Similarity Search Over The Cloud Based On Image Descriptors' Dimensions Value Cardinalities

STEFANOS ANTARIS and DIMITRIOS RAFAILIDIS, Aristotle University of Thessaloniki

In recognition that in modern applications billions of images are stored into distributed databases in different logical or physical locations, we propose a similarity search strategy over the cloud based on the dimensions value cardinalities of image descriptors. Our strategy has low preprocessing requirements by dividing the computational cost of the preprocessing steps into several nodes over the cloud and locating the descriptors with similar dimensions value cardinalities logically close. New images are inserted into the distributed databases over the cloud efficiently, by supporting dynamical update in real-time. The proposed insertion algorithm has low computational complexity, depending exclusively on the dimensionality of descriptors and a small subset of descriptors with similar dimensions value cardinalities. Finally, an efficient 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. The computation effort of the query processing algorithm is divided into several nodes over the cloud infrastructure. In our experiments with seven publicly available datasets of image descriptors, we show that the proposed similarity search strategy outperforms competitive methods of single node, parallel and cloud-based architectures, in terms of preprocessing cost, search time and accuracy.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Measurement, Experimentation

Additional Key Words and Phrases: Content-based image retrieval, large-scale similarity search, distributed databases, multimedia cloud computing

ACM Reference Format:
Antaris, S. and Rafailidis, D., Similarity Search Over The Cloud Based On Dimensions Value Cardinalities. *ACM Trans. Multimedia Comput. Commun. Appl.* V, N, Article A (January YYYY), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

With the ubiquitous presence of capture devices such as phones, digital cameras, and camcorders, the Internet has been transformed into a major channel for multimedia content delivery [Tian et al. 2010]. Searching for images in large databases based on visual similarity is a main challenge for the multimedia community [Zhang and Rui 2013]. Over the last two decades, several similarity search strategies were proposed to retrieve the visual nearest neighbors of images' high dimensional descriptors. Initially, researchers were focused on exact similarity search strategies, capable of preserving the visual nearest neighbors of sequential search. Nevertheless, exact similarity search strategies often face the Dimensionality Curse problem, increasing thus the computational cost of retrieving the visual nearest neighbours. Therefore, several

Author's address: S. Antaris and D. Rafailidis, School of Informatics, Aristotle University Campus, 54124, Thessaloniki, Greece; email: {santaris,draf}@csd.auth.gr

DOI: http://dx.doi.org/10.1145/0000000.0000000

approximate similarity search strategies were proposed, performing an approximation of the visual nearest neighbors of sequential search, by highly reducing the computational cost. For example, approximate similarity search strategies are the Dimensionality Reduction [Cheng et al. 2008; Liang et al. 2010; Wang and Binbin 2011], Data Co-Reduction [Huang et al. 2011], Vantage Indexing [Bozkaya and Ozsoyoglu 1999; Fu et al. 2000; Van Leuken and Veltkamp 2011] and Hashing methods [Gionis et al. 1999; Jegou et al. 2011; Heo et al. 2012; Liu et al. 2014]. Recently, the MSIDX method [Tiakas et al. 2013] exploited a new key factor of the image descriptor vectors, namely the dimensions value cardinalities. The dimensions value cardinalities represent the number of discrete values occurred in a specific dimension throughout a dataset of image descriptor vectors. The dimensions value cardinalities highly depend on the extraction strategy of the image descriptors. The ability of the extraction strategy to model correctly image characteristics, such as color, texture, illumination and resolution variations, play the most important role on the final performance of the image descriptor. Image descriptor vectors are extracted by forming histograms that describe the value distribution of each attribute, defining the aforementioned characteristics of each descriptor. By applying values' normalization or quantization techniques [Lowe 2004] to descriptor vectors, comparable histograms are produced for search and retrieval process. 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. By considering the dimensions value cardinalities in the search strategy, MSIDX clearly outperformed competitive similarity search strategies. Further details about similarity search based on dimensions value cardinalities can be found in the online Appendix A.

However, MSIDX, Dimensionality Reduction, Data Reduction, Vantage Indexing and Hashing methods are of single node architecture, having thus memory limitations, expensive computational preprocessing costs and high search times for very large-scale datasets of a few million or billion images. Therefore, similarity search strategies emerge the imperative need of parallelization, in order to highly reduce the computational cost. Much work has been done to parallelize database structures in recognition that modern databases tend to contain billions of images. Cloud computing is an emerging technology aimed at providing various computing and storage services over the Internet [Zhu et al. 2011]. For multimedia applications, there are strong demands for cloud computing services because of the significant amount of required computations. Moreover, in modern applications images are stored into distributed databases in different logical or physical locations, a case which is handled by distributed or cloud-based approaches. However, similarity search strategies in distributed databases or over cloud infrastructures cannot efficiently query the high-dimensional data 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 node or globally over the cloud, increasing thus the preprocessing cost; (b) such strategies do not support the efficient dynamical insertion of new images; 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 large-scale similarity search strategy over the cloud based on image descriptors' dimensions value cardinalities. Our contribution is summarized as follows,

— The preprocessing cost and the memory requirements are low, by dividing the computational effort into several nodes over the cloud, efficiently. In each node, the pre-

processing step locates the descriptors with similar dimensions value cardinalities logically close. This comes in contrast to the most related work of similarity search strategies in distributed databases or over cloud infrastructures, which have the high preprocessing requirement of building complex index structures either locally per node or globally over the cloud.

— New images are inserted into the distributed databases efficiently, by supporting dynamical update in real-time. The proposed insertion algorithm over the cloud is of low computational complexity, since it depends on (a) the descriptors' dimensionality and (b) a small subset of descriptors that have similar dimensions value cardinalities.

— The query processing steps are divided into several nodes over the cloud infrastructure, efficiently. Therefore, the high-computational cost is significantly reduced. In the searching strategy, the dimensions of image descriptor vectors are prioritized, assuming that dimensions with high value cardinalaties have more discriminative power. In doing so, the query processing over the cloud achieves high accuracy in low search time.

In our experiments with seven publicly available datasets of image descriptor vectors, we show that the proposed similarity search strategy outperforms competitive methods of single node, parallel and cloud-based architectures in terms of preprocessing cost, search time and accuracy. The remainder of the paper is organized as follows. Section 2 summarizes the related work of similarity search strategies in distributed databases and over cloud infrastructures. Then, in Section 3 we present the proposed similarity search strategy over the Cloud based on image descriptors' Dimensions Value Cardinalities (CDVC). In Section 4 we evaluate the proposed CDVC framework against state-of-the-art similarity search strategies and Section 5 concludes the paper.

2. RELATED WORK

Several exact similarity search strategies in distributed databases were proposed, such as M-Chord by Batko et al. [2008]. To further speed up the search process several approximate distributed similarity search algorithms were introduced [Novak et al. 2008; 2012; Zhu et al. 2012; Vlachou et al. 2012]. All the aforementioned similarity search strategies in distributed databases rely on an underlying structured P2P network and focus on the parallelism of the query processing. Each node of the P2P network has a replication instance of the multimedia data and a local indexer. Therefore, the replication of the multimedia data in several nodes hinders the insertion of new multimedia objects, since they need to be inserted into multiple nodes. Additionally, the aforementioned similarity search strategies have high computational cost to build a global indexer from local ones. Meanwhile, several similarity search strategies over cloud infrastructures were proposed. The main differences between distributed and cloud-based approaches are the different strategies that are followed for handling the problems of (a) scalability & elasticity; and (b) data consistency & integrity. Distributed approaches do not handle the incremental growth of resources due to the many changes that are required in both the hardware and software. Also, in a distributed architecture, several replication techniques are used to ensure data consistency into the distributed memory, by adding thus an intensive computation cost. On the contrary, cloud infrastructures focus on maximizing the effectiveness of the shared resources by dynamically reallocating the nodes based on the service's load capacity. Additionally, cloud infrastructures provide their own data management system, where distinct nodes access common data, ensuring thus the data consistency and integrity. For instance, Wang et al. [2010] propose RT-CAN, a distributed R-tree indexer to support multidimensional range queries over cloud infrastructures. RT-CAN integrates

the CAN-based routing protocol [Ratnasamy et al. 2001] and each node uses its own R-tree structure to index the data that are locally stored. Nevertheless, using an R-Tree structure locally, RT-CAN is not suitable for the high dimensional descriptors of multimedia, since a significant preprocessing cost is required.

Additionally, multiple randomized KD-trees were proposed by Silpa-Anan and Hartley [2008] to generate space partitions to accelerate similarity search. In the query stage, the search is performed simultaneously in the multiple trees, through a shared priority queue. Muja and Lowe [2009] perform a wide range of comparisons showing that multiple randomized KD trees are suitable for similarity search, by also performing an automatic configuration method for the internal parameters of the multiple randomized KD-trees method. Aly et al. [2011] used the MapReduce architecture to efficiently build and distribute the KD-Tree indexer (distributed KD-trees) for millions of images. 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 nodes, called leaf-nodes. Then, similarity search is performed into the leaf-nodes and 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 distributed KD-trees over cloud infrastructures is that a high preprocessing cost is required to construct both global and local complex index structures per node. Muja and Lowe [2014] proposed FLANN, where they 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 in distributed databases 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 [Jia et al. 2010; Wang et al. 2014], which differ in the way they perform the space partitioning, by using different partitioning functions. All the aforementioned tree-based methods have high preprocessing cost to build the complex structures in large-scale high dimensional datasets. Despite the fact that tree-based methods achieve high accuracy, a significant cost is required to search the constructed trees in parallel and to retrieve the most similar results.

Several nearest neighbor graph techniques [Wang et al. 2012; Wang et al. 2013] have been proposed for similarity search. Nearest neighbor graph techniques build an approximate nearest neighbor graph structure in which multimedia-points are vertices and edges connect each multimedia-point to its nearest neighbors. Afterwards, similarity search is performed by guiding the search in the neighborhood graph. However, the nearest neighbor graph methods suffer from an expensive construction of the graph structure [Muja and Lowe 2014; Wang et al. 2014].

