Parallel Similarity Search based on the Dimensions Value Cardinalities of Image Descriptor Vectors

Dimitrios Rafailidis
Department of Informatics
Aristotle University, 54124 Thessaloniki, Greece
draf@csd.auth.gr

Yannis Manolopoulos
Department of Informatics
Aristotle University, 54124 Thessaloniki, Greece
manolopo@csd.auth.gr

ABSTRACT

In this paper, we propose a parallel similarity search strategy based on the dimensions value cardinalities, an inherit characteristic of image descriptor vectors. Our strategy has low preprocessing requirements by dividing the computational cost of the preprocessing steps into several machines and locating the descriptors with similar dimensions value cardinalities logically close. Additionally, an efficient parallel query processing algorithm is proposed, where the dimensions of image descriptors are prioritized in the searching strategy, assuming that dimensions of high value cardinalities have more discriminative power than the dimensions of low ones. In our experiments with publicly available datasets of 80 million and 1 billion images, we show that the proposed method outperforms state-of-the-art parallel similarity search strategies, in terms of preprocessing cost, search time and accuracy. Finally, we made our source code publicly available.

1. INTRODUCTION

Over the last two decades, several approximate similarity search strategies have been proposed, e.g. Dimensionality Reduction [3], Data Co-Reduction [5] and Hashing methods [4, 9], to accelerate the search process of the high-dimensional image vectors, a.k.a. descriptor vectors. Recently, the MSIDX method exploited a new key factor of the image descriptor vectors, namely the dimensions value cardinalities [11]. The dimensions value cardinalities represent the number of discrete values occurred in a specific dimension throughout a database of image descriptor vectors. The dimensions value cardinalities highly depend on the extraction strategy of the image descriptors. In particular, image descriptor vectors are extracted by forming histograms that describe the value distribution of each attribute, defining the characteristics of each descriptor, e.g. color, texture, shape etc. By applying values' normalization or quantization techniques to descriptor vectors, comparable histograms are produced for search and retrieval purposes [7]. However, the value cardinalities of each dimension vary significantly, depending on the descriptors' extraction strategy. For instance, high dimensional descriptors that come from the bag-of-words process with large number of centroids, tend to be sparse holding zeros in many dimensions. Therefore, these dimensions do not have more discriminative power than the rest of dimensions of high value cardinalities [11]. By considering the dimensions value cardinalities in the search strategy, MSIDX clearly outperformed competitive similarity search strategies.

However, MSIDX, Dimensionality Reduction, Data Reduction, Vantage Indexing and Hashing methods either employ a single CPU-core or work on a single machine, having thus memory limitations, expensive computational preprocessing costs and high search times for very large-scale databases of a few million or billion images. Much work has been done to parallelize similarity search in recognition that modern databases tend to contain billions of images. However, parallel similarity search strategies cannot efficiently query the high-dimensional vectors of images mainly for the following reasons: (a) complex index structures (M-Trees, R-Trees, KD-Trees, etc.) are required to be built either locally per machine or globally over all machines, increasing thus the preprocessing cost, (b) such strategies do not support the efficient dynamical insertion of new images, and finally, (c) they fail to preserve the visual nearest neighbors of sequential search efficiently in low search time (Section 2).

In this paper, we propose a parallel similarity search strategy based on image descriptors' dimensions value cardinalities. Our contribution is summarized as follows: **(C1):** The preprocessing cost and the memory requirements are low, by efficiently dividing the computational effort into several machines. In each machine, the preprocessing step locates the descriptors with similar dimensions value cardinalities logically close. This contrasts to the most related work of parallel similarity search strategies, which have the high preprocessing requirement of building complex index structures either locally per machine or globally over all the machines. **(C2):** New images are efficiently inserted into the databases by supporting dynamic real-time update. Based on our complexity analysis we mathematically prove that the proposed insertion algorithm depends on: (a) the descriptors' dimensionality, and (b) a small subset of descriptors that have similar dimensions value cardinalities. Thus, the computational cost of the insertion algorithm is preserved low. **(C3):** The query processing is efficiently divided into several machines by significantly speeding up the search process. In our par-

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695857

allel searching strategy, the dimensions of image descriptor vectors are prioritized, assuming that dimensions with high value cardinalities have more discriminative power. The proposed query processing achieves high accuracy in low search time.

The rest of the paper is organized as follows. Section 2 summarizes the related work of parallel similarity search strategies, whereas Section 3 introduces the preliminaries of similarity search based on dimensions value cardinalities. Then, in Section 4 we present the proposed Parallel similarity search strategy based on image descriptors' Dimensions Value Cardinalities (P-DVC). In Section 5 we evaluate the proposed P-DVC method and finally, Section 6 concludes the paper.

2. RELATED WORK

Several parallel similarity search strategies have been proposed. For instance, Aly et al. [1] used the Map-Reduce architecture to efficiently build and parallelize the KD-Tree index. A single KD-Tree is considered, where the top of the tree is located on a single root-node and the bottom part of the tree is divided into several machines, called leaf-nodes. Then, similarity search is performed into the leaf-nodes, whereas the root-node aggregates the results of the leaf-nodes and returns the top-k results. The disadvantage of the aforementioned similarity search strategy of [1] is that a high preprocessing cost is required to construct both global and local complex index structures per machine.

In [8], the FLANN library was extended, where authors examined the best performance between the priority search k-means trees, the multiple randomized KD-trees and the hierarchical clustering tree. FLANN performs an automatic configuration for the internal parameters of the examined methods (e.g. number of randomized trees, branching factor, number of k-means iterations) and use hyperparameters to control the relative importance of the build/preprocessing time and memory overhead in the overall cost. Finally, the FLANN library performs similarity search across multiple machines of a computer cluster, based on a Map-Reduce like algorithm and a Message Passing Interface (MPI) specification. There are several variations of the KD-trees such as trinary projection trees [12], which differ in the way they perform the space partitioning by using different partitioning functions. Despite the fact that tree-based methods achieve high accuracy, a significant cost is required to search the constructed trees in parallel.

