

# Faster Proximity Searching in Metric Data<sup>\*</sup>

Edgar Chávez<sup>1</sup> and Karina Figueroa<sup>1,2</sup>

<sup>1</sup> Universidad Michoacana, México.

{elchavez, karina}@fismat.umich.mx

<sup>2</sup> DCC Universidad de Chile, Chile

**Abstract.** A number of problems in computer science can be solved efficiently with the so called *memory based* or *kernel* methods. Among this problems (relevant to the AI community) are multimedia indexing, clustering, non supervised learning and recommendation systems. The common ground to this problems is satisfying proximity queries with an abstract metric database.

In this paper we introduce a new technique for making practical indexes for metric range queries. This technique improves existing algorithms based on pivots and signatures, and introduces a new data structure, the *Fixed Queries Trie* to speedup metric range queries. The result is an  $O(n)$  construction time index, with query complexity  $O(n^\alpha)$ ,  $\alpha \leq 1$ . The indexing algorithm uses only a few bits of storage for each database element.

## 1 Introduction and Related Work

Proximity queries are those extensions of the exact searching where we want to retrieve objects from a database that are *close* to a given query object. The query object is not necessarily a database element. The concept can be formalized using the metric space model, where a distance function  $d(x, y)$  is defined for every site in a set  $\mathbb{X}$ . The distance function  $d$  has *metric* properties, i.e. it satisfies  $d(x, y) \geq 0$  (positiveness),  $d(x, y) = d(y, x)$  (symmetry),  $d(x, y) = 0$  iff  $x = y$  (strict positiveness), and the property allowing the existence of solutions better than brute-force for similarity queries:  $d(x, y) \leq d(x, z) + d(z, y)$  (triangle inequality).

The database is a set  $\mathbb{U} \subseteq \mathbb{X}$ , and we define the query element as  $q$ , an arbitrary element of  $\mathbb{X}$ . A similarity query involves additional information, besides  $q$ , and can be of two basic types of proximity queries:  $(q, r)_d = \{u \in \mathbb{U} : d(q, u) \leq r\}$ . *Metric Range* queries and  $nn_k(q)_d = \{u_i \in \mathbb{U} : \forall v \in \mathbb{U}, d(q, u_i) \leq d(q, v) \text{ and } |\{u_i\}| = k\}$ . *K nearest neighbor* query.

The problem have received a lot of attention in recent times, due to an increasing interest in indexing multimedia data coming from the web. For a detailed description of recent trends in the also called *distance based indexing* the reader should see [6]. If the objects are vectors (with coordinates), then a recent *kd-tree* improvement can be used [7], we are interested in the rather more general case of non-vectorial data.

---

<sup>\*</sup> Partially supported by CONACyT grant R-36911A and CYTED VII.19 RIBIDI

## 1.1 Related Work

There are two basic paradigms for distance based indexing, pivot-based algorithms and local partition algorithms, as described in [6], there the authors state the general idea for *local partition algorithms* which is to build a locality-preserving hierarchy, and then to map the hierarchy levels to a tree. *Pivoting algorithms*, on the other hand, are based on a mapping to a vector space using the distance to a set of distinguished sites in the metric space. Since our algorithm is also pivot-based we concentrate on this family of algorithms.

An abstract view of a pivot based algorithm is as follows. We select a set of  $l$  pivots  $\{p_1, \dots, p_l\}$ . At indexing time, for each database element  $a$ , we compute and store  $\Phi(a) = (d(a, p_1) \dots d(a, p_l))$ . At query time, for a query  $(q, r)_d$ , we compute  $\Phi(q) = (d(q, p_1) \dots d(q, p_l))$ . Now, we can discard every  $a \in \mathbb{U}$  such that, for some pivot  $p_i$ ,  $|d(q, p_i) - d(a, p_i)| > r$ , or which is the same, we discard every  $a$  such that

$$\max_{1 \leq i \leq l} |d(q, p_i) - d(a, p_i)| = L_\infty(\Phi(a), \Phi(q)) > r .$$