The 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 and then given a query and a desired candidate list T the inverted index generates a list of T multimedia-points close to the query. Babenko and Lempitsky [2012] proposed the Inverted Multi-Index which replaces the standard quantization in an inverted index with product quantization, splitting high dimensional vectors into more detailed dimension groups. The key idea is to use a product quantizer which generates 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 a

inverted index generates. In doing so, 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. Also, Norouzi and Fleet [2013] proposed the Cartesian k-means method for similarity search with a parameterization of the cluster centers such that number of centers is super-linear in the number of the involving parameters. In particular, given a query, the distances between the query and a subset of centers is pre-computed and stored in a query lookup table, which is used to compare all candidate points to the query. The number of the clusters play an essential role in the performance of the Cartesian k-means algorithm, since large number of clusters improve the search accuracy while significantly increasing the search time and the preprocessing cost. Moreover, both the Inverted Multi-Index and Cartesian k-means do not work in parallel in the preprocessing and search steps.

In contrast to the aforementioned similarity search methods, Norouzi et al. [2014] 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 Norouzi et al. [2014] uses different hashing methods such as LSH [Gionis et al. 1999] or MLH [Norouzi and Fleet 2011]. 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 distributed implementation of multi-index hashing is straightforward, in which each substring hash table is stored on a separate node, by parallelizing thus the similarity search process. However, the preprocessing cost is high, since parallelization is not supported.

3. THE PROPOSED CDVC FRAMEWORK

The architecture of the proposed CDVC framework is presented in Figure 1. CDVC supports the following functionalities: (a) parallel *preprocessing* of datasets' descriptor vectors for storage management in distributed databases over the cloud, (b) *insertion* of new images in real-time and (c) efficient *query processing* for searching the top-k similar results to a query image q.

The *preprocessing* phase is initialized by the *CDVC Scheduler* to analyze the N images, stored in the distributed databases over the cloud. A *Cloud Storager Interface* acts as a middleware to connect the distributed databases with the CDVC components. In particular, this interface permits the components to interact with the distributed databases, through the *Cloud Database Server*, in order to retrieve and store the available image data. After the preprocessing phase has finished, users are able to access the proposed CDVC framework using personal devices, such as computers, laptops or smartphones. Users pose image queries to the CDVC framework to retrieve top-k results from the distributed databases. This is achieved within the *insertion* and *query processing* phases. The search process starts with the *CDVC Scheduler*, which receives the D-dimensional descriptor vector $\mathbf{v}_q$, of query image q [1]. The *CDVC Scheduler* component orchestrates the components of the *insertion* and the *query processing*

[1]In this paper we assume that extraction of descriptor vectors of images is performed locally and not over the cloud. Nevertheless, several works, such as the work of Jarrah and Guan [2008], achieve to parallelize the descriptor vector extraction process.

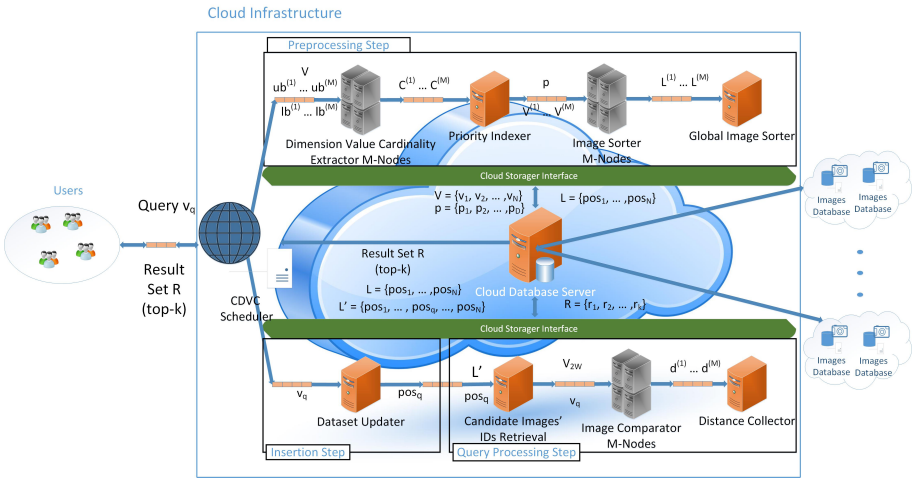


Fig. 1. CDVC framework.

phases to initially insert the new image query q and furthermore, to return the top-k similar image results in set $\mathcal{R}$.

According to the architecture of the CDVC framework in Figure 1, the *preprocessing* phase takes as input a set $\mathcal{V}$ of the N D-dimensional descriptor which are stored in the distributed databases via the *CDVC Scheduler* and the *Cloud Database Server*. The goal of the *preprocessing* phase is to generate a global double linked list $\mathbf{L}$ with the logical sorted positions of all N descriptor vectors based on DVC, assuming that dimensions with high DVC are more discriminative. The *preprocessing* phase consists of three steps, including four basic components of the CDVC framework: (1) the set of M *Dimension Value Cardinality Extractor* nodes (preprocessing step 1); (2) the *Priority Indexer* (preprocessing step 2); (3) the set of M *Image Sorter* nodes (preprocessing step 3); and (4) the *Global Image Sorter* (preprocessing step 3). The role of each *Dimension Value Cardinality Extractor* node is to calculate the value cardinalities of a subset of dimensions of the N D-dimensional image descriptors based on M different predefined lower $lb^{(m)}$ and upper $ub^{(m)}$ dimensions' bounds, with $m = 1, \ldots, M$. The M different dimension value cardinality vectors $\mathbf{C}^{(m)}$ are provided as input to the *Priority Indexer* component which is responsible to merge the M different dimension value cardinality vectors $\mathbf{C}^{(m)}$ into a global dimension value cardinality vector $\mathbf{C}$. The output of the *Priority Indexer* is a priority indexer vector $\mathbf{p}$ of the dimensions, which is calculated by sorting the global dimension value cardinality vector $\mathbf{C}$ in a descending order, assuming that dimensions with high DVC are more discriminative. Then, based on the generated priority index vector $\mathbf{p}$, each *Image Sorter* node retrieves a subset of the image descriptor vectors $\mathcal{V}^{(m)}$ to perform the logical sorting, generating thus M different double linked lists $\mathbf{L}^{(m)}$, which contain the logical sorted positions of the image descriptors in $\mathcal{V}^{(m)}$. The M different double linked lists $\mathbf{L}^{(m)}$ are merged into a global double linked list $\mathbf{L}$ by the *Global Image Sorter*. After the *preprocessing* step terminates, the double linked list $\mathbf{L}$ and the priority indexer $\mathbf{p}$ are both stored into the *Cloud Database Server* for the *insertion* and the *query processing* steps. Given a query $\mathbf{v}_q$ the *CDVC Scheduler* initiates firstly the *insertion* and then the *query processing* step, where the goal is to retrieve the top-k similar image results in a set $\mathcal{R}$. The goal of the *insertion* step is to identify the position pos_q in the double linked list $\mathbf{L}$ based on the priority index vector $\mathbf{p}$ and furthermore to insert the query in the position pos_q by updating the double linked list from $\mathbf{L}$ to $\mathbf{L}'$ via the *Dataset Updater* component.

Afterwards, given the updated double linked list $\mathbf{L}'$ and the position pos_q of the image query q, the *Candidates Images' IDs Retrieval* generates a set $\mathcal{V}_{2W}$ of image descriptors which are located in the update double linked list $\mathbf{L}'$, in W previous and W next to the position pos_q. The set $\mathcal{V}_{2W}$ of image descriptor vectors and the query image descriptor vector $\mathbf{v_q}$ are retrieved by the set of M *Image Comparator* nodes which are responsible for calculating the respective distances between $\mathbf{v_q}$ and $\mathcal{V}_{2W}$. The M different sets $d^{(m)}$ of the calculated distances are collected by the *Distance Collector*, which generates the top-k similar image results in set $\mathcal{R}$. Finally, $\mathcal{R}$ is returned to the users via the *Cloud Database Server* and the *CDVC Scheduler*. In the remainder of the section, we describe the basic components of *CDVC*, accompanied by the implementation details of the algorithms and the respective complexity analysis.

3.1. Preprocessing Over The Cloud

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 set of M *Dimension Value Cardinality Extractor* nodes (Section 3.1.1), (b) the *Priority Indexer* (Section 3.1.2), (c) the set of M *Image Sorter* nodes and the *Global Image Sorter* (Section 3.1.3).
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 nodes for parallelization over the cloud infrastructure, i.e. M *Dimension Value Cardinality Extractor* and M *Image Sorter* nodes.

3.1.1. Dimension Value Cardinality Extractor Nodes - 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* node
Parameters: M, the number of *Dimension Value Cardinality Extractor* nodes.
Algorithm Description
The outline of a *Dimension Value Cardinality Extractor* node is presented in Algorithm 1. M nodes are allocated 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 node to define the dimensions' range that the m-th node will process. Therefore, each node 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[2] $\mathbf{HashMap}_h$, 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 by the *Cloud Database Server*, 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 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 to 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

[2]http://en.wikipedia.org/wiki/Java_collections_framework

value cardinalities vector $\mathbf{c}^{(m)}$ is returned for further analysis to the *Priority Indexer* component (preprocessing step 2).

ALGORITHM 1: Dimension Value Cardinality Extractor Node

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 node
$ub^{(m)}$: dimensions' upper bound for the m-th Dimension Value Cardinality Extractor node
Output: $\mathbf{c}^{(m)}$: the dimensions value cardinalities vector of the m-th node

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 node architecture 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 nodes, the complexity to compute the value cardinalities for all D dimensions is

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

3.1.2. Priority Indexer - 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 Indexer* 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. In doing so, dimensions with high discriminative power are highly prioritized, reflecting to the dimensions of high value cardinality.

ALGORITHM 2: Priority Indexer

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 Indexer* algorithm is analogous to the dimensions' number (D) of the image descriptor vectors and the number of nodes (M) that are assigned to the *Dimension Value Cardinality Extractor* component. 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 $O(D \cdot logD)$ cost. Summarizing, the total complexity of the *Priority Indexer* algorithm is

$$O(M \cdot D) + O(D \cdot logD) \tag{2}$$

3.1.3. Image Sorter Nodes and 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 Sorter* nodes and then the *Global Image Sorter* component performs the final sorting in $\mathbf{L}$ with a single node.
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 the *Image Sorter* nodes.
Algorithm Description
Image Sorter nodes: Each *Image Sorter* node applies 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* node, $m = 1, \ldots, M$. Initially, the *Image Sorter* node 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 quick

sort 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 to 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 $\mathbf{L}^{(m)}$. The aforementioned process is performed recursively until the global double linked list $\mathbf{L}$ contains the positions of all N descriptor vectors. Then, the global double linked list $\mathbf{L}$ is finally stored to each distributed database, via the *Cloud Database Server*, so as to preserve the final logical sorting of the 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 $\mathbf{p}$. In doing so, the *Dataset Updater* component is able to efficiently retrieve the minimum number of image descriptor vectors in the insertion step of CDVC, as we will describe in the following Section.

