This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

In-memory Stream Indexing of Massive and Fast Incoming Multimedia Content

Stefanos Antaris, *Student Member, IEEE*, and Dimitrios Rafailidis

Abstract—In this article, a media storm indexing mechanism is presented, where media storms are defined as fast incoming batches. We propose an approximate media storm indexing mechanism to index/store massive image collections with varying incoming image rate. To evaluate the proposed indexing mechanism, two architectures are used: i) a baseline architecture, which utilizes a disk-based processing strategy and ii) an in-memory architecture, which uses the Flink distributed stream processing framework. This study is the first in the literature to utilize an in-memory processing strategy to provide a media storm indexing mechanism. In the experimental evaluation conducted on two image datasets, among the largest publicly available with 80M and 1B images, a media storm generator is implemented to evaluate the proposed media storm indexing mechanism on different indexing workloads, that is, images that come with high volume and different velocity at the scale of 10^5 and 10^6 incoming images per second. Using the approximate media storm indexing mechanism a significant speedup factor, equal to 26.32 on average, is achieved compared with conventional indexing techniques, while maintaining high search accuracy, after having indexed the media storms. Finally, the implementations of both architectures and media storm indexing mechanisms are made publicly available.

Index Terms—In-memory processing, multimedia storage and search, stream processing.

1 Introduction

Approximate nearest neighbor (ANN) search over large amounts of multimedia content is an interesting and fundamental problem in multimedia analysis and retrieval [1], [2], [3]. ANN search has been widely used for image similarity search [4], [5], [6], matching local features [7], [8], and parameter estimation [9]. With the proliferation of social networks and mobile multimedia applications, ANN search has become even more demanding, as it needs to maintain high search accuracy over a massive amount of multimedia content [10]. To achieve high search accuracy, ANN search frameworks focus on two main processes: i) the efficient index/storage; and ii) the similarity search. Therefore, a serious question has been raised: how can we process and index/store massive amounts of fast incoming images? In this article, we define ***media storms*** as *"large batches of fast incoming images, typically $10^5 - 10^6$ images per second on average"* [11]. Nowadays, there are several applications that require the media storm management. For example, the continuously growing and massive image collections that come from social networks [1], like Flickr [12], a public picture sharing site, which reported that it has received 1.42, 1.6 and 1.83 million per day, on average, in 2012, 2013 and 2014, respectively, making clear that the number of incoming images will keep increasing every year [13]. Also, the explosion of the Internet of Things (IoT) with nearly 8 billion devices connected to the Internet by 2018, will increase the number of incoming images at an unexpected rate [14]. Therefore, we need a mechanism that can efficiently and reliably process and index media storms to large numbers of geographically distributed data centers. Processing media storms is computationally intensive; therefore, the challenge is to support an indexing/storage mechanism that processes the fast incoming image collections and correctly index them in a limited time to maintain high search accuracy.

Over the last decade, several frameworks of image similarity search strategies of multimedia content, also known as *image descriptors*, in distributed databases have been proposed [15], [16], [17], [18], [19]. However, these strategies are designed in a manner that a collection of descriptors is processed once and indexed in a distributed database while search queries are executed over the already processed descriptors. When new descriptors seek to insert the framework, they are indexed in a sequential fashion, as no parallelization at the indexing process is provided.

Meanwhile, indexing media storms has been studied in [1]; however, after the media storms have been indexed, the complex data structures of multiple randomized KD-trees are used, thus, having high computational cost in the online search [20]. Additionally, all the aforementioned similarity search strategies are designed as a disk-based processing mechanism, which requires a vast amount of I/O operations in order to process the incoming image collections. Therefore, this article focuses on the case of in-memory distributed stream processing of the fast incoming media storms. In order to perform in-memory distributed stream processing, several existing frameworks, such as Apache Flink [21], Apache Spark [22], Apache Storm [23], offer various libraries to efficiently process a vast amount of data in a limited time. However, to the best of our knowledge, none of the above frameworks provides a library to support stream indexing of media storms.

In-memory mechanisms become the new trend for multiple high-performance systems, such as database, Big Data analytics and real-time processing systems, where the memory becomes the primary data management source [24].