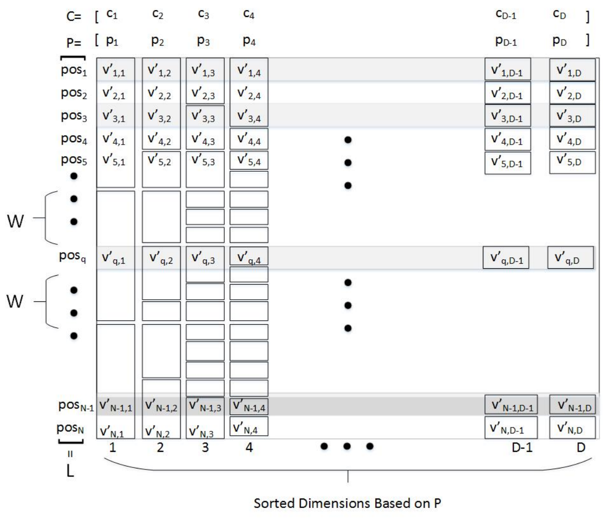
Additionally, inverted index algorithms have been widely used for similarity search due to their small memory cost. An inverted index initiates by clustering algorithms to build a codebook with K codewords, splitting the dataset into K lists. Then given a query and a desired candidate list T, the inverted index generates a list of T multimedia-points close to the query. In [2], an Inverted Multi-Index has been proposed to replace the standard quantization in an inverted index with product quantization, by splitting high dimensional vectors into more detailed dimension groups. The key idea is to use a product quantizer generating an exponentially large codebook at very low memory/time cost. The product quantization of the vectors is performed so that the K^2 lists correspond to all possible pairs of codewords, generating thus a more detailed subdivision of the search space, compared to the K lists that an inverted index generates. This way, the candidate lists produced by querying multi-indices were more accurate, compared to standard inverted indices. However, the size of the list of the T multimedia points plays a crucial role in the performance of Inverted Multi-Index, where large sizes of lists significantly increase the search time and the preprocessing cost of Inverted Multi-index. Moreover, Inverted Multi-Index does not work in parallel.

In contrast to the aforementioned similarity search methods, Norouzi et al. [10] proposed a multi-index hashing framework by avoiding to construct complex index structures. Binary codes from the database are indexed M times into M different hash tables, based on M disjoint binary substrings. For large-scale datasets, the substrings must be chosen so that the set of candidates is small and the storage requirements are low. The framework of [10] uses different hashing methods, such as LSH [4] or MLH [9]. Consequently, the framework preserves the search accuracy of the used hashing methods. Since the multi-index hashing framework performs exact similarity search in the hamming space, the framework's performance highly depends on the search accuracy of the used hashing methods. The big advantage of the framework is that a parallel implementation of multi-index hashing is straightforward, where each substring hash table is stored in a separate machine, by parallelizing thus the similarity search process. However, the preprocessing cost is high, since parallelization is not supported.

3. SIMILARITY SEARCH BASED ON DVC

The basic idea of similarity search based on Dimensions Value Cardinalities (DVC) according to [11] is: *to reorder the storage positions of images' descriptors according to value cardinalities of their dimensions, by performing a multiple sort algorithm, to increase the probability of having two similar images in storage positions that do not differ more than a specific global constant range, denoted by a parameter 2W*. Dimensions Value Cardinalities (DVC) are defined as the unique numbers that occur in the dimensions of the image descriptor vectors. Depending on the extraction strategy of the image descriptor vector there are three cases when calculating DVC: **Integer values**: In case of integer values, only the different values are considered and the total count is the value cardinality, denoted by c_j for the j-th dimension of the image descriptor, with $j \in \{1, \dots, D\}$. **Normalized real values**: In case of real values produced by normalization of previous integer values, the calculation strategy of the value cardinality c_j is the same with the case of integer values, due to the restricted value cardinality of the original integer-valued descriptor vector. **Real values**: In case that the extraction process of the descriptor generates real values, the calculation strategy of the value cardinality c_j is performed after limiting the decimal accuracy of the descriptor values, as in the case of integer values. However, in practice, the extracted descriptors have a limited decimal accuracy, usually between 4 and 6 decimals, due to space and computational restrictions [11]. In our experiments (Section 5) no additional value quantization was used in all evaluation datasets.

An overview of similarity search based on DVC is presented in Figure 1. Given a set $\mathcal{V}$ of N D-dimensional image descriptor vectors $\mathbf{v}_i \in \mathcal{V}$, $i = 1, \dots, N$, we firstly calculate the c_j values of the dimensions of the image descriptor vectors according the previous three cases. Then, we sort the dimensions of the image descriptors in a descending order,



C= [c1 c2 c3 c4 cD-1 cD]
P= [p1 p2 p3 p4 pD-1 pD]
pos1
pos2
pos3
pos4
pos5
W
posq
W
posN-1
posN
L
Sorted Dimensions Based on P

Figure 1: Similarity Search Based On DVC.

assuming that dimensions with high DVC (c_j values) are more discriminative [11]. For all sorted c_j values, we generate a respective priority vector $\mathbf{p}$ corresponding to the sorted dimensions, i.e. higher priority (p_j) for the j-th dimension based on the higher c_j value. This means that based on the priority vector $\mathbf{p}$ each value $v'_{i,j}$ [1] in Figure 1 corresponds to the highest DVC (c_j) value of the j-th dimension of the i-th descriptor vector $\mathbf{v}$. As depicted in Figure 1, based on the priority vector $\mathbf{p}$, the image descriptors are grouped as follows: firstly, descriptors are grouped based on the dimension with the highest DVC (c_j) value, i.e. those descriptors that have the same value in the first sorted dimension. For instance, in Figure 1 the first five image descriptors have the same $v'_{i,1}$ value ($i = 1 \ldots, 5$) in the first sorted dimension (j=1). After generating the first group based on the first sorted dimension, descriptors that are in the same group, are recursively grouped based on the second sorted dimension (j=2). For instance, in the previous example the group of the first 5 descriptors splits into two groups, i.e. the first 4 image descriptors have the same $v'_{i,2}$ value ($i = 1, \ldots, 4$) in the second sorted dimension, and the 5th image descriptor generates a second group based on $v'_{5,2}$. This procedure is repeated until the last sorted D-th dimension is reached. In case of groups with only one $v'_{i,j}$ value the procedure is not applied to these groups. After the image descriptors have been grouped and sorted recursively based on $\mathbf{p}$, the positions pos_i, of the N image descriptors are stored in the double linked list $\mathbf{L}$. In case of an external query q, the goal is to firstly identify the correct position pos_q in the double linked list $\mathbf{L}$ based on the priority vector $\mathbf{p}$. After identifying the correct position pos_q, $2W < N$ image descriptors in W previous and W next to position pos_q are retrieved to search for the top-$k < 2W$ most similar results.