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  v_aj = value of v_a in dimension j = 1, ..., D;
2  v_bj = value of v_b in dimension j = 1, ..., D;
3  for j = 1 to D do
4      if v_aj > v_bj then
5          return 1;
6      end
7      if v_aj < v_bj then
8          return -1;
9      end
10 end
11 return 0;
```

Complexity Analysis
Given M *Image Sorter* nodes, the total complexity of the *Image Sorter* and *Global Image Sorter* is $O(D \cdot \frac{N}{M} \cdot log\frac{N}{M})$. Summarizing the total complexity of the *Preprocessing* step consists of the complexities of (a) *Dimension Value Cardinality Extractor*; (b) *Priority Indexer*; and (c) *Images Sorter* and *Global Image Sorter*. Therefore, based on (1) and (2), the total complexity of the *Preprocessing* step is

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

3.2. Insertion Algorithm

Role: To insert a new image descriptor vector $\mathbf{v}_q$ to the cloud infrastructure, 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 via the *Cloud Database Server* 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 $\mathbf{v}_q$ to the storage via the *Cloud Database Server*;
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 quick sort 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

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

3.3. 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 Comparator* nodes; (2) $2W$, the search radius.
Algorithm Description

Firstly, the insertion algorithm is utilized, in order 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 nodes over the cloud. The *Candidate Images' IDs Retrieval* component aggregates 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* node, i.e. $\mathbf{v}_q$ and $\mathbf{v}_i$, $i = 1, \ldots, 2W$. The output of each *Image Comparator* node 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 by the *Distance Collector* node and stored into a min-Heap[3] $\mathcal{H}$. 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 CDVC'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** node, 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. In order 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* node, based on a predefined distance measure $d(\cdot)$, e.g. L1, L2, squared-L1, etc. Since the *Image Comparator* node contains the most essential process of the query processing algorithm, on each m-th *Image Comparator* node 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

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

$$\mathcal{V} = \begin{cases} \mathbf{v}_1 = \langle 9,6,3,1,2,7\rangle \\ \mathbf{v}_2 = \langle 9,6,0,1,3,8\rangle \\ \mathbf{v}_3 = \langle 9,5,3,0,5,7\rangle \\ \mathbf{v}_4 = \langle 9,6,3,0,4,6\rangle \\ \mathbf{v}_5 = \langle 9,4,3,1,9,6\rangle \\ \mathbf{v}_6 = \langle 9,6,8,1,10,9\rangle \\ \mathbf{v}_7 = \langle 9,3,3,0,8,9\rangle \\ \mathbf{v}_8 = \langle 9,6,4,0,6,4\rangle \\ \mathbf{v}_9 = \langle 9,6,6,0,7,0\rangle \\ \mathbf{v}_{10} = \langle 9,6,6,0,6,0\rangle \end{cases} \mathcal{V}'^{(1)} = \begin{cases} \mathbf{v}'_5 = \langle 9,6,3,4,1,9\rangle \\ \mathbf{v}'_3 = \langle 5,7,3,5,0,9\rangle \\ \mathbf{v}'_4 = \langle 4,6,3,6,0,9\rangle \\ \mathbf{v}'_2 = \langle 3,8,0,6,1,9\rangle \\ \mathbf{v}'_1 = \langle 2,7,3,6,1,9\rangle \end{cases} \mathcal{V}'^{(2)} = \begin{cases} \mathbf{v}'_6 = \langle 10,9,8,6,1,9\rangle \\ \mathbf{v}'_7 = \langle 8,9,3,3,0,9\rangle \\ \mathbf{v}'_9 = \langle 7,0,6,6,0,9\rangle \\ \mathbf{v}'_8 = \langle 6,4,4,6,0,9\rangle \\ \mathbf{v}'_{10} = \langle 6,0,6,6,0,9\rangle \end{cases}$$

$$\mathcal{V}' = \begin{cases} \mathbf{v}'_6 = \langle 10,9,8,6,1,9\rangle \\ \mathbf{v}'_5 = \langle 9,6,3,4,1,9\rangle \\ \mathbf{v}'_7 = \langle 8,9,3,3,0,9\rangle \\ \mathbf{v}'_9 = \langle 7,0,6,6,0,9\rangle \\ \mathbf{v}'_8 = \langle 6,4,4,6,0,9\rangle \\ \mathbf{v}'_{10} = \langle 6,0,6,6,0,9\rangle \\ \mathbf{v}'_3 = \langle 5,7,3,5,0,9\rangle \\ \mathbf{v}'_8 = \langle 4,6,3,6,0,9\rangle \\ \mathbf{v}'_9 = \langle 3,8,0,6,1,9\rangle \\ \mathbf{v}'_1 = \langle 2,7,3,6,1,9\rangle \end{cases}$$

Fig. 2. Toy example.

similarity search time. Then, each calculated distance $d(\mathbf{v}_{iter}, \mathbf{v}_q)$ is retrieved by the *Distance Collector* node, via a queue, which stores the respective distance and the candidate image ID 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 $\mathcal{R}$. Finally, the top-k IDs in $\mathcal{R}$ are stored into the cloud storager, via the *Cloud Database Server* and returned to users via the *CDVC Scheduler* component.

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 (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 nodes, assigned to the *Distance Comparator* component. 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* node. Additionally, since $2W$ image IDs are inserted into the heap $\mathcal{H}$, the insertion of each image t into the heap is performed in $O(log2\cdot W)$. However, in our implementation we improved the complexity by preserving in the heap the k most similar images to query q, over the execution of the *Query Processing* algorithm. In particular, let c be the current candidate image for being inserted into the heap $\mathcal{H}$. Also, let t_k be the image ID with the k largest distance from query q, where t_k is currently stored into the heap $\mathcal{H}$. If the condition $d(\mathbf{v}_q, \mathbf{v}_c) < d(\mathbf{v}_q, \mathbf{v}_{tk})$ does hold, then c is discarded. Otherwise, t_k is removed from the heap $\mathcal{H}$ and candidate c is inserted. Therefore, over the execution of the *Query Processing* algorithm, k images are preserved into the heap $\mathcal{H}$ and thus, the complexity of the insertion of each image t into the heap is reduced from $O(log2W)$ to $O(logk)$. In doing so, the complexity of the distance calculations is $O(\frac{2\cdot W}{M\cdot T}\cdot D\cdot logk)$. Finally, the retrieval of the k image IDs from the heap $\mathcal{H}$ has a $O(k)$ complexity. Summarizing, the final complexity of the CDVC's *Query Processing* algorithm is

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

3.4. Toy Example

In Figure 2, we present an example of $N = 10$ image descriptor vectors of $D = 6$ dimensions of *integer* value type [Tiakas et al. 2013]. The stored dataset over the cloud

infrastructure is denoted by the set $\mathcal{V}$ of descriptor vectors. In the *Preprocessing* phase, 2 nodes of each of the following components are assigned: *Dimension Value Cardinality Extractor*, *Image Sorter* and *Image Comparator*. After the *Preprocessing* phase terminates, the final sorted set $\mathcal{V}'$ is generated. Then, the goal is to retrieve the top-1 similar image descriptor $\in \mathcal{V}'$ to a query descriptor vector $\mathbf{v}_{11} \notin \mathcal{V}'$.

In the *Preprocessing* step 1 (Section 3.1.1), the 2 *Dimension Value Cardinality Extractor* nodes have lower lb, 1 and 4, and upper ub, 3 and 6, dimensions' bounds. Thus, the dimensions' ranges are [1 3] and [4 6] for *Dimension Value Cardinality Extractor* nodes 1 and 2, respectively. The generated value cardinalities vectors are $\mathbf{c}^{(1)} = \langle 1, 4, 5, 0, 0, 0 \rangle$ and $\mathbf{c}^{(2)} = \langle 0, 0, 0, 2, 9, 6 \rangle$ for nodes 1 and 2, respectively. For instance, $\mathbf{c}_2^{(1)}$=4 denotes that for the 1st node in the 2nd dimension of all 10 descriptor vectors in $\mathcal{V}$ there are 4 distinct values, i.e. 6, 5, 4 and 3. In the preprocessing step 2 (Section 3.1.2), the *Priority Indexer* component aggregates the $\mathbf{c}^{(1)}$ and $\mathbf{c}^{(2)}$ value cardinalities vectors into the global dimensions value cardinalities vector $\mathbf{C} = \langle 1, 4, 5, 2, 9, 6 \rangle$. Then, according to $\mathbf{C}$, the priority index vector $p = \langle 5, 6, 3, 2, 4, 1 \rangle$ is calculated, where the dimensions of high value cardinalities have high priority indexes. For instance, given D=6, the 5th dimension, with $\mathbf{C}_5 = 9$ has the highest value cardinality, and