The underlying idea of pivot based algorithms is to project the original metric space into a vector space with a contractive mapping. We search in the new space with the same radius  $r$ , which guarantees that no answer will be missed. There is, however, the chance of selecting elements that should not be in the query outcome. These false positives are filtered using the original distance. The more pivots used, the more accurate is the mapping and the number of distance computations is closer to the number of elements in the query ball. The differences between each indexing algorithm is how is implemented the search in the mapped space. A naive solution is to search exhaustively in the mapped space[9], or to use a generic spatial access method like the R-tree[8]. Both of these solutions are acceptable if the dimension of the mapped space is maintained low, but can be worse if the dimension of the mapped space is high.

A natural choice for measuring the complexity of a proximity query in metric spaces is the number of distance computations, since this operation has leading complexity. The distance computations are in turn divided in inner and outer complexity, the later is the size of the candidate list and the former the distances to the pivots. A more realistic setup must include what is called *side computations* or the cost of searching in the tree (or data structure in general) to collect the list of candidates.

Some of the pivoting algorithms[2, 4, 10] have no control on the number of pivots used. If we want to have an effective control in reducing the size of the candidate list we must have arbitrarily taller trees, or a fixed number of pivots in the mapping. This is the approach of the Fixed Height Fixed Queries Trees (FHQT) [1], where the height of the tree is precisely the number of pivots, or the dimension of the mapped space. A serious drawback of the FHQT is the amount of memory used, because even if a branch in the tree collapses to a single path for each additional node we must save the distance and some additional node information.

Actual instances of datasets show that the optimum number of pivots (balancing the external and internal complexity) cannot be reached in practice. For example, when indexing a dictionary of 1,000,000 English words we can use as much as 512 pivots without increasing the number of distance computations.

The Fixed Queries Array [5] showed a dramatic improvement (lowering) on the size of the index induced by completely eliminating the tree, and working only with an ordered array representing the tree. In this approach every tree traversal can be simulated using binary search, hence adding a  $\log n$  penalty in the side computations but notably decreasing the *external complexity* or the size of the candidate list.

In this paper we will show a further improvement on the FQA algorithm, eliminating the logarithmic factor in the searching. This will be done by designing a way to manipulate whole computer words, instead of fetching bytes inside them.

## 1.2 Basic Terminology

**The set of pivots**  $\mathbb{K} = \{p_1, \dots, p_k\}$  is a subset of the metric space  $\mathbb{X}$ .

**A discretization rule** is an injective function  $\delta_p : \mathbb{R}^+ \times \mathbb{K} \rightarrow \{0, \dots, 2^{b_p} - 1\}$  mapping positive real numbers into  $2^{b_p}$  discrete values. The discretization rule depends on the particular pivot  $p$ . The preimage of the function defines a partition of  $\mathbb{R}^+$ . We assume  $\delta_p(r)$  will deliver a binary string of size  $b$ . Defining the discretization rule as pivot-depending allows to emphasize the importance of a particular pivot. For simplicity the same rule may be applied to all the pivots.

**A signature function for objects** will be a mapping  $\delta^* : \mathbb{X} \rightarrow \{0, 1\}^m$  with  $m = \sum_{i=1}^k b_{p_i}$ , given by the rule  $\delta^*(o) = \delta_{p_1}(d(o, p_1)) \cdots \delta_{p_k}(d(o, p_k))$ .

An infinite number of discretization rules may be defined for  $b$  bytes. Some of them will lead to better filtering methods and some of them will not filter at all. The definition of the discretization rule is not essential for the *correctness* of the filtering method, but for its efficiency. We can generalize the signature function for intervals, in the same way a function is extended from points to sets.