• *S. Antaris and D. Rafailidis are with the Department of Informatics, Aristotle University, 54124 Thessaloniki,Greece.*
E-mail: {santaris,draf}@csd.auth.gr

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

Shifting the data storage from disk to the memory enables the systems to achieve over 100 times better response time and throughput. However, given the resource limitations, the overall performance of in-memory processing on a distributed environment can be affected by several design choices: i) *data replication* that improves the performance of processing the same content among different nodes but allocates more memory than needed; ii) efficient *caching* algorithms that load data from the disk to the main memory to reduce the I/O latency, while preserving a copy of the cached data to the disk to guarantee data availability; and iii) *data partitioning* algorithms that minimize the required data transfers across different nodes.

Therefore, performing an in-memory distributed indexing mechanism of media storms poses the following main challenges: i) *scalability* is a required characteristic of indexing media storms in order to index a massive amount of images in a streaming mode into distributed databases and efficiently search in high accuracy; ii) indexing media storms should be performed at *low latency* and no bottlenecks should be experienced by the incoming image collections; and iii) to perform an efficient distributed mechanism, a *robust communication* messaging system is needed in order to exchange data using message passing.

1.1 Contribution and Outline

In the preliminary study [11], a parallel media storm indexing mechanism, which handles massive amounts of fast incoming descriptors, was introduced. The media storm indexing mechanism in [11] utilizes the MapReduce paradigm to parallelize the sequential indexing mechanism of the *Cloud Based on Image Descriptors' Dimensions Value Cardinalities (CDVC)* framework [20] and index the media storms in a DVC-based data structure [25]. Despite the fact that the media storm indexing mechanism in [11] substantially reduces the computational time of the CDVC's indexing mechanism, it suffers from high latency as it performs ***exact indexing***, which is formally defined as *"the indexing of each of the incoming descriptor in the exact position of the DVC-based data structure"*. Therefore, in this article, the contribution is summarized as follows: i) an approximate indexing mechanism is proposed using the MapReduce paradigm to reduce the high latency and high CPU issue of [11]; ii) two different architectures, a disk-based and an in-memory architecture, are presented in order to evaluate the benefit of the in-memory processing in the proposed mechanism, while the performance of the proposed media storm indexing mechanism is examined on different burst incoming rates within several "storming time frames", that is, the duration that the storm lasts; and iii) we verify that the media storms have been correctly indexed in the DVC-based data structure by performing search queries, outperforming state-of-the-art methods in terms of indexing, search time and accuracy.

The rest of this article is organized as follows, Section 2 reviews the related work of similarity search strategies and in-memory stream processing; Section 3 presents the preliminaries of the previously proposed DVC-based strategy and outlines the CDVC framework; Section 4 defines the problem and identifies the CPU and latency issues that the exact indexing mechanism of [11] poses; Section 5 details the two architectures to implement the approximate media storm indexing mechanism and handle the identified issues from the previous section; Section 6 evaluates the proposed mechanism on different indexing workloads; and finally, Section 7 concludes this paper and provides interesting future directions.

2 Related Work

2.1 Similarity Search Strategies in Distributed Databases

Over the last decades several similarity search strategies in distributed databases, such as M-Chord [26], RT-CAN [16], DKdt [27] and MT-Chord [19], were proposed. These similarity search strategies require the construction of complex data structures, such as R-trees [16], [19], [26], KD-trees [27], in order to allocate the multimedia data on the distributed databases. The construction and the maintenance of such complex data structures require a significant processing cost due to the high-dimensional multimedia content [20]. Therefore, the aforementioned similarity search strategies are inefficient to store and accurately searched over massive and fast incoming multimedia content.

To avoid the construction of complex data structures that cannot handle the high-dimensional data, hashing methods have been widely used [28], [29], [30], [31]. Hashing strategies are divided into data-independent [4], [32], [33] and data-dependent [6], [29], [34], [35], [36], [37] and are widely used for ANN search because of their low storage cost and fast query speed [32]. The main goal of hashing strategies, such as Spectral Hashing [6], Iterative Quantization [34] and Anchor Graph Hashing [35] is to preserve the similarities of the training data using linear or nonlinear functions when projecting the data to the Hamming space. However, the aforementioned hashing strategies provide no parallelization while indexing new descriptors and performing ANN search. The framework in [28] proposed a Multi-Index Hashing (MIH) strategy, which utilizes different hashing methods such as Locallity-Sensitive Hashing (LSH) [38] and Minimal Loss Hashing (MLH) [29] to store the descriptors M times in M different hash tables, using M disjoint binary substrings and therefore search the multimedia content in a parallel manner. Consequently, MIH's search accuracy relies on the used hashing method. However, indexing the incoming descriptors is performed in a sequential order, as MIH does not support parallelization when learning the new binary codes. Another strategy to handle the high-dimensional content of the multimedia data is the Inverted Multi-Index (IMI) [39], which utilizes the Product Quantization (PQ) strategy [40] to split the high-dimensional descriptor vectors into dimension groups and store them in inverted indexes. Although both MIH and IMI manage to efficiently handle high-dimensional multimedia data and achieve high search accuracy, no parallelization is supported during the insertion of the incoming multimedia content, thus being inefficient to index/store media storms. Our proposed indexing mechanisms not only achieve high search accuracy but also provide a parallel indexing mechanism for incoming media storms.