thus $p_1 = 5$, whereas the 1st dimension, with $\mathbf{C}_1 = 1$ has the lowest value cardinality and thus $p_6 = 1$. Then, in preprocessing step 3 (Section 3.1.3), the 2 *Image Sorter* nodes retrieve the priority index vector $\mathbf{p}$ as well as the subsets of image IDs 1-5 and 6-10, respectively. Firstly, each *Image Sorter* node reorders the dimensions of the descriptors based on $\mathbf{p}$, from the highest to the lowest priority index. For instance, according to priority index vector $p = \langle 5, 6, 3, 2, 4, 1 \rangle$ the $m = 1$ *Image Sorter* node reorders the dimensions of descriptor $\mathbf{v}_1 = \langle 9, 6, 3, 1, 2, 7 \rangle$, generating descriptor $\mathbf{v}'_1 = \langle 2, 7, 3, 6, 1, 9 \rangle$ of sorted dimensions. Then, the positions of the descriptors are sorted in descending order based on Algorithm 3, generating the $\mathcal{V}'^{(1)}$ and $\mathcal{V}'^{(2)}$ subsets. In doing so, the two *Image Sorter* nodes compute the lists of logical positions $\mathbf{L}^{(1)} = \langle 5, 3, 4, 2, 1 \rangle$ and $\mathbf{L}^{(2)} = \langle 6, 7, 9, 8, 10 \rangle$, respectively. Finally, the *Global Image Sorter* node retrieves both $\mathbf{L}^{(1)}$ and $\mathbf{L}^{(2)}$ position lists and iteratively compares the respective descriptors based on Algorithm 3, until the final $\mathcal{V}'$ is generated. Therefore, the double linked list $\mathbf{L} = \langle 6, 5, 7, 9, 8, 11, 10, 3, 4, 2, 1 \rangle$ is generated. For instance, $\mathbf{L}_1 = 6$ denotes that the descriptor $\mathbf{v}'_6$ is located in the first position pos_1 in $\mathbf{L}$. Finally, the dimension with the highest cardinality value (the first sorted dimension) is defined as the primary key pk, depicted in green fonts in Figure 2.

Given, the new query descriptor vector $\mathbf{v}_{11} = \langle 9, 5, 3, 0, 6, 3 \rangle \notin \mathcal{V}'$, the *Dataset Updater* component initializes the insertion algorithm (Section 3.2). According to $p = \langle 5, 6, 3, 2, 4, 1 \rangle$, the query vector is reordered into $\mathbf{v'}_{11} = \langle 6, 3, 3, 5, 0, 9 \rangle$. Then, according to Algorithm 4 and based on the primary key (the first sorted dimension in $\mathcal{V}'$), images 8 and 10 form the $\mathcal{V}_{pk}$ set, since $\mathbf{v'}_{8,1} = \mathbf{v'}_{10,1} = \mathbf{v'}_{11,1} = 6$. Image IDs 8 and 10 are sorted based on their storage position in $\mathbf{L}$, i.e. pos_5 and pos_6, respectively. Since $pos_6 > pos_5$, we set $\mathbf{v}_{hp} = \mathbf{v'}_{10}$, i.e. the descriptor in the $\mathcal{V}_{pk}$ set of the highest position in list $\mathbf{L}$. Then, $\mathbf{v'}_{11}$ is compared with the $\mathbf{v'}_{10}$ descriptor vector based on Algorithm 3, and then $\mathbf{v'}_{11}$ descriptor vector is stored in position pos_6 in $\mathbf{L}$. Finally, the double linked list is updated as $\mathbf{L} = \langle 6, 5, 7, 9, 8, \mathbf{11}, 10, 3, 4, 2, 1 \rangle$.

Let the search radius W be equal to 2 for the query processing algorithm (Section 3.3). The *Candidate Images' Retrieval* component retrieves a set $\mathcal{V}_{2W}$ of $2W = 4$ candidate image IDs that are the 2 descriptors prior and the 2 descriptors next to position pos_6 of query 11 in the updated list $\mathbf{L}$, with $\mathcal{V}_{2W} = \{\mathbf{v}'_9, \mathbf{v}'_8, \mathbf{v}'_{10}, \mathbf{v}'_3\}$. The 2 *Images Comparator* nodes retrieve $\mathbf{v}'_{11}$ and subsets $\mathcal{V}^{(1)}_{2W} = \{\mathbf{v}'_9, \mathbf{v}'_8\}$ and $\mathcal{V}^{(2)}_{2W} = \{\mathbf{v}'_{10}, \mathbf{v}'_3\}$, respectively. For the first *Images Comparator* node, the distances between the descriptor

vectors in $\mathcal{V}_{2W}^{(1)}$ and $\mathbf{v}'_{11}$ are 3.4641 and 1.732 respectively, based on the Euclidian distance measure. For the second *Images Comparator* node, the distances between the $\mathcal{V}_{2W}^{(2)}$ and $\mathbf{v}'_{11}$ are 4.359 and 7.746, respectively. The calculated distances along with the corresponding image IDs are retrieved by the *Distance Collector* component and stored into the min-heap $\mathcal{H}$. The top-1 result is image 8, having the minimum distance $d(\mathbf{v}'_8, \mathbf{v}'_{11})$=1.732. Finally, image 8 is returned to the *CDVC Scheduler* component.

4. EXPERIMENTS

4.1. Datasets

In our experiments we evaluate the proposed CDVC framework on seven publicly available datasets of image descriptor vectors. The first 5 lower-scale datasets were also used in the experimental evaluation of the MSIDX method of [Tiakas et al. 2013] of single node architecture [4]. The ImageClef Wikipedia Retrieval 2010 Collection [5] features N=240K images. The collection provides global CIME descriptors [Stehling et al. 2002] of 64-dimensions (**CIME-240K-64d**), global CEDD descriptors [Chatzichristos and Boutalis 2008] of 144-dimensions (**CEDD-240K-144d**) and global SURF descriptors [CBay et al. 2008] of 5000-dimensions (**SURF-240K-5000d**). Additionally, we evaluate the proposed framework on the TEXMEX collection [6] featuring two descriptor datasets of N=1M images, namely local SIFT [Lowe 2004] descriptors of 128-dimensions (**SIFT-1M-128d**) and global GIST [Oliva and Torralba 2001] descriptors of 960-dimensions (**GIST-1M-960d**). For the very large-scale experiments we evaluate CDVC on the Tiny Image collection [7] of N=80M images of GIST descriptors of 348-dimensions (**GIST-80M-348d**) and N=1B images of SIFT descriptors of 128-dimensions of the TEXMEX collection (**SIFT-1B-128d**). The last two large-scale datasets were also used in the experimental evaluation of the Multi-Index Hashing framework of [Norouzi et al. 2014] of parallel architecture.

4.2. Experimental Settings

Following the evaluation protocol of [Heo et al. 2012; Tiakas et al. 2013], we performed 1,000 test queries, which were randomly chosen and did not participate into the training/preprocessing phase. For each query, the search accuracy is measured in terms of mAP according to the following ratio:

$$mAP = \frac{|\mathcal{R}_{seq} \cap \mathcal{R}_{ind}|}{k} \tag{6}$$

where, $\mathcal{R}_{seq}$ is the set of the top-k results (Euclidean neighbors) retrieved by the sequential search based on the Euclidean distance, and $\mathcal{R}_{ind}$ 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 variable, which is defined as the average search accuracy of the 1,000 performed queries.

Two are the most crucial parameters in the CDVC framework, the number of nodes M and the search radius $2W$. Parameter M denotes the number of nodes that are used for parallel processing over the cloud infrastructure affecting (a) the total preprocessing cost based on (3) and (b) the search time in the query processing algorithm based on (5). In our experiments, we evaluate the CDVC framework, using 2, 4, 6, 8 nodes for parallel processing over the cloud infrastructure. The search radius $2W$ is used in

[4]Similarity search strategies in single node architectures employ a single core CPU.
[5]http://www.imageclef.org/wikidata
[6]http://corpus-texmex.irisa.fr/
[7]http://horatio.cs.nyu.edu/mit/tiny/data/index.html

the query processing algorithm, affecting the search time based on (5) and the mAP accuracy. We varied the search radius 2W in (0.1%, 1%, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, 50%), expressed as a percentage of the N total dataset size.

In Table I, we present the insertion times and the sizes of $|\mathcal{V}_{pk}|$ for the seven evaluation datasets. According to (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 nodes and the dataset size N. For instance, **SURF-240k-5000d** has the highest dimensionality, requiring thus the most expensive insertion time. Additionally, despite the fact that the sizes of **SIFT-1M-128d** and **SIFT-1B-128d** differ in three orders of magnitude, the insertion times are comparable, 4.099 and 4.217 msecs respectively, due to the datasets same dimensionality and similar $|\mathcal{V}_{pk}|$ sizes, i.e. 147 and 192. Finally, the insertion times are significantly lower than the online search times of CDVC (Figures 3-5).