4. THE PROPOSED P-DVC METHOD

4.1 Problem Formulation

Given (a) M machines, (b) the set $\mathcal{V}$ of the N D-dimensional image descriptor vectors, and (c) the descriptor vector $\mathbf{v}_q$ of a query image q [2], P-DVC supports the following functionalities: (a) parallel *preprocessing* of the N descriptor vectors to generate the global double linked list $\mathbf{L}$ with the logical sorted positions of all N descriptor vectors based on DVC, (b) *insertion* of the query $\mathbf{v}_q$ to global double linked list $\mathbf{L}$ in real-time, by computing the correct position pos_q, and (c) parallel *query processing* to retrieve the top-k similar results to query $\mathbf{v}_q$ in a result set $\mathcal{R}$ based on a predefined constant range $2W$, expressed as a percentage of the dataset N.

[1]The initial value $v_{i,j}$ has been reordered based on $\mathbf{p}$.

4.2 Preprocessing Algorithms

Role: To generate the global double linked list $\mathbf{L}$ with the logical sorted positions of all N descriptor vectors. The basic components are: (a) the M *Dimension Value Cardinality Extractors* (Section 4.2.1), (b) the *Priority Index* (Section 4.2.2), (c) the M *Image Sorters* and the *Global Image Sorter* (Section 4.2.3).
Input: $\mathcal{V}$, the set of the N D-dimensional image descriptor vectors $\mathbf{v}_i \in \mathcal{V}$, $i = 1 \ldots, N$.
Output: (1) $\mathbf{L}$, the global double linked list, (2) $\mathbf{p}$, the priority index vector of the D dimensions, (3) pk, the primary key to the dimension with the highest value cardinality.
Parameters: M, the number of assigned machines, i.e. M *Dimension Value Cardinality Extractors* and M *Image Sorters.*

4.2.1 *M Dimension Value Cardinality Extractors (Preprocessing Step 1)*

Role: To calculate the value cardinalities of the N D-dimensional image descriptors.
Input: $\mathcal{V}$, the set of the N D-dimensional image descriptor vectors.
Output: M different dimensions value cardinalities vectors $\mathbf{c}^{(m)}$, where $\mathbf{c}^{(m)}$ is the dimensions value cardinalities vector of the m-th *Dimension Value Cardinality Extractor*, with $m = 1, \ldots, M$.
Parameters: M, the number of *Dimension Value Cardinality Extractors.*

Algorithm Description

The outline of an m-th *Dimension Value Cardinality Extractor* is presented in Algorithm 1. M machines are assigned to calculate the dimensions value cardinalities. Let set $\mathcal{S}$ denote the image IDs, with $|\mathcal{S}| = N$. To avoid calculating all value cardinalities of the overall D dimensions of the N image descriptor vectors, upper $ub^{(m)}$ and lower $lb^{(m)}$ dimensions' bounds, $m = 1, \ldots, M$, are initialized for an m-th machine to define the dimensions' range that the m-th machine will process. Therefore, each machine is responsible for the computation of $\lceil \frac{D}{M} \rceil$ dimensions value cardinalities of the N image descriptor vectors. In line 1, $\lceil \frac{D}{M} \rceil$ distinct hash-maps[3] $\mathbf{HashMap}_h$, with $h = 1, \ldots, \lceil \frac{D}{M} \rceil$ are initialized to efficiently insert a new dimension's value or search for an already existing one. In line 2, the value cardinalities vector $\mathbf{c}^{(m)}$ is initialized. Therefore, in lines 3 to 12, N image descriptor vectors $\mathbf{v}_i$ are retrieved, with $i = 1, \ldots, N$. In line 4 the respective descriptor $\mathbf{v}_i \in \mathcal{V}$ is retrieved separately for each iteration, avoiding thus the bulk-loading of

[2]Here we assume that the extraction of image descriptor vectors is performed locally on each machine. Nevertheless, several works, such as [6], achieve to parallelize the descriptor vector extraction process.
[3]`http://en.wikipedia.org/wiki/Java\_collections\_framework`

all N descriptors. For each j-th dimension of a descriptor $\mathbf{v}_i$ within the range of the lower $lb^{(m)}$ and upper $ub^{(m)}$ dimensions' bounds, the existence of the corresponding $\mathbf{v}_{ij}$ value, $i = 1, \ldots, N$ and $j = lb^{(m)}, \ldots, ub^{(m)}$ is inspected to the corresponding hash-map $\mathbf{HashMap}_h$. If the $\mathbf{v}_{ij}$ value does not exist in the hash-map $\mathbf{HashMap}_h$, then it is inserted. Consequently, in lines 13-17, after the N descriptor vectors have been analyzed, the number of distinct values that are in each hash-map $\mathbf{HashMap}_h$ are assigned to the value cardinalities vector $\mathbf{c}^{(m)}$. Then, the value cardinalities vector $\mathbf{c}^{(m)}$ is returned for further analysis to the *Priority Index* at the next preprocessing step 2.

ALGORITHM 1: m-th Dimension Value Cardinality Extractor

```
   Input: $\mathcal{S}$: the set of IDs of the $N$ descriptor vectors
   $lb^{(m)}$: dimensions' lower bound for the $m$-th Dimension
   Value Cardinality Extractor
   $ub^{(m)}$: dimensions' upper bound for the $m$-th Dimension
   Value Cardinality Extractor
   Output: $\mathbf{c}^{(m)}$: the dimensions value cardinalities vector of
           the $m$-th machine
1  set $\lceil \frac{D}{M} \rceil$ hash-maps $\mathbf{HashMap}_h \leftarrow \emptyset$, $\forall h = 1, \ldots, \lceil \frac{D}{M} \rceil$;
2  set $\mathbf{c}^{(m)} \leftarrow \emptyset$
3  foreach $i \in \mathcal{S}$ do
4      $\mathbf{v}_i$ = retrieve the $i$-th descriptor vector;
5      set $h = 0$;
6      for $j = lb^{(m)}$ to $ub^{(m)}$ do
7          if $\mathbf{v}_{ij} \notin \mathbf{HashMap}_k$ then
8              insert the $\mathbf{v}_{ij}$ value into hash-map $\mathbf{HashMap}_h$;
9          end
10         $h = h + 1$;
11     end
12 end
13 set $h = 0$;
14 for $j = lb^{(m)}$ to $ub^{(m)}$ do
15     $\mathbf{c}_j^{(m)}$ = number of distinct values inserted in hash-map
       $\mathbf{HashMap}_h$;
16     $h = h + 1$;
17 end
18 return $\mathbf{c}^{(m)}$;
```