**A discretization rule for intervals** is denoted  $\delta_p([r_1, r_2]) = \{\delta_p(r) | r \in [r_1, r_2]\}$ . Formally  $\delta_p$  is defined for positive real numbers. This definition may be extended to the domain of real values, assuming  $\delta([r_1, r_2]) = \delta([0, r_2])$  if  $r_1 < 0$ .

**A signature function for queries** will be a function  $\delta^* : \mathbb{X} \times \mathbb{R}^+ \rightarrow 2^{\{0,1\}^m}$ , mapping queries  $(q, r)_d$  into a signature set. The signature of a query will be given by the following expression

$$\delta^*((q, r)_d) = \{\delta_{p_1}([d(q, p_1) - r, d(q, p_1) + r])\} \quad (1)$$

$$\{\delta_{p_2}([d(q, p_2) - r, d(q, p_2) + r])\} \cdots \quad (2)$$

$$\{\delta_{p_k}([d(q, p_k) - r, d(q, p_k) + r])\} \quad (3)$$

which is *any* ordered concatenation of the signature sets for the corresponding intervals in each pivot.

**Claim 1** If an object  $o \in \mathbb{X}$  satisfies a query  $(q, r)_d$ , then  $\delta^*(o) \in \delta^*((q, r)_d)$ .

To prove the above claim, it is enough to observe that  $\delta^*((q, r)_d)$  is an extension of  $\delta^*(\cdot)$ , and hence  $o \in (q, r)_d$  implies  $\delta^*(o) \in \delta^*((q, r)_d)$ .

**The candidate list** is the set  $[q, r]_d = \{o | \delta^*(o) \in \delta^*((q, r)_d)\}$ . Note that the candidate list is a superset of the query outcome, in other words,  $(q, r)_d \subseteq [q, r]_d$ .

**Remark 1** Computing the candidate list  $[q, r]_d$  implies only a constant number of distance evaluations, namely the number of pivots.

**Remark 2** The candidate list asymptotically approaches the size of the query outcome, as the number of pivots increases. This fact has been proved in [6] but it is easily verified, observing that  $(q, r)_d \subseteq [q, r]_d$  and that increasing the number of pivots never increases the size of  $[q, r]_d$ . For any finite sample  $\mathbb{U}$  of  $\mathbb{X}$ , if we take all the elements of  $\mathbb{U}$  as pivots, then  $(q, r)_d = [q, r]_d$  provided  $(q, r)_d \subseteq \mathbb{U}$ . This proves the assertion.

**Remark 3** Increasing the number of pivots increases the number of distance evaluations in computing  $\delta^*((q, r)_d)$ . This is trivially true, since computing  $\delta^*(\cdot)$  implies  $k$  distance computations. If we take as pivots all of  $\mathbb{U}$ , then we are changing one exhaustive search for another.

## 2 The Index

The goal of an indexing algorithm is to obtain the set  $(q, r)_d$  using as few distance computations as possible. The indexing algorithm consist in preprocessing the data set (a finite sample  $\mathbb{U}$  of the metric space  $\mathbb{X}$ ) to speed up the querying process. Let us define some additional notation to formally describe the problem and the proposed solution.

**The index of  $\mathbb{U}$**  denoted as  $\mathbb{U}^*$  is the set of all signatures of elements of  $\mathbb{U}$ .  $\mathbb{U}^* = \delta^*(\mathbb{U}) = \{\delta^*(o) | o \in \mathbb{U}\}$ .

**Remark 4**  $|\mathbb{U}^*| \leq |\mathbb{U}|$ . The repeated signatures are factored out and all the matching objects are allocated in the same bucket.

With this notation  $[q, r]_d$  may be computed as  $[q, r]_d = \delta^*(\mathbb{U}) \cap \delta^*((q, r)_d)$ . To satisfy a query we exhaustively search over  $[q, r]_d$  to discard false positives. The complete algorithm is described in figure 1.