The necessity of parallelization in order to manage the rapidly increasing volumes of multimedia content is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

discussed in [1], where multimedia content is processed in batch using the MapReduce paradigm. The incoming descriptors are inserted in the Hadoop framework and processed in parallel. Although [1] accelerates the processing of media storms using the MapReduce paradigm, Hadoop as a substrate of the framework hinders the indexing of media storms. The HDFS memory caching mechanism [41] enables Hadoop for in-memory processing. However, Hadoop is a framework primarily for batch processing rather than stream processing. Moreover, [1] uses randomized KD-trees to maintain the data allocation of the multimedia content, the maintenance of which provides additional computational cost and makes the framework inefficient to handle the massive amount of multimedia content [20]. On the other hand, our approach provides an in-memory stream processing mechanism that indexes the new descriptor vectors in non-complex data structures in order to process massive media storms.

2.2 In-memory streaming processing

To overcome the I/O latencies that the disk-based frameworks pose and index the media storms in real-time, in-memory big data management and processing techniques are required. Over the last decade, several big data processing frameworks have been introduced that are based on the MapReduce paradigm and perform in-memory processing, such as Apache Spark [22], Apache Flink [21] and Apache Storm [23]. Apache Spark utilizes in-memory data structures, called resilient distributed datasets (RDDs), to perform big calculations by pinning memory. Apache Spark offers a microbatch stream processing model that collects the data generated over a time-window and process them in a batch form. Apache Flink utilizes the Lambda Architecture [42] to accelerate the streaming process and is optimized for cyclic or iterative processes by using iterative transformation on collections. Apache Storm architecture consists of the input stream, called spouts and the computation logic, called bolts. The stream processing frameworks offer several libraries (MLib [43] and H_2O [44] for Apache Spark; FlinkML [45], SAMOA [46] and Gelly [47] for Apache Flink; and SAMOA [46] for Apache Storm) to support machine learning and graph analysis. However, to the best of our knowledge, there is no stream processing framework that supports a mechanism for indexing the high dimensional data of media storms.

3 PRELIMINARIES

3.1 The DVC-based strategy

Definition 1 (DVC). *"The DVC of a dimension $j \in 1, \ldots, D$ is the number of unique values at the j-th dimension of all the descriptors in an image collection [11], [20], [25]."*

According to the analysis in [25], descriptors' DVC have the following three important properties:

Property 1. *"DVC distributions highly depend on the descriptors' extraction strategies [25]."*

Property 2. *"Each descriptor extraction method tends to produce similar DVC distributions for different dataset sizes; therefore, ANN search strategies that exploit DVC can scale, as the DVC distributions over the dimensions are preserved, irrespective of the dataset sizes [25]."*

Property 3. *"Descriptors' DVC have a strong impact on the ANN search strategies, both in terms of search time and accuracy [25]."*

Proposition 1. *"In the DVC-based strategy, the goal is to reorder the storage positions of descriptors according to the 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 $2W$ [25]."*

3.2 The CDVC framework

CDVC is a cloud-based similarity search framework based on the DVC-based strategy [20] that supports the following functionalities: (1) *parallel preprocessing* of datasets' descriptors for the storage management in the distributed databases over the cloud; (2) *indexing* of a new descriptor q in real-time; (3) *parallel query processing* (similarity search) in the cloud to retrieve the top-k results. In Table 1, the notation of CDVC is presented.