Complexity Analysis
For each dimension j of the N descriptor vectors, the value cardinality is calculated in $O(N)$, since the common-used hash-map structure requires $O(1)$ complexity for insertion. Accordingly, in a single machine the complexity for the dimensions value cardinalities' computation would be $O(N \cdot D)$. However, in our approach, since the dimensions of each vector are divided into M distinct machines-*Dimension Value Cardinality Extractors*, the complexity to compute the value cardinalities for all D dimensions is:

$$O(N \cdot \frac{D}{M}) \qquad (1)$$

4.2.2 Priority Index (Preprocessing Step 2)

Role: To calculate the priority index vector $\mathbf{p}$ of the D dimensions.
Input: M different dimensions value cardinalities vectors $\mathbf{c}^{(m)}$ (Algorithm 1).
Output: $\mathbf{p}$, the priority index vector, where $\mathbf{p}_j$, $j = 1, \ldots, D$, is the priority index of the j-th dimension.
Parameters: -

Algorithm Description
The outline of *Priority Index* is presented in Algorithm 2. In lines 1-7, the algorithm aggregates the values of the M different $\mathbf{c}^{(m)}$ vectors to generate the global value cardinalities vector $\mathbf{C}$, which contains the value cardinalities of all D dimensions of the N image descriptor vectors. In line 8, the global dimensions value cardinalities vector $\mathbf{C}$ is sorted in descending order, generating thus the final $\mathbf{C}'$ sorted vector. Since dimensions with high value cardinalities have more discriminative power, the higher the value cardinality of the j-th dimension is, the higher the respective priority index for the j-th dimension must be. Respectively, in line 9, the priority index vector $\mathbf{p}$ is generated based on the sorted value cardinalities vector $\mathbf{C}'$, providing high priority to the dimensions of high value cardinalities. This way, dimensions with high discriminative power are highly prioritized, reflecting to the dimensions of high value cardinality.

ALGORITHM 2: Priority Index

```
   Input: $M$ $\mathbf{c}^{(m)}$: the $M$ different dimensions value
          cardinalities vectors
   Output: $\mathbf{p}$: the priority index vector, where $p_j$ is the
           priority index of the $j$-th dimension, $\forall j = 1, \ldots, D$
1  for $j = 1$ to $D$ do
2      for $m = 1$ to $M$ do
3          if $\mathbf{c}_j^{(m)} \neq 0$ then
4              $\mathbf{C}_j = \mathbf{c}_j^{(m)}$;
5          end
6      end
7  end
8  sort dimensions value cardinalities vector $\mathbf{C}$ in descending
   order to generate the sorted vector $\mathbf{C}'$ ;
9  create the priority index $\mathbf{p}_j$ of dimension $j$ based on the
   sorted value cardinalities vector $\mathbf{C}'_j$, $\forall j = 1, \ldots, D$;
10 return $\mathbf{p}$;
```

Complexity Analysis
The complexity analysis of the *Priority Index* algorithm is analogous to the dimensions' number (D) of the image descriptor vectors and the number of machines (M) that are assigned to the *Dimension Value Cardinality Extractors* in the previous step. The aggregation of the M different $\mathbf{c}^{(m)}$ vectors requires a $O(D \cdot M)$ complexity. The cardinalities sort procedure and, respectively, the priority index creation is performed using the quick sort algorithm in $O(D \cdot \log D)$ cost. Summarizing, the total complexity of the *Priority Index* algorithm is:

$$O(M \cdot D) + O(D \cdot \log D) \qquad (2)$$

4.2.3 M Image Sorters and The Global Image Sorter (Preprocessing Step 3)

Role: To generate the global double linked list $\mathbf{L}$ with the logical sorted positions of all N descriptor vectors. The descriptor vectors' sorting process is divided into M *Image Sorters* and then the *Global Image Sorter* component performs the final sorting in $\mathbf{L}$ with a single machine.
Input: (1) $\mathbf{p}$, the priority index vector (Algorithm 2), (2) $\mathcal{V}$, the set of the N image descriptors.
Output: (1) $\mathbf{L}$, the global double linked list, (2) pk, the primary key to the dimension with the highest value cardinality.
Parameters: M, the number of *Image Sorters.*

Algorithm Description
M Image Sorters: Each *Image Sorter* performs the dimensions' sorting to $\lceil \frac{N}{M} \rceil$ descriptor vectors. Therefore, a set $\mathcal{V}^{(m)}$ of image descriptor vectors constitute the input of the m-th *Image Sorter*, $m = 1, \ldots, M$. Initially, each *Image Sorter* reorders the descriptor vectors' dimensions based on the priority index $\mathbf{p}$ (Algorithm 2). Then, the descriptor vectors in $\mathcal{V}^{(m)}$ are sorted in descending order by performing the quicksort algorithm based on the comparative Algorithm 3, which produces a new set $\mathcal{V'}^{(m)}$. Then, a double linked list $\mathbf{L}^{(m)}$ is computed, where $\mathbf{L}_i^{(m)}$ denotes the ID of the image descriptor vector, which is allocated in position pos_i in list $\mathbf{L}^{(m)}$, $i = 1, \cdots, \lceil \frac{N}{M} \rceil$.

Global Image Sorter: The computed M distinct double linked lists $\mathbf{L}^{(m)}$ are retrieved by the *Global Image Sorter*. Based on Algorithm 3, *Global Image Sorter* compares the M distinct $\mathbf{L}_1^{(m)}$ descriptor vectors, where $\mathbf{L}_1^{(m)}$ is the ID of the descriptor vector with the highest position in $\mathbf{L}^{(m)}$. According to the comparison, the respective ID of the descriptor vector is inserted into the first available slot of a global double linked list $\mathbf{L}$. Then, the inserted ID to the global list $\mathbf{L}$ is removed from the respective list $\mathbf{L}^{(m)}$. This process is performed recursively until the global double linked list $\mathbf{L}$ contains the positions of all N descriptor vectors.