We assume a blind use of  $\delta_p(\cdot)$ , the discretization function (optimizing  $\delta_p(\cdot)$  has been studied empirically in [5]), since we are interested in a fast computation of  $[q, r]_d$ . Nevertheless a fair choice is to divide the interval of minimum to maximum distances in  $2^b$  equally spaced slices.

### 2.1 Lookup Tables

The core of the searching problem with signatures consist in computing  $[q, r]_d$ . Observe that there are exponentially many elements in  $\delta^*((q, r)_d)$ . The signatures are obtained as an ordered concatenation of signatures for each pivot. If each pivot produces  $\nu_{p_i}$  signatures for its range, then we will have up to  $\prod_{i=1}^k \nu_{p_i}$  signatures. For example, for 32 pivots generating as few as 2 signatures for a query, the number of signatures is  $2^{32}$ . Instead we can split each signature in  $t$  computer words. A query signature may be represented as  $t$  arrays with at most

**Generic Metric Range Search**  
 $(\mathbb{X}, d)$  the metric space,  $\mathbb{U}$  is a database (a sample of  $\mathbb{X}$ ) of size  $n$   
 $\mathbb{K} = \{p_1, \dots, p_k\}$  is the set of pivots.  $k = |\mathbb{K}|$ ,  $b_{p_i}$  the number of  
bits for each pivot  $\delta(\cdot)$  the discretization rule,  $q \in \mathbb{X}$  the query object  
**Startup:**  $(\mathbb{U}, \mathbb{K}, \{b_{p_i}\}, \delta(\cdot))$   
1. Compute  $\mathbb{U}^*$   
**Search:**  $(q, r)$   
2. Compute  $[q, r]_d$   
for each  $o \in [q, r]_d$  do  
if  $d(q, o) \leq r$   
 $(q, r)_d \leftarrow o$   
fi  
od

**Fig. 1.** A generic indexing algorithm based on signatures. The index is  $\mathbb{U}^*$ , the objective is to compute  $(q, r)_d$  given  $q$  and  $r$ .

$2^w$  elements. A signature  $a_1 \dots a_m$   $a_i \in \{0, 1\}$  is splitted in  $t$  binary strings of size  $w$  of the form  $A_1 \dots A_t$  with  $A_i$  computer words. Each  $A_i$  is called a coordinate of a signature. A query signature will be represented as  $t$  sets of coordinates.

**A lookup table for query signatures** is an array  $L_j[]$  of  $2^w$  booleans,  $1 \leq j \leq t$ .  $L_j[i] = true$  if and only if  $i$  appears in the  $j$ -th set of coordinates. Note that computing  $L_j[]$  can be done in constant time (at most  $2^w$ ).

**Remark 5** We can decide if a particular signature is in the signature of the query by evaluating the boolean *AND* expression  $L_j[A_1] \otimes \dots \otimes L_j[A_t]$ , which takes at most  $t$  table fetches for an evaluation, on the average we may use fewer than  $t$  operations.

## 2.2 Sequential Scan

Remark 5 leads to a direct improvement of the sequential scan (FQS). We will compare the straight sequential scan with two better alternatives. If the query  $(q, r)_d$  has low selectivity the size of the candidate list  $[q, r]_d$  will be very large. An clever (sublinear) algorithm for finding the candidate list will be time consuming. Figure 2 illustrates the procedure to obtain the candidate list.

## 2.3 The Fixed Queries Array with Lookup Tables

The Fixed Queries Array (FQA) was proposed in [5] to increase the filtering power of a pivoting algorithm. Unlike sequential scan the approach is sublinear. The general idea of FQA is to keep an ordered array of the signatures. For the first pivot the query signature will be an interval in the array of signatures. This interval may be found using binary search with an appropriate mask. The

```

Compute  $[q, r]_d$  using Sequential Scan (FQS)
 $\mathbb{U}^*$  is the signature set  $n$ ,  $L_j[]$  is the lookup table for the query  $(q, r)_d$ 
1. Compute  $[q, r]_d$ 
   for each signature  $s = A_1 \cdots A_t \in \mathbb{U}^*$  do
     for  $j=1$  to  $t$  do
       if not  $L_j[A_j]$ 
         next  $s$ 
       fi
     od
   [math>q, r]_d \leftarrow \text{Object}(s)
od