TABLE 1
NOTATION IN CDVC

Symbol	Description
N	dataset size
D	dimensionality of descriptors
M	number of nodes in the cloud
$\mathbf{L}$	the double linked list with the images' multi-sorted logical storage positions based on their DVC
$\mathbf{p}$	the D-dimensional priority index vector based on DVC
$\mathbf{C}$	the D-dimensional DVC vector
T	number of threads
$\mathcal{V}_{pk}$	the set of images with the same primary key pk at the first sorted dimension with the highest DVC
k	the top-k results

Overview: The preprocessing analyzes the N images, stored in the distributed databases over the cloud. After the preprocessing has finished, image queries are posed to the CDVC framework to retrieve the top-k results from the distributed databases. This is achieved within the *indexing and query processing*. The search process receives the D-dimensional descriptor $\mathbf{v}_q$, of a query image q and initially indexes it to the distributed databases. Afterwards, it compares the new descriptor query q with the already stored images and return the top-k similar image results.

Preprocessing: It takes as input a set $\mathcal{V}$ of the N D-dimensional descriptors which are stored in the distributed databases and generates a global double linked list $\mathbf{L}$ with the logical sorted positions of all N descriptors based on DVC. The preprocessing consists of four main steps: *Step #1*, the value cardinalities of a subset of dimensions of N D-dimensional descriptors based on M different predefined lower $lb^{(m)}$ and upper $ub^{(m)}$ dimensions' bounds, with $m = 1, \ldots, M$ nodes, are calculated and then M dimension value cardinality vectors $\mathbf{C}^{(m)}$ are generated; *Step #2*, M different dimension value cardinality vectors $\mathbf{C}^{(m)}$ are merged into a global dimension value cardinality vector $\mathbf{C}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

The output is a priority index 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; in *Step #3*, each of the $m = 1, \ldots, M$ nodes retrieves a subset of the descriptors $\mathcal{V}^{(m)}$, and based on the generated priority index vector $\mathbf{p}$ the logical sorting is performed, thus generating M different double linked lists $\mathbf{L}^{(m)}$, which contain the logical sorted positions of the descriptors in $\mathcal{V}^{(m)}$. The M different double linked lists $\mathbf{L}^{(m)}$ are merged into a global double linked list $\mathbf{L}$ in *Step #4*.

Proposition 2. *"The preprocessing step in the CDVC framework has the following complexity [20]*:

$$O\left(N \cdot \frac{D}{M}\right) + O(M \cdot D) + O(D \log D) + O\left(D \cdot \frac{N}{M} \cdot \log \frac{N}{M}\right) \quad (1)$$

Indexing: Given a new query $\mathbf{v}_q$, the correct position pos_q in the double linked list $\mathbf{L}$ based on the priority index vector $\mathbf{p}$ is defined and furthermore, the $\mathbf{v}_q$ is indexed in the position pos_q by updating the double linked list from $\mathbf{L}$ to $\mathbf{L}'$. This is achieved by identifying the set $\mathcal{V}_{pk}$, which is formally defined as follows:

Definition 2 (The $\mathcal{V}_{pk}$ set). *"The set of the already stored descriptors that have the same value - primary key pk with query $\mathbf{v}_q$ at the first sorted dimension with the highest DVC [20]."*:

Proposition 3. *"The complexity of the indexing step is independent of the dataset size N, and it is linearly correlated with the size $|\mathcal{V}_{pk}|$, that is, the number of the unique primary keys, and the dimensionality D of the descriptors [20]"*:

$$O(|\mathcal{V}_{pk}| \cdot \log |\mathcal{V}_{pk}|) + O(D) \quad (2)$$

It is noteworthy to mention that no parallelization is applied during the indexing step of CDVC and each new query $\mathbf{v}_q$ is indexed in a sequential order [11].

Query Processing: Given the updated double linked list $\mathbf{L}'$ and the position pos_q of the descriptor query q, a set $\mathcal{V}_{2W}$ of descriptors are generated which are located in the updated double linked list $\mathbf{L}'$, in W previous and W next to the position pos_q. The set $\mathcal{V}_{2W}$ of descriptors and the query descriptor $\mathbf{v}_q$ are then used to calculate the respective distances between $\mathbf{v}_q$ and $\mathcal{V}_{2W}$ using T parallel threads. The M different sets $d^{(m)}$ of the calculated distances are then merged to generate the top-k similar image results.

