertions and deletions, while preserving the retrieving performance. We note that when we insert a new object into the index, a new permutant can also be added. So, each object in the index has a dynamic permutation. On the other hand, we need to support the case when we delete an object which is a permutant.

In order to deal with permutations that are continuously changing, for each object we split its permutation in buckets. This way, we can limit the scope of the changes. The number of buckets is a parameter we study experimentally. All these buckets make a valid permutation and the last bucket is considered *in process*. Formally, let B be the size of a bucket, then every object has a permutation divided in pieces of size B. That is, every object $u \in \mathbb{U}$ has a permutation Π_u divided in $\lceil \frac{|\Pi_u|}{B} \rceil = m$ pieces.

The main idea is processing small permutations of size B. Therefore, we will consider three sections: a list of bucket completed, a bucket of size B (which store the bucket in process), and a list of computed distances D of size B, to manage the distances for the bucket in process. Formally, for an object $u \in \mathbb{U}$, we have its complete permutation Π_u divided in $m = \lceil \frac{|\Pi_u|}{B} \rceil$ pieces. Therefore, $\Pi_u = \Pi_u^1, \Pi_u^2, \ldots, \Pi_u^m$. Particularly, Π_u^m is the permutation in process. We also need a small array D for the distances of the bucket Π_u^m.

Inserting an Object When inserting an object into the database we have two possibilities: it is a simple object or is also a new permutant. In first case, the object computes all the distances to the set of permutants and computes its permutation Π. The cost is $O(k)$ distances.

The interesting case is when an object v becomes a permutant (in this work, we chose the permutants at random). Firstly, v computes $d(u, v)$ for all $u \in \mathbb{U}$. Next, u modifies both its Π_u^m and its vector of distances D. In Algorithm 1 we show details of the insertion process as permutant.

Algorithm 1 InsertionAPermutant(p)

```
1: INPUT: Let p be the new permutant
2: Let U = {u_1, ..., u_n} be our database
3: for each u ∈ U do
4:     d1 = d(p, u)
5:     insert p in bucket Π_u^m and d1 at D
6:     Rebuild the small bucket Π_u^m
7: end for
```

Notice that when the bucket m is completed, it is transfered to the list of bucket completed and when a new permutant arrives, we use a new small bucket, and this is now the bucket Π_u^m.

For example: Let be u an element of the database, $B = 4$ and $\Pi_u = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, that is:

- List of bucket completed = $\{\Pi_u^1 = \{1,2,3,4\}, \Pi_u^2 = \{5,6,7,8\}\}$.
- Permutation in process = $\Pi_u^3 = \{9,10\}$
- Distances $D = [d(u,p_9), d(u,p_{10})]$

Comparing small permutations Using Spearman Footrule, we can compare two permutations in two ways. Let Π_u and Π_q be the permutations of an element u and a query q:

- If we consider use a sequential number for every permutant like the previous example, then we can compare all the permutation in a classic way, that is: $S_F(\Pi_u, \Pi_q) = \sum_{1 \leq i \leq k} \left| \Pi_u^{-1}(i) - \Pi_q^{-1}(i) \right|$.
- If we numerate each bucket separately, we just need numbers in $[1, B]$. So, we have $\Pi_u = \{1,2,3,4,1,2,3,4,1,2\}$, where $\{\Pi_u^1 = \{1,2,3,4\}, \Pi_u^2 = \{1,2,3,4\}\}$ and $\Pi_u^3 = \{1,2\}$. In this case we compute:

$$S_F(\Pi_u, \Pi_q) = \sum_{1 \leq i \leq m} \sum_{1 \leq j \leq B} \left| (\Pi_u^i)^{-1}(j) - (\Pi_q^i)^{-1}(j) \right|$$

  Notice, that we can get the same performance that the original technique. However, this alternative allows better compaction of the permutation.

Deleting Permutants In this case, we consider two element types. The first one is a simple object, which can be deleted without any consideration from the database. The second type is a permutant, that can also be deleted without modify the permutations of objects in $\mathbb{U}$, because the order in the rest of elements is conserved. A bucket with less than three permutants can be deleted, because is too short to help in the retrieving process.

Searching The search process is almost identical to the basic PBI one. The only consideration is that we need to compute the query permutation according to the buckets, and then we compare the permutations as we explain above.

4 Experiments

In this section we evaluate and compare the performance of our technique in different metric spaces, such as synthetic vectors on the unitary cube and NASA images. The experiments were run on an Intel Xeon workstation with 2.4 GHz CPU and 32 GB of RAM with Ubuntu server, running kernel 2.6.32-22.

4.1 Synthetic Databases

In these experiments, we used a synthetic database with vectors uniformly distributed on the unitary cube $[0,1]^D$, in order to control the dimensionality of the space. This also allows us to define some extra parameters. We use 80,000 points in different dimensions $D = 16$, 32, 64, and 128, under Euclidean distance.