Finally, in the current step of the preprocessing phase, we set a primary key *pk* to the dimension with the highest value cardinality, that is the dimension with the highest priority index based on vector $\mathbf{p}$. As we will explain in the following section, the reason for setting a primary key *pk* is to efficiently retrieve the minimum number of image descriptor vectors in the insertion step of P-DVC.

ALGORITHM 3: Compare Image Descriptors

```
   Input: $\mathbf{v}_a$,$\mathbf{v}_b$: $D$-dimensional descriptor vectors
   Output: 1 (if $\mathbf{v}_a > \mathbf{v}_b$), -1 if( $\mathbf{v}_a < \mathbf{v}_b$), 0 if($\mathbf{v}_a = \mathbf{v}_b$)
 1 $\mathbf{v}_{aj}$ = value of $\mathbf{v}_a$ in dimension $j = 1, ..., D$;
 2 $\mathbf{v}_{bj}$ = value of $\mathbf{v}_b$ in dimension $j = 1, ..., D$;
 3 for $j = 1$ to $D$ do
 4     if $\mathbf{v}_{aj} > \mathbf{v}_{bj}$ then
 5         return 1;
 6     end
 7     if $\mathbf{v}_{aj} < \mathbf{v}_{bj}$ then
 8         return -1;
 9     end
10 end
11 return 0;
```

Complexity Analysis
The complexity of the *M Image Sorters* and the *Global Image Sorter* is $O(D \cdot \frac{N}{M} \cdot \log \frac{N}{M})$. Summarizing, the total complexity of *Preprocessing* is the aggregation of the complexities of: (a) the *M Dimension Value Cardinality Extractors*, (b) the *Priority Index*, and (c) the *M Images Sorters* and the *Global Image Sorter*. Therefore, based on Eqs. (1) and (2), the total *Preprocessing* cost is:

$$O(N \cdot \frac{D}{M}) + O(M \cdot D) + O(D \cdot \log D) + O(D \cdot \frac{N}{M} \cdot \log \frac{N}{M}) \quad (3)$$

4.3 Insertion Algorithm

Role: To insert a new image descriptor vector $\mathbf{v}_q$ by updating the global double linked list $\mathbf{L}$.

Input: (1) $\mathbf{v}_q$, the new image descriptor vector, (2) $\mathbf{p}$, the priority index vector (preprocessing step 2), (3) $\mathbf{L}$, the global double linked list (preprocessing step 3), (4) *pk*, the primary key to the dimension with the highest value cardinality (preprocessing step 3).

Output: pos_q, the logical position of the new image descriptor vector $\mathbf{v}_q$ in the updated global double linked list $\mathbf{L}$.

Parameters: -

Algorithm Description
The insertion algorithm is presented in Algorithm 4. In line 1, a set $\mathcal{V}_{pk}$ is generated ($|\mathcal{V}_{pk}| \ll N$), which consists of the descriptor vectors with primary key *pk* equal to $\mathbf{v}_{qp_1}$. Value $\mathbf{v}_{qp_1}$ is the value of the highest priority dimension of the new descriptor $\mathbf{v}_q$. In line 2, the set $\mathcal{V}_{pk}$ of descriptor vectors are sorted in descending order based on the logical positions in the global double linked list $\mathbf{L}$. Based on Algorithm 3, the descriptor vector $\mathbf{v}_q$ is compared with descriptor $\mathbf{v}_{hp}$, i.e. the descriptor vector in $\mathcal{V}_{pk}$ of the highest position in $\mathbf{L}$, to determine the logical position pos_q (lines 3-7). Finally, in line 8, the new image q is inserted into the allocated position pos_q and the global double linked list $\mathbf{L}$ is updated, respectively.

ALGORITHM 4: Insertion Algorithm

```
   Input: $\mathbf{v}_q$: the $D$-dimensional descriptor vector of image $q$
   $\mathbf{p}$: the priority index vector of the dimensions
   $\mathbf{L}$: the double linked list of the logical positions
   $pk$: the primary key to the dimension with the highest value
   cardinality
   Output: $pos_q$: the position of image $q$ in the updated
           double linked list $\mathbf{L}$
 1 generate set $\mathcal{V}_{pk}$ of the descriptor vectors with primary key
   $pk$ equal to value $\mathbf{v}_{qp_1}$;
 2 sort $\mathbf{v} \in \mathcal{V}_{pk}$ in descending order based on the logical
   positions in $\mathbf{L}$ and retrieve the first descriptor $\mathbf{v}_{hp} \in \mathcal{V}_{pk}$ of
   the highest position;
 3 if (Compare Images $\mathbf{v}_{hp}$ and $\mathbf{v}_q$) = 1 then
 4     set $pos_q$ after the position of $\mathbf{v}_{hp}$ in $\mathbf{L}$;
 5 else
 6     set $pos_q$ prior to the position of $\mathbf{v}_{hp}$ in $\mathbf{L}$;
 7 end
 8 insert descriptor vector $\mathbf{v}_q$;
 9 update the double linked list $\mathbf{L}$;
10 return $pos_q$;
```

Complexity Analysis
The complexity of the insertion algorithm is highly correlated to the size of the subset $\mathcal{V}_{pk}$ and the dimensionality D of the image descriptor vectors. To perform the sorting of descriptors in $\mathcal{V}_{pk}$ based on their logical locations in $\mathbf{L}$, the quicksort algorithm requires a $O(|\mathcal{V}_{pk}| \cdot \log |\mathcal{V}_{pk}|)$ cost. Then, since the comparison is always performed between the descriptor vector $\mathbf{v}_q$ and the descriptor vector $\mathbf{v}_{hp} \in \mathcal{V}_{pk}$ of the highest logical position in $\mathbf{L}$, a $O(D)$ complexity is required. Therefore, the total complexity of the insertion algorithm is:

$$O(|\mathcal{V}_{pk}| \cdot \log |\mathcal{V}_{pk}|) + O(D) \quad (4)$$

4.4 Query Processing Algorithm

Role: To retrieve the top-k results of the query image descriptor vector $\mathbf{v}_q$.