Proposition 4. *"Given the search radius $2W$ as a percentage of the dataset size N, a set of M computational nodes and T parallel threads per node, the complexity of the query processing step in CDVC is linearly correlated with the dimensionality D of the descriptors and sublinearly correlated with the dataset size N, and it is equal to [20]"*:

$$O\left(\frac{2 \cdot W}{M \cdot T} \cdot D\right) + O(k) \quad (3)$$

4 Handling Media Storms with MapReduce

This section presents how the CDVC indexing step can be adapted to fit with the MapReduce paradigm. First, the problem that the CDVC indexing confronts is defined and then moved to the details of the proposed media storm indexing mechanism. Two approaches of the media storm indexing mechanism are described: i) an exact MapReduce approach that is presented in [11] and identifies the position of the incoming descriptors in the double linked list $\mathbf{L}$; and ii) a MapReduce approach that assigns the logical position to each descriptor in an approximate manner using the notion of the ***root descriptor*** (Definition 5), in order to reduce the computational complexity and latency of the exact approach.

4.1 Problem Definition

The indexing step of an incoming descriptor $\mathbf{v}_q$ in the CDVC framework locates its logical position in the already preprocessed double linked list $\mathbf{L}$, resulting in high search accuracy and low search time [20]. However, the indexing step in the CDVC framework does not support the case of indexing many descriptors simultaneously, but indexes and processes each descriptor separately and not in a batch form. A media storm is formally defined as follows:

Definition 3 (Media Storm). *"A Media Storm is defined as a consecutive sequence of batches, where each batch contains a set $\mathcal{X}$ of incoming descriptors at a certain time step within the storming time frame, that is the duration of media storm."*

A time step equal to one second is set, having an incoming batch $\mathcal{X}$ per one second. The sizes of the batch $|\mathcal{X}|$ may vary within the storming time frame, corresponding to the real-case scenario. The challenges of indexing media storms are: (1) to preserve the CDVC's search in low time, by correctly identifying the $|\mathcal{X}|$ logical positions of the incoming images in the double linked list $\mathbf{L}$; and (2) to keep the computational cost of the media storm indexing mechanism low. The problem of indexing media storms in the CDVC framework is formally defined as follows:

Definition 4 (Problem Definition). *"The problem of indexing media storms is to correctly index a set $\mathcal{X}$ of incoming descriptors in parallel, and update the double linked list from $\mathbf{L}$ to $\mathbf{L}'$, by preserving the multi-sorted logical positions based on the priority index $\mathbf{p}$."*

4.2 Exact Indexing Mechanism with MapReduce (EMR)

In EMR, the indexing of a set $\mathcal{X}$ incoming descriptors can be divided into three main steps using two map functions and one reduce function [11]. During *Map #1*, the set $\mathcal{X}$ of the incoming descriptor vectors are split into M computational nodes, where each image descriptor vector $\mathbf{v}_q \in \mathcal{X}$ is sorted based on the priority index $\mathbf{p}$. During *Map #2*, the sorted image descriptor vectors $\mathcal{X}_{pk}$ with the same primary key pk are grouped into the same computational node $m = 1, \ldots, M$ following the MapReduce paradigm. The subset $\mathcal{X}_{pk}$ is combined with the set $\mathcal{V}_{pk}$ and has generated the double linked sublist $\mathbf{L}^{(m)}$. Finally, during the *Reduce* step, the M different double linked sublists $\mathbf{L}^{(m)}$ are collected by one computation node and are merged to update the global double linked list from $\mathbf{L}$ to $\mathbf{L}'$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

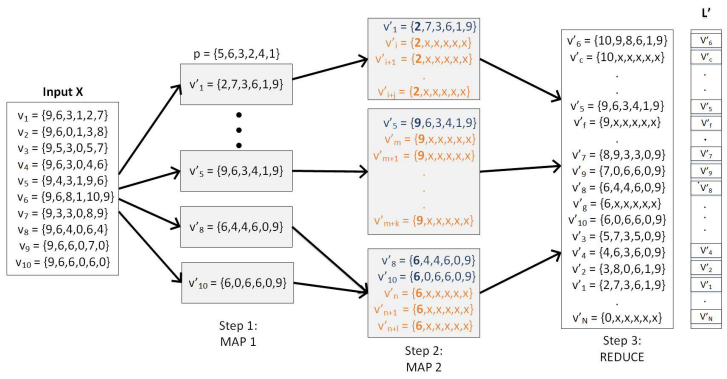

Fig. 1. Running example of the proposed media storm indexing mechanism. In the Map 2 step, blue fonts denote the sets $\mathcal{X}_{pk}$ of the incoming batch $\mathcal{X}$, while red fonts denote the already stored descriptors $\mathcal{V}_{pk}$, with the same primary key pk.