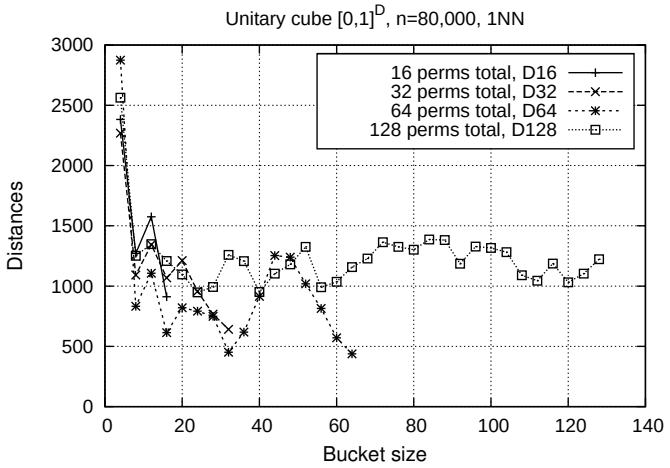

Fig. 1. Example of our technique. All points use the same amount of memory in the index. For example, line with $\times$ means 32 permutants and the first point has $B = 4$ that is $32/4 = 8$ buckets. The next point is $32/8$ we are using 4 buckets, and so on. The last point has $32/32 = 1$ bucket, that is the original idea. Notice that axe x represents the size of bucket.

4.2 Optimal Value of B

For this experiment, different values of B (bucket size) are plotted in Fig. 1 for different dimensions. Notice that if $B = m$ then we have the original permutation based index. In axe x, we change the size of bucket, the values start at 4, and increases in values of 4. In this case, we represent distances computed for 1NN in dimension 16, 32, 64, and 128. Notice that the last point is the value of the original technique. This plot shows that we can get a better performance with a dynamic technique. For example, in dimension 128 the original technique makes 1224 distances (1 completed bucket of $B = 128$) while using $B = 24$ (that is, 5 buckets), only 948 computations of distances are required for the same query, that is a 27% less distances.

4.3 NASA images

We use a set of 40,700 images from NASA, represented as 20-dimension feature vectors. For simplicity we compare the vectors with the Euclidean distance. This dataset is available at `www.dimacs.rutgers.edu/Challenges/Sixth/software.html`.

Notice that in this datase, our proposal keeps its performance. Fig 2 shows that after some B size our technique can improves the original idea. For example, 128 permutants with $B = 100$, or $B = 52$.

5 Conclusions

In this paper we present a technique to turn the permutation based index (PBI) into a dynamic one. That is, an index that support both insertions and deletions

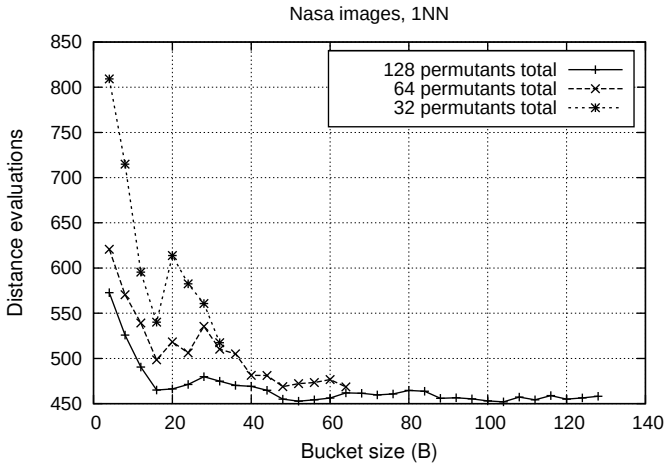

Fig. 2. Our approach keeps its performance on the real database of NASA images. All points use the same amount of memory.

of objects, while preserving the unbeatable performance of the original PBI. To do so, we process the complete permutation by parts.

As future work, we plan to test our technique in other metric spaces and research another alternatives to grant dynamism to the PBI strategy.

References

1. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: Proc. 3rd Intl. Conf. Scalable Information Systems. ICST (2008), article 28
2. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI) 30(9), 1647–1658 (2009)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (Sep 2001)
4. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
5. Figueroa, K., Paredes, R.: List of clustered permutations for proximity searching. In: Proc. 6th Intl. Conf. Similarity Search and Applications (SISAP 2013). pp. 50–58. LNCS 8199, Springer (2013)
6. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions Database Systems 28(4), 517–580 (2003)
7. Mohamed, H., Marchand-Maillet, S.: Quantized ranking for permutation-based indexing. In: Proc. 6th Intl. Conf. Similarity Search and Applications (SISAP 2013). pp. 103–114. LNCS 8199 (2013)
8. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search – The Metric Space Approach, Advances in Database System, vol. 32. Springer (2006)