Table I. INSERTION TIME (MSEC) AND $|\mathcal{V}_{pk}|$

	CIME-64d	**CEDD-144d**	**SURF-5000d**	**SIFT-128d**	**GIST-960d**	**SIFT-1B-128d**	**GIST-80M-384d**
Insertion time	2.726	6.824	19.123	4.099	13.087	4.217	8.27
$\lvert\mathcal{V}_{pk}\rvert$	26	6	1,000	147	2,692	192	2,235,194

The proposed CDVC framework [8] was implemented in Java using the Windows Azure Emulator [9] 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. Since our experiments were performed on emulated clouds, we report average CPU time. Moreover, on each node of our proposed CDVC framework, we allocated 1 CPU and 2 GB RAM for all CDVC components, whereas in the case of the *Image Comparator* node we allocated 4 CPUs and 4 GB RAM providing the maximum amount of T=8 parallel threads. Finally, in our online Appendix B, we report the experimental results on the two large-scale datasets **SIFT-1B-128d** and **GIST-80M-348d** over a real cloud infrastructure, including the network latency and the CPU overhead of the proposed CDVC framework.

4.3. Comparison Of CDVC Against Similarity Search Strategies in Single Node Architectures

In Figure 3, we present the experimental results in the **CIME-240K-64d**, **CEDD-240K-144d**, **SIFT-1M-128d SURF-240K-5000d** datasets for 100-NN queries and the **GIST-1M-960d** dataset for 1000-NN queries, where CDVC for all node variations clearly outperforms the MSIDX method of single node architecture. This happens because CDVC is based on our cloud-based architecture of the parallel query processing algorithm (Algorithm 5). Moreover, by increasing the number of nodes in CDVC the search time is significantly decreased. CDVC has similar performance in terms of mAP with the MSIDX method for the same $2W$ search radius, since both methods are based on dimensions value cardinalities of the images descriptor vectors. In doing so, both techniques assume that dimensions of high value cardinalities have more discriminative power and thus prioritize the dimensions in the searching strategy, accordingly. However, the search time is highly reduced, especially in the case of CVDC with 8 nodes, compared to MSIDX. Additionally, since the MSIDX method is of single node architecture, it is not able to work in distributed databases, having also memory limitations for very large-scale datasets, in contrast to our cloud-based architecture of CDVC. Moreover, in Figure 3(f) we compare CDVC against state-of-the-art hashing methods of single node architecture, as presented in [Heo et al. 2012]. The

[8] We made the source code of CDVC publicly available at http://github.com/stefanosantaris/CDVC
[9] http://www.windowsazure.com/en-us/

hashing methods are: LSH [Gionis et al. 1999]; LSH-ZC [Datar et al. 2004]; PCA-ITQ [Gong and Lazebnik 2011]; GSPICA-RBF [He et al. 2011]; RMMH [Joly and Buisson 2011]; LSBC [Raginsky and Lazebnik 2009]; SpecH [Weiss et al. 2008]; and SPH [Heo et al. 2012]. In Figure 3(f), we varied the number of bits from 32 to 512 for the hashing methods. For 32, 64, 128, 256, 512 bits the search times of the hashing methods are comparable (i.e. 364, 537, 743, 1014 and 1438 msecs, respectively). In order to have similar search times for each number of bits variation, we varied the search radius $2W$ for MSIDX and CDVC with 8 nodes analogously, i.e. (5%, 7.5%, 10%, 15%, 22.5%) and (7.5%, 12.5%, 25%, 42.5%, 50%). CDVC clearly overcomes the rest similarity search strategies of single node architectures in terms of mAP for the same search time.

In Table II, the preprocessing time requirements of CDVC are presented in the first 5 lower-scale evaluation datasets. We also report the respective requirements for MSIDX and SPH of single node architectures, which outperform the rest of hashing methods in terms of mAP for comparable online search time. CDVC has clearly lower preprocessing requirements, especially in the case of using 8 nodes, compared to the rest of similarity search strategies of single node architectures. This happens because the preprocessing algorithm of CDVC performs in parallel over the cloud, by significantly decreasing the time requirements. Finally, the influence of CDVC's preprocessing steps to the overall time is presented and discussed in our online Appendix C.

Table II. PREPROCESSING TIME REQUIREMENTS IN THE LOWER-SCALE DATASETS (SEC)

	CIME-240K-64d	**CEDD-240K-144d**	**SIFT-1M-128d**	**GIST-1M-960d**	**SURF-240K-5000d**
CDVC (2 nodes)	2.12	2.83	16.49	19.35	20.36
CDVC (4 nodes)	1.97	2.03	12.90	16.08	18.48
CDVC (6 nodes)	1.82	1.86	10.67	13.8	16.76
CDVC (8 nodes)	**1.8**	**1.75**	**9.98**	**11.08**	**12.98**
MSIDX	3.90	4.21	20.28	24.68	29.93
SPH (64 bits)	75.5	104.71	114.37	1,149.37	3,263.96
SPH (128 bits)	164.13	257.68	243.74	2,350.71	7,041
SPH (256 bits)	373.34	534.76	510.68	4,669.38	13,759.61
SPH (512 bits)	837.15	1,168.82	1,103.28	9,405.82	28,129.58
SPH (1024 bits)	2,020.5	2,638.82	2,402.61	19,186.04	58,079.64

4.4. Comparison of CDVC Against Similarity Search Strategies In Parallel Architectures And Cloud Infrastructures

In the next set of experiments, we compared CDVC against the Multi Index Hashing [10] (MIH) method of Norouzi et al. [2014] of parallel architecture. The MIH method achieves parallel processing by using multiple hash tables. For making fair comparison, we evaluate the performance of CVDC of M nodes against MIH of M hash tables. Following the experimental evaluation of MIH, we used the hashing methods of Locality Sensitive Hashing (LSH) and Minimal Loss Hashing (MLH) of Norouzi and Fleet [2011]. 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. The number of hash tables were varied as the nodes in CDVC, i.e. 2, 4, 6 and 8 nodes/hash tables. Additionally, we compared CDVC against the distributed KD-Trees similarity search strategy over cloud infrastructures [11] by Aly et al. [2011]. Following the evaluation strategy of Aly et al. [2011], in the online search process we used the parameter of backtracking steps, denoting the fixed budget

[10] In our experiments, we used the implementation of MIH, publicly available at https://github.com/norouzi/mih

[11] We made the implementation publicly available at http://github.com/stefanosantaris/KD_Trees.

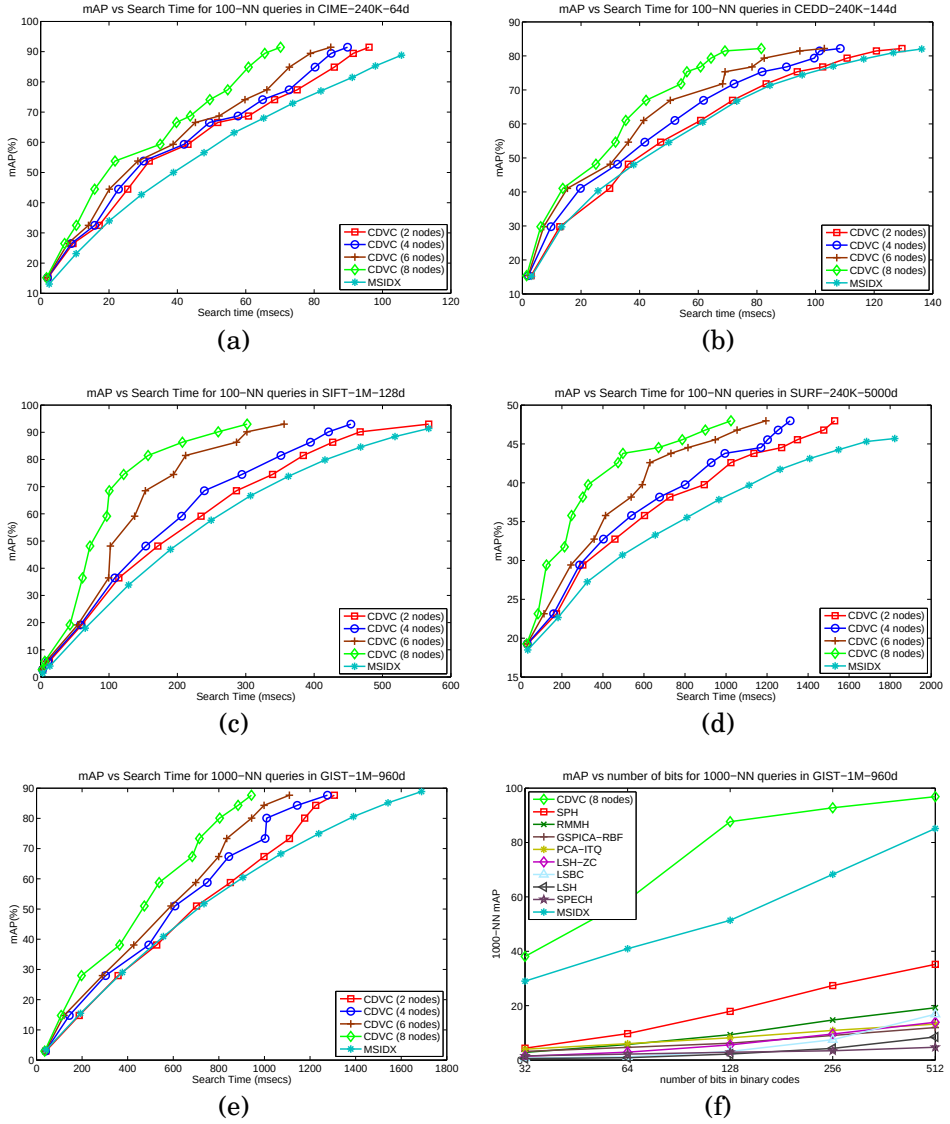


Fig. 3. Performance of CVDC on the evaluation datasets of (a) **CIME-240K-64d**, (b) **CEDD-240K-144d**, (c) **SIFT-1M-128d** (d) **SURF-240K-5000d** and (e) **GIST-1M-960d**, as presented in [Tiakas et al. 2013]. (f) Comparison of CDVC against state-of-the-art hashing methods, as presented in [Heo et al. 2012]. For the same number of bits variations we varied the search radius $2W$ of MISDX and CDVC analogously to have comparable search time with the hashing methods.

for doing backtracking steps for every dimension which is shared among all the KD-Trees searched for this dimension. The backtracking steps 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. Following Aly et al. [2011], the number of KD-trees is equal to the number of the CPU-cores. Also, we compared CDVC against FLANN [Muja and Lowe 2014]. Following the evaluation strategy of Muja and Lowe [2014] on large-scale datasets we used the randomized KD-trees algorithm using