Proposition 5. *"The indexing step in EMR has the following complexity [11]"*:

$$O\left(\frac{|\mathcal{X}|\cdot D}{M}\right)+O(|\mathcal{X}|)+O\left(\frac{|\mathcal{X}|\cdot|\mathcal{V}_{pk}|}{|\mathcal{X}_{pk}|\cdot M}\right)+O(M) \quad (4)$$

Figure 1 presents a running example of the proposed media storm indexing mechanism. The input is the set of $\mathcal{X}$=10 descriptors $\mathbf{v}_i$, $i \in 1\dots10$, with D=6 dimensions and the priority index $\mathbf{p} = \langle 5,6,3,2,4,1\rangle$, following the paradigm of [11]. In the running example, ten descriptors are considered as an incoming batch at a time step of one second within the storming time frame. In *Step #1*, the dimensions of the ten descriptors are reordered according to $\mathbf{p}$, generating ten reordered descriptors $\mathbf{v}'$. In *Step #2*, according to the primary key pk, the reordered vectors $\mathcal{X}_{pk}$ are grouped with the preprocessed vectors in the respective set $\mathcal{V}_{pk}$ of the already stored descriptors. For instance, $\mathcal{X}_{pk}$=$\{\mathbf{v}'_1\}$ is grouped with $\mathcal{V}_{pk}$=$\{\mathbf{v}'_i,\dots\mathbf{v}'_{i+j}\}$, because they have the same primary key pk=2. Also, $\mathcal{X}_{pk}$=$\{\mathbf{v}'_8,\mathbf{v}'_{10}\}$ are grouped with $\mathcal{V}_{pk}$=$\{\mathbf{v}'_n,\dots\mathbf{v}'_{n+l}\}$, with the same pk=6. The respective group of descriptors are reordered, generating M different sublists $\mathbf{L}^{(m)}$. In *Step #3* of Figure 1, the M different sublists $\mathbf{L}^{(m)}$ are merged to update the double linked list $\mathbf{L}$ accordingly and to store the incoming descriptors of $\mathcal{X}$.

Although the EMR mechanism ensures the identification of the exact position in the double linked list $\mathbf{L}$ for each of the incoming descriptors $\mathbf{v}_q$, it presents two main drawbacks as stated below:

Remark 1 (High Latency). *Each of the M computational nodes needs to retrieve the subset $\mathcal{V}_{pk}$ from the distributed database in order to process the subset $\mathcal{X}_{pk}$, posing high latency when the size of the subset $\mathcal{V}_{pk}$ is large.*

Remark 2 (High CPU). *The indexing media storms is directly affected by the size of the subsets $\mathcal{X}_{pk}$ and $\mathcal{V}_{pk}$, increasing thus the comparisons occurred as the subsets $\mathcal{X}_{pk}$ and $\mathcal{V}_{pk}$ are high enough.*

4.3 Approximate Indexing Mechanism with MapReduce (AMR)

To efficiently solve the drawbacks of high latency and CPU, an approximate media storm indexing mechanism with MapReduce (AMR) is presented in Algorithm 1.

Algorithm 1: Approximate Media Storm Indexing Algorithm

Input: (1) $\mathcal{X}$, (2) $\mathbf{p}$
Output: $\mathbf{L}'$
1 **MAP 1**
2 *Input :* (1) pairs $< \mathbf{v}_i, \mathbf{p} >$, with $\mathbf{v}_i \in \mathcal{X}$, (2) $\mathbf{p}$
3 *Method*
4 $\quad \mathbf{v}'_i \leftarrow$ Sort $\mathbf{v}_i$ based on $\mathbf{p}$
5 $\quad pk \leftarrow \mathbf{v}'_{i1}$
6 $\quad$ Emit $< pk, \mathbf{v}'_i >$
7 **MAP 2**
8 *Input:* (1) pairs $< pk, \mathcal{X}_{pk} >$, (2) $\mathbf{v}^*_{pk}$
9 *Method*
10 $\quad \mathbf{L}^{(m)} \leftarrow$ sort $< \mathbf{v}'_1, \mathbf{v}'_2, \dots, \mathbf{v}'_{|\mathcal{X}_{pk}|} >$ with $\mathbf{v}^*_{pk}$
11 $\quad$ Emit <key, $\mathbf{L}^{(m)}$ >
12 **REDUCE**
13 *Input :*<key, $< \mathbf{L}^{(1)}, \mathbf{L}^{(2)}, \mathbf{L}^{(3)}, \dots, \mathbf{L}^{(m)} >$>
14 *Method*
15 $\quad \mathbf{L}' \leftarrow$ Merge $< \mathbf{L}^{(1)}, \mathbf{L}^{(2)}, \mathbf{L}^{(3)}, \dots, \mathbf{L}^{(m)} >$
16 $\quad$ Emit <key, $\mathbf{L}'$ >