Input: (1) $\mathbf{v}_q$, the new image descriptor vector, (2) $\mathbf{p}$, the priority index vector (preprocessing step 2), (3) $\mathbf{L}$, the global

double linked list (preprocessing step 3), (4) *pk*, the primary key to the dimension with the highest value cardinality (preprocessing step 3).
Output: $\mathcal{R}$, the result set of the top-k results.
Parameters: (1) M, the number of *Image Comparators*, (2) $2W$, the search radius.

Algorithm Description
Firstly, the insertion algorithm is utilized to calculate the logical position pos_q in the double linked list $\mathbf{L}$. Then, the query processing algorithm compares the query image descriptor vector $\mathbf{v}_q$ to $2W$ descriptor vectors $\mathbf{v}_i$, with $i = 1, \ldots, 2W$, where W is a user defined search radius. The similarity search is divided into M machines to retrieve the $2W$ candidate image IDs for query q. W is a constant range denoting the number of the candidate images prior and next to pos_q in the double linked list $\mathbf{L}$. Then, two descriptor vectors are always reserved into one *Image Comparator*, i.e. $\mathbf{v}_q$ and $\mathbf{v}_i$, $i = 1, \ldots, 2W$. The output of each *Image Comparator* is the respective distance between the query descriptor vector $\mathbf{v}_q$ and the candidate image descriptor vector $\mathbf{v}_i$. Each calculated distance is retrieved and stored into a min-Heap[4] $\mathcal{H}$. After all the $2W$ comparisons have been performed, the top-k image IDs similar to query q form the result set $\mathcal{R}$. The algorithm of P-DVC's *Query Processing* step is presented in Algorithm 5.

ALGORITHM 5: Query Processing Algorithm

```
   Input: $\mathbf{v}_q$: the $D$-dimensional descriptor vector of the query
          image $o_q$
   $\mathbf{p}$: the $D$-dimensional priority index vector
   $k$: the number of top-$k$ results
   $W$: the search radius
   $pk$: the primary key to the dimension with the highest value
   cardinality
   Output: the top-$k$ results set $\mathcal{R}$ of the query image $q$
 1 set $\mathcal{R} \leftarrow \emptyset$;
 2 set min-heap $\mathcal{H} \leftarrow \emptyset$
 3 $pos_q$ = Insert($\mathbf{v}_q$) based on Algorithm 4;
 4 generate $\mathcal{V}_{2W}$ by retrieving $W$ descriptors prior to $pos_q$
   and $W$ descriptors next to $pos_q$;
 5 for $iter$ = 1 to $2 \cdot W$ do
 6     compute the distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ from an available
       $m$-th Image Comparator, with $\mathbf{v}_{iter} \in \mathcal{V}_{2W}$;
 7     insert the ID of image descriptor vector $\mathbf{v}_{iter}$ and the
       distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ into $\mathcal{H}$;
 8 end
 9 for $iter$ = 1 to $k$ do
10     retrieve and remove the image $t$ located on top of $\mathcal{H}$;
11     $\mathcal{R} = \mathcal{R} \bigcup t$;
12 end
13 return the top-$k$ results set $\mathcal{R}$;
```

In line 3, the query descriptor vector $\mathbf{v}_q$ is inserted based on Algorithm 4, returning the respective position pos_q in $\mathbf{L}$. Then, $\mathbf{v}_q$ is stored and the linked list $\mathbf{L}$ is updated respectively. To increase the probability of retrieving the top-k most similar images, a specific constant range is used, denoted by the search radius W, with $W \gg k$. In line 4, a set $\mathcal{V}_{2W}$ of descriptor vectors is generated. The $2W$ descriptor vectors are the W previous and W next to the position pos_q in $\mathbf{L}$. In lines 5 to 8, $\forall\ \mathbf{v}_{iter} \in \mathcal{V}_{2W}$, the respective distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ is calculated by an m-th available *Image Comparator*, based on a predefined distance measure $d(\cdot)$, e.g. L1, L2, squared-L1, etc. Since the *Image Comparator* contains the most essential process of the query processing algorithm, on each m-th *Image Comparator* multi-core instances are assigned to parallelize each distance calculation in T threads. In doing so, the computational cost of each distance calculation is further reduced, achieving thus lower similarity search time. Then, each calculated distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ candidate image ID is inserted into a minimum-heap $\mathcal{H}$ (line 7). Then, in lines 9-12, after all $2W$ distance measurements have been completed, the top-k image IDs are extracted by the top of the minimum-heap $\mathcal{H}$. The final set of top-k IDs constitutes the result set $\mathcal{R}$.

[4]http://en.wikipedia.org/wiki/Min-max_heap

Complexity Analysis
The complexity of the query processing algorithm is calculated as the aggregation of: (a) the allocation of the storage position pos_q based on the *Insertion* Algorithm 4, (b) the construction of the minimum-heap structure $\mathcal{H}$, and (c) the generation the result set $\mathcal{R}$. Based on Eq. (4), the complexity of allocating position pos_q is $O(|\mathcal{V}_{pk}| \cdot \log |\mathcal{V}_{pk}|) + O(D)$. Moreover, the distance calculations of the $2W$ closest images' positions are performed by the corresponding *M Distance Comparators*. Therefore, the complexity of the distance calculations is $O(\frac{2 \cdot W}{M \cdot T} \cdot D)$, where T is the number of threads that are used to parallelize each distance calculation in each *Image Comparator*. Additionally, since $2W$ image IDs are inserted into the heap $\mathcal{H}$, the insertion of each image into the heap is performed in $O(\log 2 \cdot W)$. However, over the execution of the *Query Processing* algorithm, k images are preserved into the heap $\mathcal{H}$, i.e. the already examined images that have the lowest distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ to the query and thus, the complexity of the insertion of each image into the heap is reduced from $O(\log 2W)$ to $O(\log k)$. In doing so, the complexity of the distance calculations is $O(\frac{2 \cdot W}{M \cdot T} \cdot D \cdot \log k)$. Finally, the retrieval of the k image IDs from the heap $\mathcal{H}$ has a $O(k)$ complexity. Summarizing, the final complexity of the *Query Processing* algorithm is:

$$O(|\mathcal{V}_{pk}| \cdot \log |\mathcal{V}_{pk}|) + O(D) + O(\frac{2 \cdot W}{M \cdot T} \cdot D) + O(k) \quad (5)$$

5. EXPERIMENTS

5.1 Datasets And Settings

In our experiments we evaluate P-DVC on the Tiny Image collection [5] of N=80M images of GIST descriptors of D=348-dimensions (**GIST-80M-348d**) and N=1B images of SIFT descriptors of D=128-dimensions of the TEXMEX collection[6] (**SIFT-1B-128d**).