```

**Fig. 2.** A sequential scan to compute  $[q, r]_d$ . The complexity is linear on the size of the database  $\mathbb{U}$ . If the query has low selectivity the sequential scan beats a clever (sublinear) approach.

process in repeated recursively for the successive pivots, which will appear as sub-intervals. The FQA may be improved using the lookup tables, allowing whole word comparisons (which is faster than fetching many times a few bits inside).

Given a query  $(q, r)_d$ , the **signature vector** for the  $i$ -th coordinate will be denoted as  $A_i[]$ . Figure 3 describes the recursive procedure to compute  $[q, r]_d$ . The FQA implementation with lookup tables is faster in practice than the plain algorithm in [5], and has the same theoretical  $O(\log(n))$  penalty.

## 2.4 The Fixed Queries Trie

Since we are using the signatures as strings, and want to find matching strings we have a plethora of data structures and algorithms from the string pattern matching community. Among them we selected a *trie* to index the signatures set  $\mathbb{U}^*$ . We will denote this trie as Fixed Queries Trie or *FQh*, the lowercase to distinguish between the trie and the tree (the Fixed Queries Tree) FQT described in [2].

Building the FhQt is even faster than building the FQA, the reason is that we don't need to use a sort, inserting each site's signature is done in time proportional to the signature length. The construction complexity, measured in distance computations, is then  $O(n)$  (as the FQA). Nevertheless the side computations for the construction is  $O(n)$  instead of  $O(n \log n)$ .

A trie is an  $m$ -ary tree with non-data nodes (routing only) and all the data at the leafs [3]. Searching for any string takes time proportional to the string size, independent of the database size. For our purposes the basic searching algorithm is modified to allow multiple string searching, and we make heavy use of the lookup table for routing. Our string set will be represented as a lookup table, this lead to a slightly different rule for following a node. Instead of following a

```

Compute  $[q, r]_d$  using FQA
 $\mathbb{U}^*$  is the signature set  $n$ , which is ordered by coordinates.
 $\{A_i[]\}$  are the signature vectors of query  $(q, r)_d$ 
Compute  $[q, r]_d$ 
1. function FQA(int j,  $\mathbb{V}^*$ )
    if j=t then
        for each  $A \in A_t[]$  do
             $[q, r]_d \leftarrow \text{Object}(\text{Select}(\mathbb{V}, A, t))$ 
        od
    return
    fi
    for each  $A \in A_j[]$  do
        FQA(j+1, Select( $\mathbb{V}, A, j$ ))
    od

```

**Fig. 3.** Computing  $[q, r]_d$  using a recursive binary search. The function  $\text{Select}(\mathbb{V}, A, j)$  obtains the signatures in  $\mathbb{V}$  whose  $j$ -th coordinate matches  $A$ . It may be implemented in logarithmic time if  $\mathbb{V}$  is ordered, which takes no extra memory, but a  $n \log(n)$  penalty for signature ordering.

node matching the  $i$ -th character of the string ( $i$  the trie level) we will follow the node if *any* coordinate in the set matches the node label. In other words, we will follow the node if the lookup table is *true* for the corresponding node and level.

The number of operations for this search will be proportional to the number of successful searches, or the number of leafs visited, times the length of the strings. In other words the complexity will be  $O(|[q, r]_d|t)$ . This represent a  $\log(n)$  factor, with respect to the FQA implementation.

The space requirements for the FQt may be smaller than that of FQA. The number of paths in the FQt is the number of elements in the signature array, but the trie will factor out matching coordinates and will use additional memory for pointers. The net result is that the trie will use *about* the same amount of memory, is built faster and uses less time for matching queries.