Following the MapReduce paradigm, the algorithm consists of the following steps:

Map 1: The batch $\mathcal{X}$ of the incoming descriptors is divided into M computational nodes. Each node takes as input a descriptor $\mathbf{v}_i \in \mathcal{X}$ and the priority index $\mathbf{p}$ (line 2). Then, to comply with the DVC-based strategy, the descriptor $\mathbf{v}_i$ is reordered based on $\mathbf{p}$ (line 4). The value of the first dimension of the descriptor $\mathbf{v}'_{i1}$ is set as a primary key pk (line 5). The output of the Map 1 step is a pair $<pk, \mathbf{v}'_i>$, considering the primary key pk as key, and the reordered descriptor $\mathbf{v}'_i$ as value. The total complexity of the Map 1 step is $O\left(\frac{|\mathcal{X}|\cdot D}{M}\right)$.

Map 2: For each primary key pk, the m-th computational node, with $m \in 1\dots M$, fetches the sets $\mathcal{X}_{pk}$ from the Map 1 step, as well as the *root descriptor* $\mathbf{v}^*_{pk}$, which is formally defined as follows:

Definition 5 (Root descriptor vector $\mathbf{v}^*_{pk}$). *"The first ordered descriptor in the double linked list $\mathbf{L}$ that has the same primary key pk with the subset $\mathcal{X}_{pk}$."*

Thereafter, the m-th node compares and reorders the subset $\mathcal{X}_{pk}$ of the incoming descriptors along with the root descriptor $\mathbf{v}^*_{pk}$, in order to generate the double linked sublist $\mathbf{L}^{(m)}$ (line 17) in $O\left(\frac{|\mathcal{X}|}{|\mathcal{X}_{pk}|\cdot M}\right)$. Finally, the same key is set to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

all the M different computational nodes' outputs, to process all the output pairs by a single reducer.

Reduce: The M different double linked sublists $\mathbf{L}^{(m)}$, produced by the Map 2 step, are merged to update the global double linked list from $\mathbf{L}$ to $\mathbf{L}'$, thus indexing the incoming descriptors of set $\mathcal{X}$, in $O(M)$.

Proposition 6. *"Summarizing, the total complexity of the AMR is"*:

$$O\left(\frac{|\mathcal{X}| \cdot D}{M}\right) + O(|\mathcal{X}|) + O\left(\frac{|\mathcal{X}|}{|\mathcal{X}_{pk}| \cdot M}\right) + O(M) \quad (5)$$

AMR indexes the incoming descriptors in a position relatively close to the root descriptor $\mathbf{v}^*_{pk}$ in the double linked list $\mathbf{L}$, instead of the exact position that the EMR assigns using the whole set $\mathcal{V}_{pk}$. The benefits of AMR against EMR are the following:

Remark 3 (Low Latency). *The M computational nodes retrieve from the distributed databases only the root descriptor, $\mathbf{v}^*_{pk}$, that substantially reduces the latency when comparing with EMR that retrieves the whole subset of $\mathcal{V}_{pk}$ (Remark 1).*

Remark 4 (Low CPU). *The subset $\mathcal{X}_{pk}$ is compared and re-ordered only with the root descriptor $\mathbf{v}^*_{pk}$ and indexing media storms is independent of the size of the subset $\mathcal{V}_{pk}$.*

5 Underlying Architectures

The proposed media storm indexing mechanism AMR is designed to be executed in two different architectures, following the disk-based (baseline) and the in-memory (Flink) strategy. In this section we present the two architectures that we have used to implement the EMR and AMR media storm indexing mechanisms.