(2, 4, 6, 8) CPU-cores to search in parallel over the constructed KD-trees. In the preprocessing step, the publicly available implementation of FLANN [12] by default uses all available CPU-cores. In Flann, we varied the number of randomized KD-trees in (2^2, 2^3,2^4,2^5,2^6,2^7,2^8,2^9,2^{10}), where we concluded to 256 and 512 for the **SIFT-1B-128d** and **GIST-80M-384d** datasets, respectively, since we noticed that a further increase of the number of randomized KD trees, slightly increases the mAP accuracy without paying off in terms of search time. Following [Muja and Lowe 2014;Wang et al. 2014], in our experiments we used the parameter of the branching factor, varied in (2^2, 2^3,2^4,2^5,2^6,2^7,2^8,2^9,2^{10}). We also report the experimental results against two single node architectures Inverted Multi-Index [Babenko and Lempitsky 2012] and Cartesian k-means (ckmeans) [Norouzi and Fleet 2013], since both methods were also evaluated on the two large-scale datasets **SIFT-1B-128d** and **GIST-80M-348d**. According to the experimental evaluation in [Babenko and Lempitsky 2012] and [Norouzi and Fleet 2013], for Inverted Multi-Index [13] we varied the list length in the range of (2^8,2^{12},2^{16}) using a codebook with size 2^{14} and in ckmeans [14] we varied the number of centers in (2^8, 2^{16},2^{32},2^{64},2^{128},2^{256}). Similar to CDVC, the publicly available implementations of the competitive methods, by default, use the same maximum number of threads (T=8). For CDVC we varied the search radius $2W$ as in the previous set of experiments. In Figures 4 and 5, we present the experimental results of CDVC against MIH, distributed KD-Trees, FLANN, as well as against Inverted Multi-Index and ckmeans in the large-scale datasets of **SIFT-1B-128d** and **GIST-80M-348d** for 100-NN queries. In both datasets, CDVC outperforms the similarity search strategies of MIH-LSH, MIH-MLH, distributed KD-Trees, FLANN, Inverted Multi-Index and ckmeans for the same settings, i.e. number of nodes are equal to the number of hash tables or to number of cores.

In Table III, we present the preprocessing time requirements for CDVC against MIH-LSH, MIH-MLH, distributed KD-Trees, FLANN, Inverted Multi-Index and ckmeans. 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 distributed KD-Trees method, the number of nodes are 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 8 CPU-cores in the preprocessing step and thus we report only one preprocessing cost. Finally, Inverted Multi-Index and ckmeans do not support parallel preprocessing; moreover we can observe that the list length and the number of centers play essential role in the preprocessing costs of Inverted Multi-Index and ckmeans, respectively. In all cases the proposed CDVC framework has the less preprocessing requirements.

Summarizing, CDVC has extremely low preprocessing requirements, in contrast to the competitive similarity search strategies. This happens because the offline algorithms of the preprocessing phase of CDVC (Section 3.1) avoid to use complex index structures and function in parallel efficiently. For instance, in the case of **SIFT-1B-128d** with 8 nodes/HT the preprocessing phases of CDVC, MIH-MLH (16-bit) distributed KD-Trees and FLANN finish in 20.075, 63.824, 797.04 and 618 secs, respectively. For these settings, CDVC achieves a speed up factor 3, 40 and 33 in the preprocessing time, against these methods respectively. The main differences between CDVC and the competitive methods are that the MIH's preprocessing step does not function

[12]https://github.com/mariusmuja/flann
[13]https://github.com/arbabenko/MultiIndex
[14]https://github.com/norouzi/ckmeans

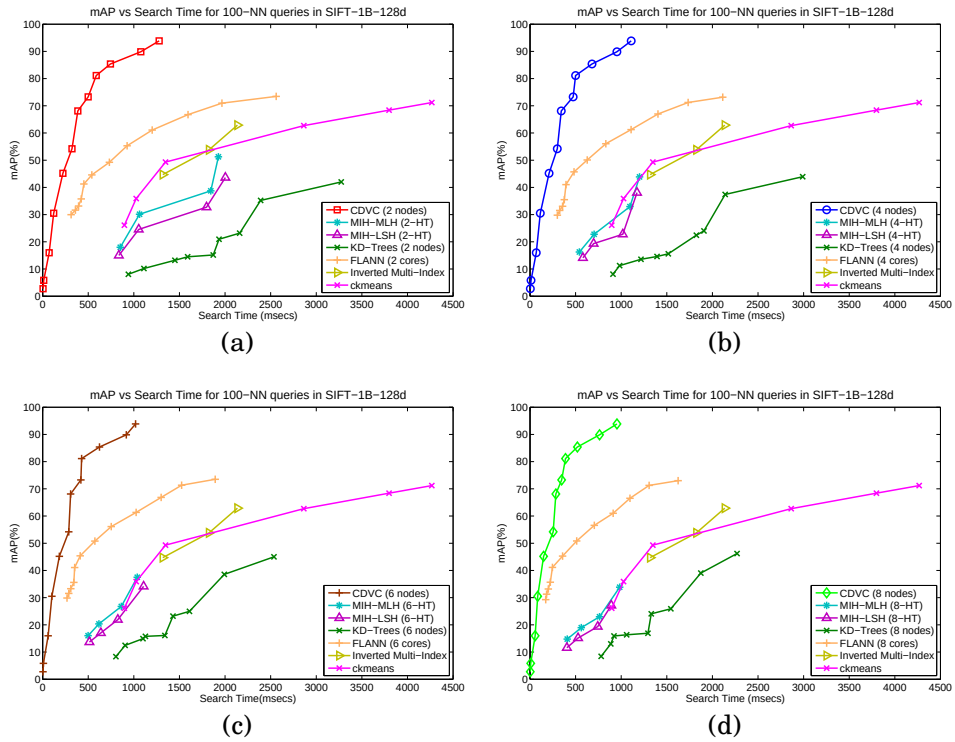


Fig. 4. Comparison of CVDC on the **SIFT-1B-128d** evaluation dataset against MIH-LSH, MIH-MLH, distributed KD-Trees over the cloud (KD-Trees) and FLANN for (a) 2, (b) 4, (c) 6 and (d) 8 nodes / Hash Tables (HT) / CPU-cores. The performances of Inverted Multi-Index and ckmeans are preserved by varying the number of nodes, since both methods do not work in parallel.

Table III. PREPROCESSING TIME REQUIREMENTS IN THE LARGE-SCALE DATASETS (SEC)

	SIFT-1B-128d	**GIST-80M-384d**		**SIFT-1B-128d**	**GIST-80M-384d**
CDVC (2 nodes)	25.07	27.68	MIH-LSH (8-bit)	55.97	69.57
CDVC (4 nodes)	22.76	25.22	MIH-LSH (16-bit)	61.76	78.15
CDVC (6 nodes)	21.81	23.73	MIH-LSH (32-bit)	87.18	102.78
CDVC (8 nodes)	**20.07**	**21.99**	MIH-LSH (64-bit)	120.84	156.80
KD-Trees (2 nodes)	948	972	MIH-MLH (8-bit)	50.35	67.2
KD-Trees (4 nodes)	898.2	943.8	MIH-MLH (16-bit)	63.82	81.30
KD-Trees (6 nodes)	892.08	894.36	MIH-MLH (32-bit)	91.07	124.67
KD-Trees (8 nodes)	797.04	830.94	MIH-MLH (64-bit)	114.92	167.32
ckmeans (2^8)	1,680	1,500	FLANN	618	918
ckmeans (2^{16})	3,720	2,580	Inverted Multi-Index (2^8)	94,740	69,660
ckmeans (2^{32})	8,580	3,840	Inverted Multi-Index (2^{12})	116,640	83,880
ckmeans (264)	17,880	7,680	Inverted Multi-Index (2^{16})	134,940	106,620
ckmeans (2^{128})	21,720	15,120			
ckmeans (2^{256})	24,840	21,360			

in parallel, whereas the distributed KD-Tree method and FLANN require a significant preprocessing cost to built the complex structures of distributed KD-Trees and randomized KD-trees, respectively. Meanwhile, Inverted Multi-Index and ckmeans have heavy computational costs, by increasing the list length and the number of centers, without paying off in terms of search time and mAP in the online search. With respect to the online query processing performance, CDVC achieves high mAP in low search time. This is achieved by exploiting the dimensions value cardinalities and efficiently splitting the computational effort of the query processing algorithm to the cloud infrastructure's nodes (Algorithm 5). This comes in contrast to the competitive

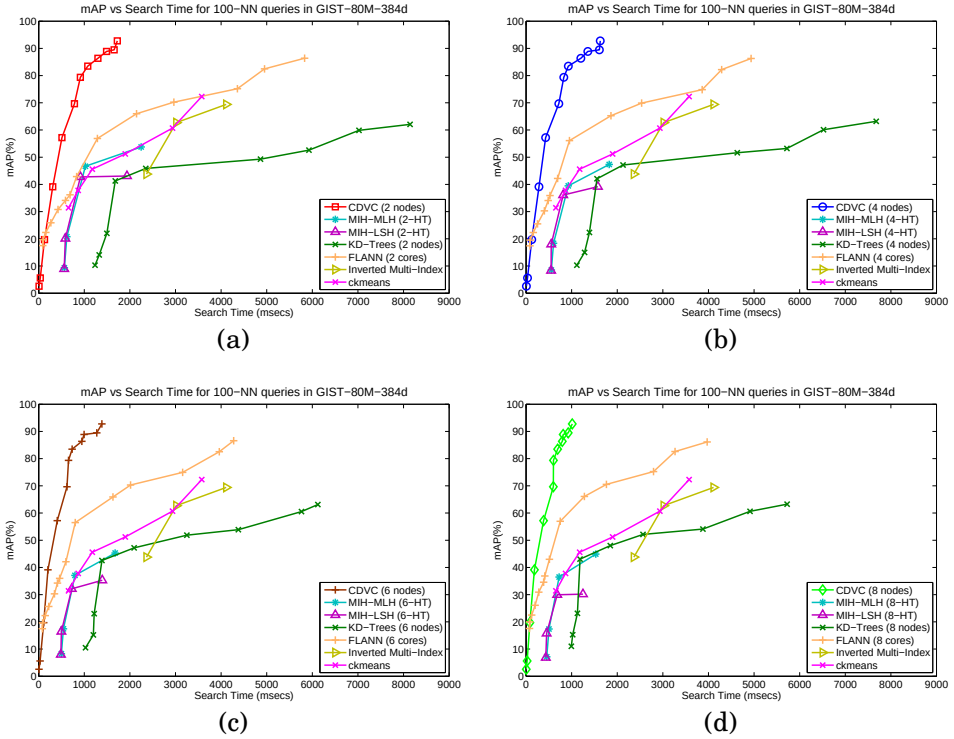


Fig. 5. Comparison of CVDC on the **GIST-80M-348d** evaluation dataset evaluation dataset against MIH-LSH, MIH-MLH, distributed KD-Trees over the cloud (KD-Trees) and FLANN for (a) 2, (b) 4, (c) 6 and (d) 8 nodes / Hash Tables (HT) / CPU-cores. The performances of Inverted Multi-Index and ckmeans are preserved by varying the number of nodes, since both methods do not work in parallel.

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, making thus the search time high.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a similarity search strategy over cloud infrastructures based on the image descriptors' Dimensions Value Cardinalities, called CDVC. The preprocessing algorithm has low requirements, by avoiding to construct complex index structures over the cloud and dividing the computation cost into several nodes. Additionally, our proposed insertion algorithm is of low computational complexity, depending on the dimensionality of the image descriptor vectors and a small subset of image descriptor vectors that have similar dimensions value cardinalities. The CDVC's query processing algorithm performs in parallel over the cloud, where the dimensions of image descriptors are prioritized in the searching strategy based on their dimensions value cardinalities. The query processing effort is divided into several nodes over the cloud infrastructure and thus the high-computational cost of similarity search is significantly reduced. Extensive experiments over seven widely used benchmark image datasets have demonstrated the effectiveness of the proposed CDVC similarity search strategy against competitive methods of single node, parallel and cloud-based architectures in terms of preprocessing cost, search time and accuracy. An interesting

topic for future work is to evaluate the proposed CDVC framework on other multimedia datasets, such as audio [Knees and Schedl 2013] or video [De Rooij and Worring 2012; Carbunar et al. 2013; Hua 2013].

REFERENCES

M. Aly, M. Munich, and P. Perona. 2011. Distributed Kd-Trees for Retrieval from Very Large Image Collections. In *Proceedings of BMVC'11*.

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

M. Batko, D. Novak, F. Falchi, and P. Zezula. 2008. Scalability comparison of peer-to-peer similarity search structures. *Future Generation Computer Systems* 24, 8 (2008), 834–848.

T. Bozkaya and M. Ozsoyoglu. 1999. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)* 24, 3 (1999), 361–404.

B. Carbunar, R. Potharaju, M. Pearce, V. Vasudevan, and M. Needham. 2013. A framework for network aware caching for video on demand systems. *ACM Trans. on Multimedia Computing, Communications, and Applications* 9, 4 (2013).

H. CBay, A. Ess, T. Tuytelaars, and L. V. Gool. 2008. SURF: Speeded Up Robust Features. *Computer Vision and Image Understanding* 110, 3 (2008), 346–359.

S. A. Chatzichristos and Y. S. Boutalis. 2008. CEDD: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval. *ICVS* 5008 (2008), 312–322.