### 3 Experimental Results

The FHQt is exactly equivalent to the FQA and the FQS, from the point of view of selectivity, internal and external complexity. The same parametric decisions can be made using exactly the same analysis. We remark that the only difference is in the amount of extra work to find the candidate list. In that view we only will make an experimental validation of the algorithms improved with lookup tables, and hence will compare only the *side computations* and in a single experiment

```

Compute  $[q, r]_d$  using FQt
 $U^*$  is the signature set  $n$ , which is arranged in a trie.  $\{L_i[]\}$ 
are the lookup tables of the query  $(q, r)_d$  Each node of the trie is labeled
after a coordinate  $U$ 
Compute  $[q, r]_d$ 
1. function FQt(node  $U, i$ )
    if  $i \geq t$  then
        if  $L_t[U]$  then
             $[q, r]_d \leftarrow \text{Object}(U.\text{signature})$ 
            return
        fi
    for each  $U- > \text{node}$  do
        if  $L_i[U]$  then
            FQt( $U- > \text{node}, i + 1$ )
        fi
    od

```

**Fig. 4.** Computing  $[q, r]_d$  using a trie. The trie is faster than the FQA because the recursion is done without searching.

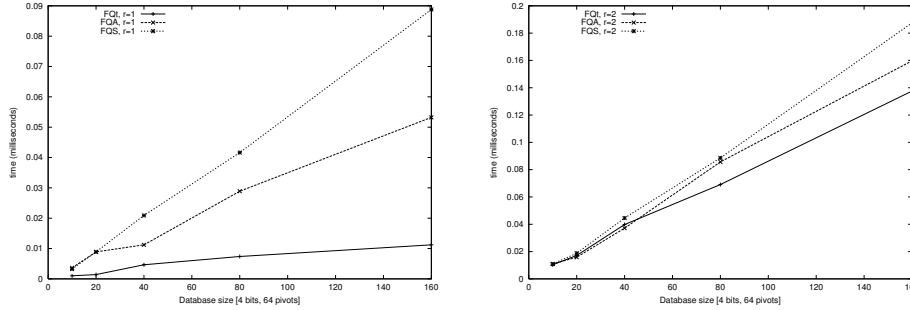
will compare the overall complexity. A more complete experimental study of the algorithm will be reported with the final version of this paper.

The three algorithms were implemented using lookup tables. The algorithms are only compared themselves, since it has been proved that FQA's may beat any indexing algorithm just by using more pivots. This is also true for FQt an FQS, hence the interest is between the three surviving alternatives. The previous implementations of the FQA (using bit masks) are beaten by the lookup table implementation by a large factor.

### 3.1 TREC

We indexed a vocabulary of the English dictionary, obtained from the TREC collection of the Wall Street Journal under the edit distance. This is a known difficult example of proximity searching, with high intrinsic dimension, as described in [6]. Figure 5 shows the overall time needed to compute a query of radius 1 and 2. Each point in the plots, for each experiment, were obtained by averaging 200 queries in the database. It can be noticed that as the query becomes less selective, the overall time of the algorithms FQA, FQS and FQS becomes practically the same. This is explained because the three algorithms have the same filtering power and the time to compute the candidate list makes a tiny difference. Figure 6 (top) shows the differences between the three filtering algorithms, at the left the overall time, and at the right only the time for filtering. Figure 6 (bottom) shows the performance of the indexes for medium and low selectivity queries (with radius one and two respectively). The time plotted

is only the filtering time, since the number of distance computations will be the same for all of them. The lookup tables are computed for the three algorithms, the construction is slightly more costly for fewer bits.



**Fig. 5.** The overall time needed to satisfy a queries of radius one (left) and two (right). The database is a sample of vocabulary of the TREC WSJ collection, under the edit distance.