Following the evaluation protocol of [8, 10, 11], we performed 1,000 test queries, where the search accuracy mAP for each query is measured according to the following ratio:

$$mAP = \frac{|\mathcal{R}_{seq} \cap \mathcal{R}_{ind}|}{k}$$

where $\mathcal{R}_{seq}$ is the set of the top-k results (Euclidean neighbors) retrieved by the sequential search based on the Euclidean distance, whereas $\mathcal{R}_{ind}$ is the set of the top-k results retrieved by the examined similarity search method. The final performance of each method is measured by the mAP

[5]http://horatio.cs.nyu.edu/mit/tiny/data/index.html
[6]http://corpus-texmex.irisa.fr/

variable, which is defined as the average search accuracy of the 1,000 performed queries.

Two are the most crucial parameters in P-DVC, the number of machines M=(2,8) and the search radius $2W$= (0.1%, 1%, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, 50%), expressed as a percentage of the N total dataset size.

According to Eq. (4), the insertion time of a new image descriptor $\mathbf{v}_q$ depends on the dimensionality D of the descriptors and the size of the small subset $|\mathcal{V}_{pk}| \ll N$. Therefore, the insertion time does not depend on the number of M machines and the dataset size N. Insertion times of P-DVC are 4.217 and 8.27 msec for **SIFT-1B-128d** and **GIST-80M-348d**, respectively.

The proposed P-DVC framework was implemented in Java using the Windows Azure Emulator [7] under the SDK 2.1. All experiments were conducted on a machine of 3.3 GHz CPU with 18 GB main memory, running Windows Server 2008 R2 Enterprise Edition 64-bit. In our experiments, we allocated 1 CPU and 2 GB RAM, corresponding into one emulated machine, and in the case of each *Image Comparator* we provided the maximum amount of T=8 parallel threads. Our source code is publicly available [8].

5.2 Results

In our experiments, we compared P-DVC against the following parallel similarity search strategies:

(1) Multi Index Hashing (MIH) [10]: MIH [9] achieves parallel processing by using multiple hash tables. For making fair comparison, we evaluate the performance of P-DVC of M machines against MIH of M hash tables. Following the experimental evaluation of MIH, we used the hashing methods of LSH and MLH [9]. For both hashing methods in the MIH framework, we varied the number of bits by 8, 16, 32 and 64. The reason for limiting the number of bits variation to 64 bits is that the implementation of MIH caused memory overflows for higher number of bits for both LSH and MLH methods. The number of the M hash tables were varied as the number of the M machines in P-DVC.

(2) KD-Trees [1]: Following the evaluation strategy of [1], we used the parameter of backtracking steps, denoting the fixed budget for doing backtracking steps for every dimension which is shared among all the KD-Trees searched for this dimension. The backtracking steps for KD-Trees [10] were varied in (0.5%, 1%, 3%, 5%, 7%, 9%, 10%, 11%, 12%), expressed as a percentage of the total dataset size N. The reason for limiting the backtracking steps is that the search time is exponentially increased for large number of backtracking steps. For the KD-Trees method, the number of the M machines in P-DVC is equal to the number of the M KD-Trees that are required to be built.

(3) FLANN[8]: Following the evaluation strategy of [8] on large-scale datasets we used the randomized KD-trees algorithm using M=(2,8) machines to search in parallel over the constructed KD-trees, by varying the branching factor as in [8, 12]. In the preprocessing step, the publicly available implementation of FLANN [11] by default uses all available machines.

[7]`http://www.windowsazure.com/en-us/`

[8]`http://delab.csd.auth.gr/~draf/pdvc.zip`

[9]`https://github.com/norouzi/mih`

[10]Following [1] we parallelized the implementation of the Caltech Large Scale Image Search Toolbox.

[11]`https://github.com/mariusmuja/flann`

(4) Inverted Multi-Index [2]: According to the experimental evaluation in [2], for Inverted Multi-Index [12] we varied the list length in the range of (2^8,2^{12},2^{16}) using a codebook with size 2^{14}. Inverted Multi-Index does not work in parallel; however it was also evaluated on the common **SIFT-1B-128d** and **GIST-80M-348d** evaluation datasets.

In Figure 2, we evaluate the performance of P-DVC against the competitive strategies. In both datasets, P-DVC outperforms the competitive similarity search strategies for the same settings, i.e. the number of M machines is equal to the number of M hash tables. This is achieved by exploiting the dimensions value cardinalities and efficiently splitting the computational effort of the query processing algorithm (Algorithm 5). This contrasts to the competitive similarity search strategies which do not reach high mAP in low search time. For instance, despite the fact that the search time of MIH is low, MIH preserves the limited mAP accuracy of the hashing methods (LSH and MLH). The most competitive method is FLANN; however, it searches the complex structure of KD-tree in parallel, increasing thus the search time.

In Table 1, we present the preprocessing cost of the examined methods. The MIH framework performs parallel processing only in the case of online similarity search by using multiple hash tables. However, the number of hash tables does not affect the offline preprocessing cost. Therefore, for the MIH framework we report the preprocessing time requirements for all number of bits variations. For the KD-Trees method, the number of machines is equal to the number of the KD-Trees that are required to be built. As aforementioned, the publicly available implementation of FLANN by default uses all available machines in the preprocessing step and thus we report only one preprocessing cost. Inverted Multi-Index does not support parallel preprocessing. The proposed P-DVC strategy has the less preprocessing requirements. This happens because the preprocessing algorithms of P-DVC (Section 4.2) avoid using complex index structures and function efficiently in parallel. The main differences between P-DVC and the competitive methods are that the MIH's preprocessing step does not function in parallel, whereas the KD-Tree method and FLANN require a significant preprocessing cost to build the complex structures of multiple KD-Trees and randomized KD-trees, respectively. Meanwhile, Inverted Multi-Index has heavy computational costs, by increasing the list length, without paying off in terms of search time and mAP in the online search.