H. Cheng, K. A. Hua, K. Vu, and D. Liu. 2008. Semi-supervised dimensionality reduction in image feature space. In *Proceedings of the 2008 ACM symposium on Applied computing*. 1207–1211.

M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of Symposium on Computational Geometry*. 253–262.

O. De Rooij and M. Worring. 2012. Efficient targeted search using a focus and context video browser. *ACM Trans. on Multimedia Computing, Communications, and Applications* 8, 4 (2012).

A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon. 2000. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The international Journal on Very Large Data Bases* 9, 2 (2000), 154–173.

A. Gionis, P. Indyk, and Motwani R. 1999. Similarity search in high dimensions via hashing. In *Proceedings of International Conference on Very Large Data Bases*. 518–529.

Y. Gong and S. Lazebnik. 2011. Iterative quantization: a procrustean approach to learning binary codes. In *Proceedings of CVPR*. IEEE, 817–824.

J. He, R. Radhakrishnan, S.F Chang, and Bauer C. 2011. Compact hashing with joint optimization of search accuracy and time. In *Proceedings of CVPR'11*. IEEE, 753–760.

J. P. Heo, Y. Lee, J. He, S. Chang, and S. Yoon. 2012. Spherical Hashing. In *Proceedings of CVPR'12*. IEEE, 2957–2964.

K.A Hua. 2013. Online video delivery: Past, present, and future. *ACM Trans. on Multimedia Computing, Communications, and Applications* 9, 1s (2013).

Z. Huang, H. T. Shen, J. Liu, and X. Zhou. 2011. Effective data co-reduction for multimedia similarity search. In *Proceedings of ACM SIGMOD*. ACM, 1021–1032.

K. Jarrah and L. Guan. 2008. Content-Based Image Retrieval via Distributed Databases. In *Proceedings of CIVR'08*. 389–394.

H. Jegou, M. Douze, and C. Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

Y. Jia, J. Wang, G. Zeng, H. Zha, and X. S. Hua. 2010. Optimizing kd-trees for scalable visual descriptor indexing. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. 3392–3399.

A. Joly and O. Buisson. 2011. Random Maximum Margin Hashing. In *Proceedings of CVPR'11*. IEEE, 873–880.

Peter Knees and Markus Schedl. 2013. A Survey of Music Similarity and Recommendation from Music Context Data. *ACM Trans. Multimedia Comput. Commun. Appl.* 10, 1, Article 2 (Dec. 2013), 21 pages.

L. Liang, X. Wang, B. Yang, and J. Peng. 2010. Image dimensionality reduction based on the HSV feature. In *Proceedings of the 9th IEEE ICCI*. 127–131.

Xianglong Liu, Yadong Mu, Bo Lang, and Shih-Fu Chang. 2014. Mixed Image-keyword Query Adaptive Hashing over Multilabel Images. *ACM Trans. Multimedia Comput. Commun. Appl.* 10, 2, Article 22 (Feb. 2014), 21 pages.

D. Lowe. 2004. Distinctive image features from scale-invariant keypoints. *Int. Journal of Computer Vision* 60, 2 (2004), 91–110.

M. Muja and D. G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISAPP (1)*. 331–340.

M. Muja and D. G. Lowe. 2014. Scalable Nearest Neighbour Algorithms for High Dimensional Data. *IEEE Trans. Pattern Anal. Mach. Intell., to appear* (2014).

M. Norouzi and D. J. Fleet. 2011. Minimal Loss Hashing for Compact Binary Codes. In *Proceedings of ICML'11*.

M. Norouzi and D. J. Fleet. 2013. Cartesian K-Means. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '13)*. IEEE Computer Society, Washington, DC, USA, 3017–3024.

M. Norouzi, A. Punjani, and D. J. Fleet. 2014. Fast Exact Search in Hamming Space with Multi-Index Hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 6 (2014), 1107–1119.

D. Novak, M. Batko, and P. Zezula. 2008. Web-scale system for image similarity search: When the dreams are coming true. In *Proceedings of the sixth international workshop on content-based multimedia indexing*. IEEE.

D. Novak, M. Batko, and P. Zezula. 2012. Large-scale similarity data management with distributed Metric Index. *Information Processing and Management* 48 (2012), 855–872.

A. Oliva and A. Torralba. 2001. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *Int. Journal of Computer Vision* 42, 3 (2001), 145–175.

M. Raginsky and S. Lazebnik. 2009. Locality-sensitive binary codes from shift-invariant kernels. In *Proceedings of NIPS'09*. 1509–1517.

S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Scenker. 2001. A scalable content-addressable network. In *Proceedings of the SIGCOMM*, Vol. 31. 161 – 172.

C. Silpa-Anan and R. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. 1–8.

R.O. Stehling, M. A. Nascimento, and A. X. Falcao. 2002. A compact and efficient image retrieval approach based on border/interior pixel classification. In *Proceedings of CIKM*. 102–109.

E. Tiakas, D. Rafailidis, A. Dimou, and P. Daras. 2013. MSIDX: Multi-Sort Indexing for Efficient Content-based Image Search and Retrieval. *IEEE Transaction on Multimedia* 15, 6 (2013), 1415–1430.

Y. Tian, J. Srivastava, T. Huang, and N. Contractor. 2010. Social Multimedia Computing. *IEEE* (2010), 27–37.

R. H. Van Leuken and R.C. Veltkamp. 2011. Selecting vantage objects for similarity indexing. *ACM Trans. on Multimedia Computing, Communications, and Applications* 7, 3 (2011).

A. Vlachou, C. Doulkeridis, and Y. Kotidis. 2012. *Metric-Based similarity search in unstructured peer-to-peer systems*. Springer-Verlag, Heidelberg.

J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo. 2013. Fast Neighborhood Graph Search Using Cartesian Concatenation. In *ICCV*. 2128–2135.

J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. 2012. Scalable k-NN graph construction for visual descriptors. In *CVPR*. 1106–1113.

J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X. S. Hua. 2014. Trinary-Projection Trees for Approximate Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 2 (2014), 388–403.

J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. 2010. Indexing multi-dimensional data in a cloud system. In *Proceedings of SIGMOD*. 591–602.

Z. Wang and H. Binbin. 2011. Locality perserving projections algorithm for hyperspectral image dimensionality reduction. In *Proceedings of the 19th International Conference on Geoinformatics*. 1–4.

Y. Weiss, A. Torralba, and R. Fergus. 2008. Spectral hashing. In *Proceedings of NIPS'08*. 1753–1760.

Lei Zhang and Yong Rui. 2013. Image Search-from Thousands to Billions in 20 Years. *ACM Trans. Multimedia Comput. Commun. Appl.* 9, 1s, Article 36 (Oct. 2013), 20 pages.

M. Zhu, D. Shen, Y. Kou, T. Nie, and G. Yu. 2012. An Adaptive Distributed Index for Similarity Queries in Metric Spaces. *Web-Age Information Retrieval, Lecture Notes in Computer Science* 7418 (2012), 222–227.

W. Zhu, C. Luo, J. Wang, and S. Li. 2011. Multimedia Cloud Computing. *IEEE Signal Processing Magazine* 28, 3 (2011), 59–69.

Online Appendix to: Similarity Search Over The Cloud Based On Image Descriptors' Dimensions Value Cardinalities

STEFANOS ANTARIS and DIMITRIOS RAFAILIDIS, Aristotle University of Thessaloniki

A. SIMILARITY SEARCH BASED ON DIMENSIONS VALUE CARDINALITIES (DVC)

According to Tiakas et al. [2013], the basic idea of similarity search based on Dimensions Value Cardinalities (DVC) is: *to reorder the storage positions of images' descriptors according to value cardinalities of their dimensions, by performing a multiple sort algorithm, in order 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 there are the three following cases when calculating DVCs:

(a) **Integer values**: In case that the values are integer, 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 $j \in \{1, \ldots, D\}$.
(b) **Normalized real values**: In case that the values are real, produced by value normalization techniques of previously 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.
(c) **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. In our experiments no additional value quantization was used in all evaluation datasets.