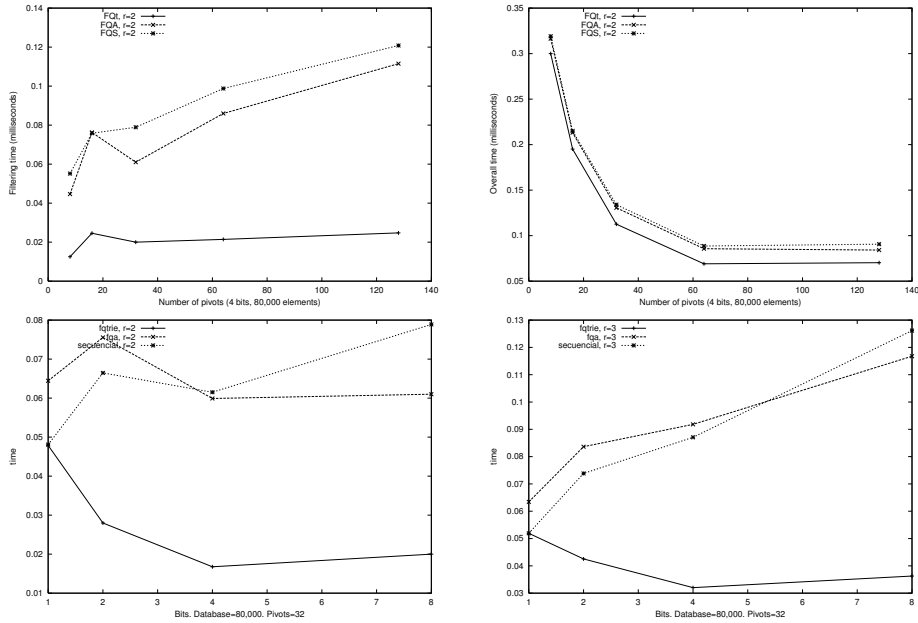
### 3.2 General metric spaces

In order to tested our algorithm in general metric spaces we use a synthetic set of random points in a  $k$ -dimensional vector space treated as a metric space, that is, we have not used the fact that the space has coordinates, but treated the points as abstract objects in an unknown metric space. The advantage of this choice is that it allows us to control the exact dimensionality we are working with, which is very difficult to do in general metric spaces. The points are uniformly distributed in the unitary cube, our tests use the  $L_2$  (Euclidean) distance, the dimension of the vector space is in the range  $4 \leq dim \leq 20$ , the database size is in the range  $10,000 \leq n \leq 160,000$ , and we perform range queries returning 0.01% of the total database size.

Figure 7 (top left) shows the performance of FQ<sub>t</sub> in different dimension, as we expected we need more time when dimension is high, even when we use different among of pivots. On top right we show distance evaluation needed to compute proximity searching. In that case we show when we use more pivots we reduce this evaluations.

On figure 7 we use different bits (bottom left) and we show how we can use less memory using FQ<sub>t</sub> without increment in excessive the overall time. Finally on bottom right we show the importance of our structure FQ<sub>t</sub>, because we can reduce the overall time with it.

Compare plots on figure 8, dimension 4 vs dimension 20, and notice how the dimension play a very important role, and we pay a huge cost in backtracking in high dimension unlike FQS.