6. CONCLUSIONS

In this paper, we presented a Parallel similarity search strategy based on the image descriptors' Dimensions Value Cardinalities, called P-DVC. By considering DVC in our similarity search strategy, we avoid constructing complex index structures (KD-trees, Inverted Multi-Index, Hash Tables) and thus we preserve the preprocessing cost low. Moreover, since no complex structures have been built, searching in these structures is avoided and thus the search time is significantly reduced, by prioritizing dimensions with high DVC in our parallel searching strategy. In doing so, the proposed method preserves the mAP accuracy high in low

[12]`https://github.com/arbabenko/MultiIndex`

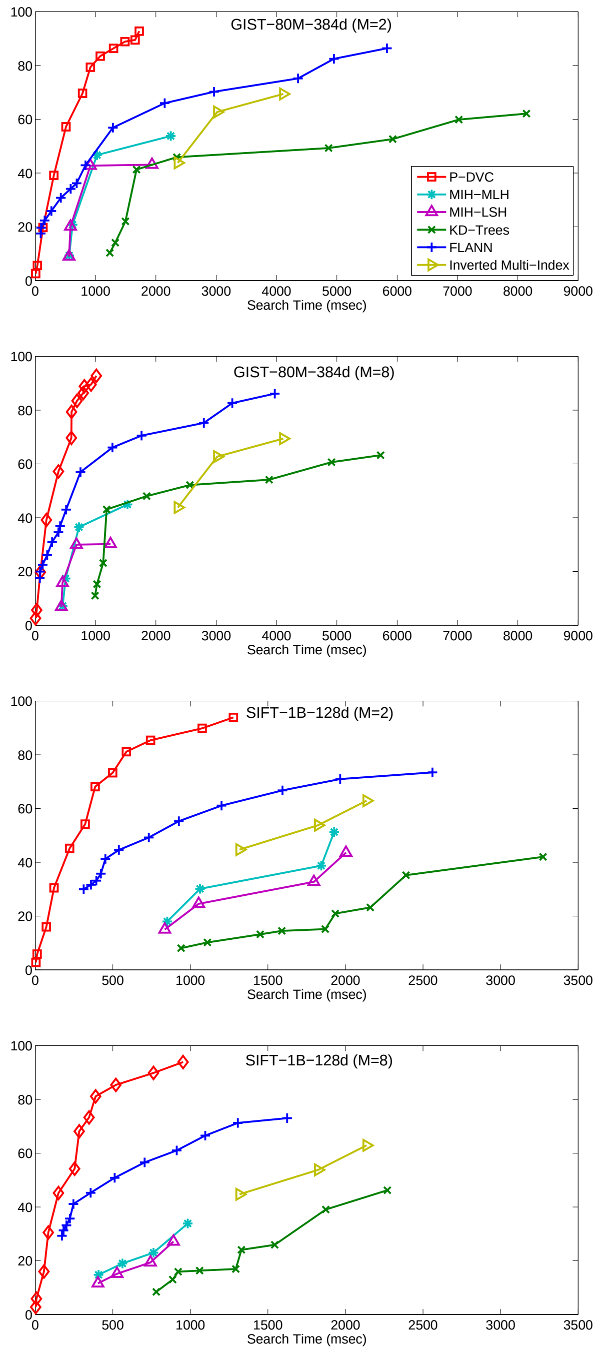


Figure 2: Performance Evaluation.

search time. An interesting topic for future work is to evaluate the proposed P-DVC framework on a real cloud infrastructure, including the network's latency and overhead.

7. REFERENCES

[1] M. Aly, M. E. Munich, and P. Perona. Distributed kd-trees for ultra large scale object recognition. In *Proceedings 22nd British Machine Vision Conference (BMVC)*, pages 1–11, 2011.

[2] A. Babenko and V. S. Lempitsky. The inverted multi-index. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3069–3076, 2012.

[3] H. Cheng, K. A. Hua, K. Vu, and D. Liu. Semi-supervised dimensionality reduction in image feature space. In *Proceedings 23rd ACM Symposium on Applied Computing (SAC)*, pages 1207–1211, 2008.

[4] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings 25th International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[5] Z. Huang, H. T. Shen, J. Liu, and X. Zhou. Effective data co-reduction for multimedia similarity search. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 1021–1032, 2011.

[6] K. Jarrah and L. Guan. Content-based image retrieval via distributed databases. In *Proceedings 7th ACM International Conference on Image and Video Retrieval (CIVR)*, pages 389–394, 2008.

[7] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[8] M. Muja and D. G. Lowe. Scalable nearest neighbour algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, to appear.

[9] M. Norouzi and D. J. Fleet. Minimal loss hashing for compact binary codes. In *Proceedings 28th International Conference on Machine Learning (ICML)*, pages 353–360, 2011.

[10] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(6):1107–1119, 2014.

[11] E. Tiakas, D. Rafailidis, A. Dimou, and P. Daras. MSIDX: multi-sort indexing for efficient content-based image search and retrieval. *IEEE Transactions on Multimedia*, 15(6):1415–1430, 2013.

[12] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X. Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(2):388–403, 2014.

Table 1: Preprocessing Cost (sec)

	SIFT-1B-128d	**GIST-80M-384d**
P-DVC (M=2)	25.07	27.68
P-DVC (M=8)	**20.07**	**21.99**
KD-Trees (M=2)	948	972
KD-Trees (M=8)	797.04	830.94
FLANN	618	918
Inverted Multi-Index (2^8)	94,740	69,660
Inverted Multi-Index (2^{12})	116,640	83,880
Inverted Multi-Index (2^{16})	134,940	106,620
MIH-LSH (8-bit)	55.97	69.57
MIH-LSH (16-bit)	61.76	78.15
MIH-LSH (32-bit)	87.18	102.78
MIH-LSH (64-bit)	120.84	156.80
MIH-MLH (8-bit)	50.35	67.2
MIH-MLH (16-bit)	63.82	81.30
MIH-MLH (32-bit)	91.07	124.67
MIH-MLH (64-bit)	114.92	167.32