An overview of similarity search based on DVC is presented in Figure 6. Given a set $\mathcal{V}$ of N D-dimensional image descriptor vectors $\mathbf{v}_i \in \mathcal{V}$, $i = 1, \ldots, N$, 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, assuming that dimensions with high DVC (c_j values) are more discriminative. For all the 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 $v'_{i,j}$ [15] in Figure 6 corresponds to the value of the highest DVC (c_j) value of the j-th dimension of the i-th descriptor vector $\mathbf{v}$. As depicted in Figure 6, 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 6 the first five image descriptors

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

DOI: http://dx.doi.org/10.1145/0000000.0000000

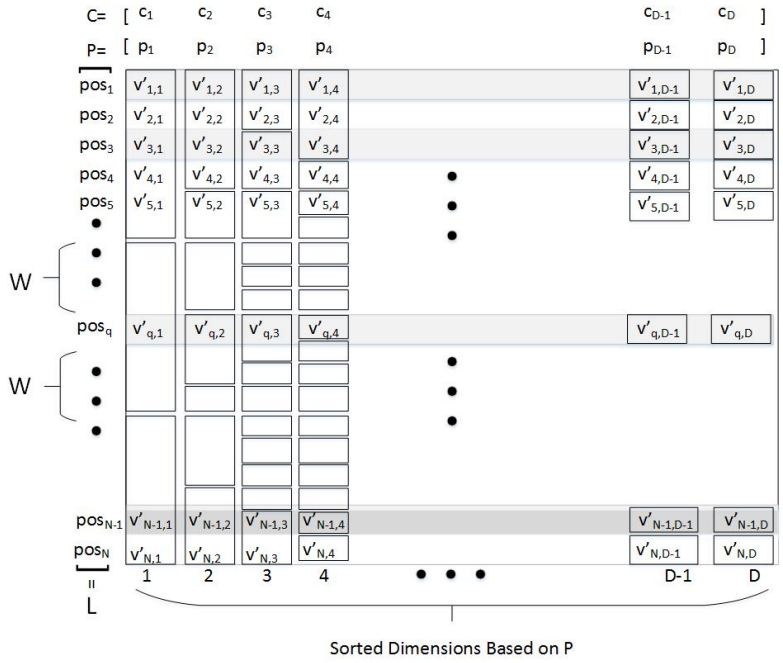


Fig. 6. Similarity Search Based On DVC.

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 5-th 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 having 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 the case of posing 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.

B. PERFORMANCE OF CDVC OVER A REAL CLOUD INFRASTUCTURE

To measure the performance of CDVC in terms of network latency and CPU overhead/usage, CDVC was evaluated on a real cloud infrastructure. Our real cloud computing infrastructure contains three distinct machines orchestrated under the Ganeti[16] virtual server management software tool. The specifications of each machine are presented in Table IV. Since the experiments in Section 4 were performed on Windows Azure Libraries, we used the RabbitMQ[17] message queuing platform equivalent to the Windows Azure Queues. Moreover, instead of using Windows Azure Tables we

[16]https://code.google.com/p/ganeti/
[17]http://www.rabbitmq.com/

utilized the HBase[18] distributed data storage. The HBase storager was installed over the cloud using 8 single-core nodes with maximum data storage size equal to 2.4 TB.

Table IV. CLOUD SPECIFICATIONS

CPU Architecture	Number of Cores	Size of RAM	Size of Disk
Intel Xeon 2.9 GHz	8	64 GB	4 TB
Intel i7 2.8 GHz	4	32 GB	2 TB
Amd 2.9 GHz	8	32 GB	2 TB

Following the experimental evaluation of Section 4, we varied the number of the M nodes in (2, 4, 6, 8). On each node, a single core virtual machine was assigned with 2 GB of RAM. Additionally, following the parallelization strategy in the *Image Comparator* node of the Query Processing Algorithm of Section 3.3 where T =8 parallel threads were used, a 4-core virtual machine was assigned on each *Image Comparator* node. In the case of the distances calculation in the *Image Comparator* node we used parallel threads in the same node and not in distinct nodes, in order to reduce the network latency that it would have been produced while transferring data between different nodes over the cloud. Finally, in contrast to the experiments of Section 4 over the emulated cloud, in our experiments over the real cloud infrastructure, the proposed CDVC framework preprocesses the N D-dimensional image descriptor vectors in a streaming mode, which means that as soon as each component finishes its task, the *CDVC Scheduler* sends the next task immediately. In Figure 7, the network latencies of the CDVC components are presented. The highest network latency is produced by the set of the M *Image Sorter* nodes in the preprocessing step, since they take as input the N/M D-dimensional descriptor vectors (the largest data transfer in terms of bytes, compared to the inputs of the rest of the CDVC components, as shown in Figure 1).

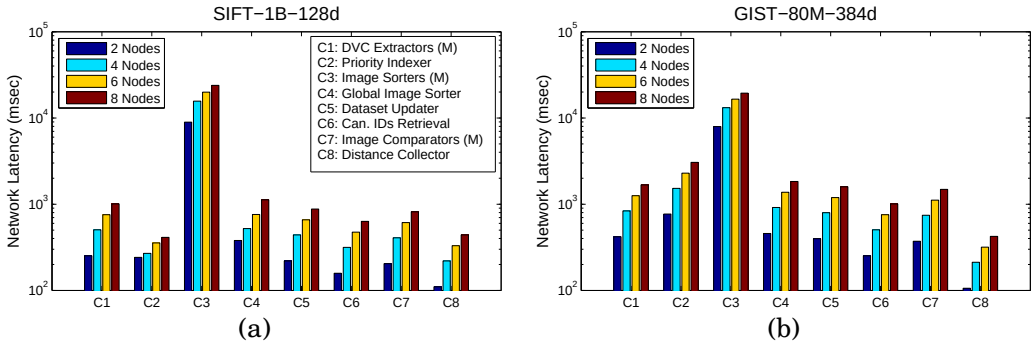


Fig. 7. Networks Latencies of the CDVC components on (a) SIFT-1B-128d and (b) GIST-80M-384d.

In Table V, we present the CPU overhead/usage of each node of the CDVC components. The CPU overhead/usage is reduced slightly (up to 4%) by increasing the number of the M nodes of the CDVC components, with the exceptional case of an *Image Sorter* node (Figure 8) whose CPU overhead/usage is highly reduced. This happens because by increasing the number of the M nodes, each *Image Sorter* node takes as input less N/M descriptor vectors to process.

In Figure 9, we present the performance of CDVC on the cloud infrastructure, including the network latencies and the CPU times of the respective CDVC components.

[18]http://hbase.apache.org/

Table V. CPU OVERHEAD/USAGE (%) OF EACH NODE OF THE CDVC COMPONENTS.

	SIFT-1B-128d	*GIST-80M-384d*
Preprocessing		
DVC Extractor	32%	36%
Priority Indexer	18%	22%
Global Image Sorter	83%	89%
Insertion		
Dataset Updater	6%	7%
Query Processing		
Candidate Images' IDs Retrieval	8%	9%
Image Comparator	80%	89%
Distance Collector	13%	15%

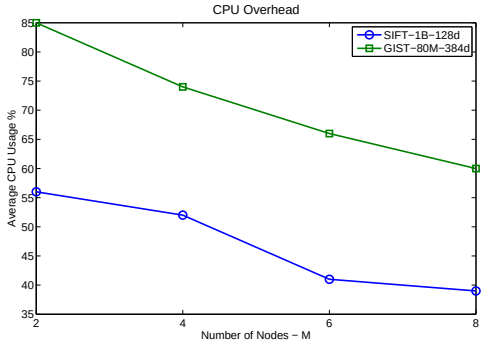
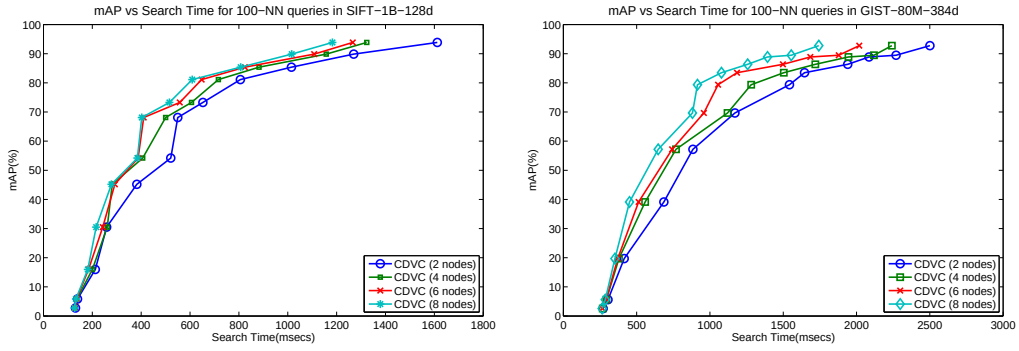


Fig. 8. (a) CPU Overhead/Usage (%) of an *Image Sorter* node in GIST-80M-384d and SIFT-1B-128d, by varying the number of the M nodes over the cloud infrastructure.

According to the workflow of our CDVC framework (Figure 1), to retrieve the result set $\mathcal{R}$ from the *CDVC Scheduler* and the *Cloud Database Server*, the required time is the aggregation of the network latencies and CPU times of *Dataset Updater*, *Candidate Images' IDs Retrieval*, *Image Comparator* and *Distance Collector*. Based on the experimental results of Figure 9, the overall search time is reduced by increasing the number of the M nodes.



(a) (b)

Fig. 9. Performance of CDVC on the cloud infrastructure for (a) SIFT-1B-128d and (b) GIST-80M-384d.

C. PERFORMANCE OF CDVC PREPROCESSING STEPS

In this Section we present the influence of each preprocessing step of CDVC to the overall preprocessing cost of Tables II and III. The total preprocessing time is calculated as the aggregation of the individual preprocessing times of the offline algorithms, i.e. (1) the M *DVC Extractor* M-Nodes (preprocessing step 1); (2) the *Priority Indexer* (preprocessing step 2); (3) the M *Image Sorter* Nodes (preprocessing step 3); and (4) the *Global Image Sorter* (preprocessing step 3). In Table VI we report the influence of each individual offline algorithm to the total preprocessing time. We can make the following observations, by increasing the M nodes, the preprocessing costs of the M *DVC Extractor* nodes, the M *Image Sorter* Nodes and the *Global Image Sorter* are decreased, analogously. This is confirmed by our respective complexity analysis of (1) and (3), where the M parameter of the nodes is in the denominator of the fraction of the respective complexity fractions, i.e. $O(N * D/M)$ and $O(D * N/M) * \log(N/M))$. On the contrary, by increasing the M nodes, the computational cost of the *Priority Indexer* is slightly increased, as expected according to our complexity analysis in (2), since the M parameter of the nodes is in the nominator of the respective complexity fraction i.e. $O(M * D) + O(D * \log D)$. However, as we experimentally show in Table VI the *Priority Indexer* has the less computational cost in the preprocessing phase.

Table VI. CPU TIME (SEC) OF CDVC PER PREPROCESSING STEP

M **Nodes**	M **DVC Extractor Nodes**	**Priority Indexer**	M **Image Sorter Nodes**	**Global Image Sorter**	**Total**
CIME-240K-64d					
2	0.317	0.135	1.038	0.63	2.120
4	0.309	0.142	0.976	0.543	1.97
6	0.27	0.159	0.883	0.509	1.821
8	0.292	0.167	0.862	0.48	1.801
CEDD-240K-144d					
2	0.501	0.169	1.275	0.898	2.843
4	0.492	0.175	1.102	0.264	2.033
6	0.46	0.183	0.846	0.371	1.86
8	0.422	0.188	0.798	0.346	1.754
SIFT-1M-128d					
2	2.907	0.155	11.087	2.338	16.486
4	2.573	0.158	7.66	2.421	12.902
6	2.11	0.165	6.99	1.407	10.672
8	1.979	0.17	6.502	1.304	9.975
GIST-1M-960d					
2	3.819	0.49	12.8	2.238	19.347
4	3.552	0.628	10.097	1.798	16.075
6	3.197	0.0.759	8.55	1.292	13.798
8	2.074	0.882	6.913	1.217	11.083
SURF-240K-5000d					
2	3.914	1.017	10.672	4.757	20.36
4	3.842	1.12	9.087	4.427	18.476
6	3.273	1.228	8.59	3.671	16.762
8	2.976	1.29	6.834	1.879	12.979
SIFT-1B-128d					
2	7.173	0.174	12.834	4.892	25.073
4	6.861	0.185	11.975	3.737	22.758
6	6.273	0.199	11.008	4.329	21.809
8	6.07	0.233	10.43	3.342	20.075
GIST-80M-384d					
2	5.973	0.299	13.972	7.44	27.684
4	5.007	0.357	12.587	7.274	25.225
6	4.22	0.366	11.29	7.856	23.732
8	3.891	0.407	9.927	7.767	21.992