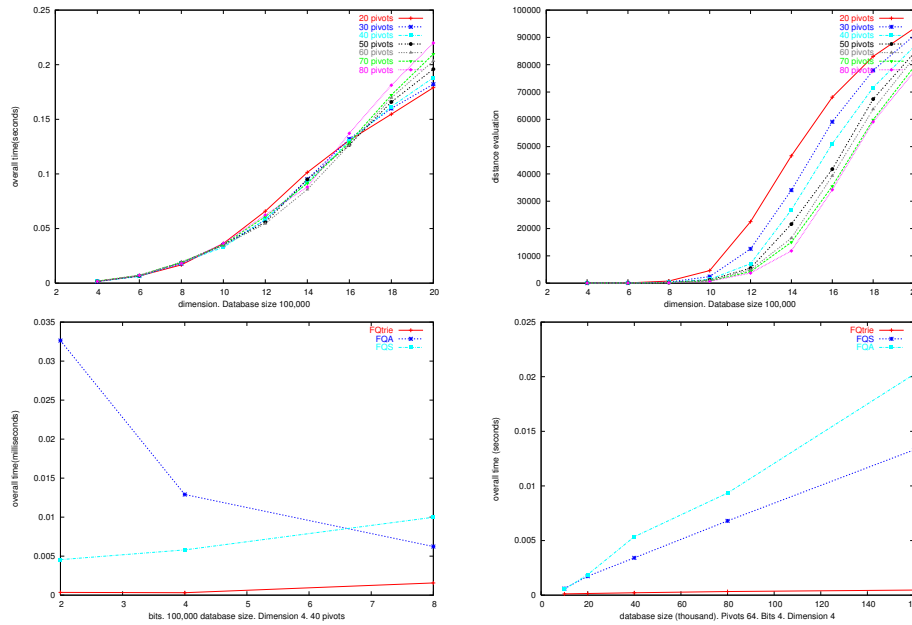


**Fig. 6.** (TOP) If we increase the number of pivots the time to obtain the candidate list is increased. The FQA and the FQS will use about the same time (about linear) for low selectivity queries, while the FQt increases very slowly. (BOTTOM) The role of the number of bits in the filtering time. For a fixed number of pivots, and a fixed database size.

## 4 Final Remarks

Pivoting algorithms have proved to be very efficient and sound for proximity searching. They have two control parameters, the number of pivots and the number of bits in the representation. Using lookup tables to implement the three alternatives presented in this paper gives a faster filtering stage. We also saw a strictly decreasing overall time for a particularly difficult example TREC dictionary and for general metric space. The experimental evidence and the analysis allows to favor the use of the FQt over the FQA or FQS in most realistic environments. In high dimension we prefer FQS because it avoids any backtracking unlike FQS and FQA.

Additionally the FQA doesn't accept insertion or deletions, unlike the FQt where insertion or deletions (of non-pivot objects) can be carried out trivially.

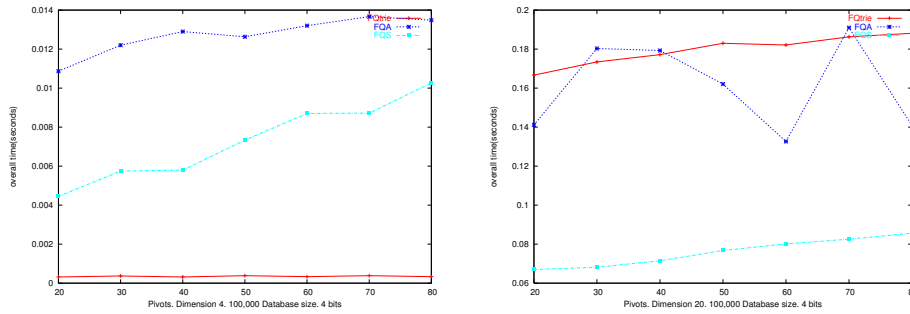


**Fig. 7.** (TOP) When we increment the number of pivots we reduce the distance evaluation but we increase lightly overall time. (BOTTOM) We show how we can reduce the use of memory with a little more time.

## 5 Bibliography

### References

1. R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.
2. R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
4. W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, 1973.
5. Edgar Chávez, Jos Luis Marroqun, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.
6. Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and Jos Luis Marroqun. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
7. Mark Derthick, James Harrison, Andrew Moore, and Steven Roth. Efficient multi-object dynamic query histograms. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 84–91. IEEE Press, 1999.



**Fig. 8.** The overall time needed to satisfy a proximity searching depends directly of dimension of data. (LEFT) dimension 4 and (RIGHT) dimension 20. On high dimension we avoid any backtracking with FQS.

8. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
9. L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.
10. P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 311–321, 1993.