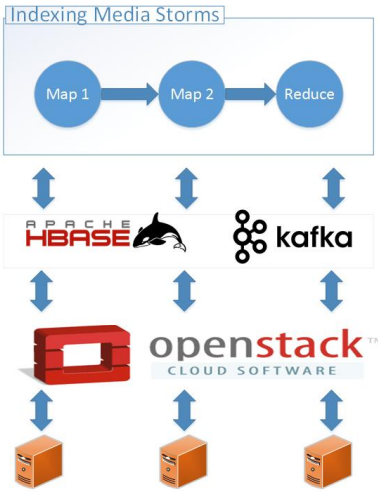

Fig. 2. Baseline architecture

5.1 Baseline Architecture

It is a disk-based architecture of AMR (Section 4.3), built over a cloud infrastructure. Figure 2 shows that the cloud infrastructure is orchestrated using the OpenStack [48] free and open-source computing software platform to create the Virtual Machine (VM) computational nodes. Moreover, Apache Kafka [49] was installed to provide an efficient and robust messaging communication between the VM computational nodes, and Apache HBase [50] for distributed data storage in order to retrieve and store the data in a distributed manner.

In the baseline architecture, each VM computational node is responsible for the execution of a different AMR step. Following this allocation strategy, the baseline architecture addresses the three challenges described in Section 1 as follows:

- The *scalability* of the baseline architecture is ensured by assigning more VM computational nodes to a specific step of AMR that requires a computationally intensive task. By allocating more VM instances in a specific step of AMR we achieve low CPU cost and indexing media storms time.
- The *low latency* is preserved based on the AMR mechanism used to index media storms. As each VM computational node at the Map 2 step requires the retrieval of only one image descriptor vector $\mathbf{v}^*_{pk}$ from the distributed database, we achieve low latency and avoid causing bottleneck during the indexing of media storms.
- The *robust messaging communication* between the VM computational nodes is established based on a messaging protocol, namely the Advanced Message Queuing Protocol (AMQP) [51]. As AMQP provides a reliable and scalable solution for the VM computational nodes' communication, message queues are installed between the respective computational nodes at each AMR step.

Distributed database HBase: The baseline architecture requires a very high throughput and cheap and elastic storage that presents low latency. Although modern distributed databases such as Cassandra [52], Redis [53], etc. present high performance results, HBase [50] is selected in the baseline architecture for the following reasons: (1) *High read/write throughput*: Indexing media storms mechanism requires high aggregate read/write throughput to store and retrieve a massive amount of descriptors form the distributed databases. HBase has higher read/write throughput than other distributed databases, thus reducing the total latency cost of AMR [54]; (2) *Data Consistency and Disaster Recovery*: Indexing media storms requires high data availability and update of the storage data centers. HBase achieves strong consistency for both read and write, while tolerating the data center failure, using data replication [55]; (3) *Scalability and Elasticity*: Indexing media storms has to store and maintain a large amount of descriptors at an unexpected rate. HBase performs stability while adding incremental capacity to the storage systems with minimal overhead. In the cases of adding capacity rapidly, HBase automatically balances load and utilization across the new hardware with no downtime [54].

Messaging Protocol Apache Kafka: Baseline architecture requires a *robust communication* protocol for the communication between the VM computational nodes. Messaging between different components is required to deliver the data safely and at scale. The reasons for selecting Apache Kafka against other popular distributed publish/subscribe mes-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

saging systems, such as RabbitMQ [56], ActiveMQ [57], etc. are summarized as follows: (1) *High throughput*: Indexing media storms requires the transfer of event data (descriptors) between different VM computational nodes with extremely high velocity. Apache Kafka offers high throughput over 1 million events per second, by grouping the event data in small batches of 1-20 events [58]. (2) *Scalability*: When indexing media storms it is critical to increase the number of VM instances according to the data size. This means that the VM computational nodes cannot be coupled directly. Kafka is designed as a distributed system that provides asynchronous decoupling and hence it is able to easily scale out, reducing the complexity of integration when new resources are added [59]. (3) *Failure recovery*: Indexing media storms requires a robust communication service, even in the case of a VM failure. Apache Kafka persists messages on the VM computational nodes' disk and thus can be used for consumption when a VM is recovered. Moreover, Kafka offers replication strategies in order to ensure high data availability when a failure occurs.

Remark 5 (I/O bottleneck). *"AMR (Section 4.3) is designed in a fashion of processing large amounts of data in a small amount of time. Therefore, accessing the distributed databases every time a media storm occurred limits the performance of the proposed approach because of the additional I/O latency during the retrieval process. Moreover, the new indexed descriptors participate in the indexing process of the next media storm, thus making the proposed approach an iterative process unable to support the high access latency to the distributed databases."*

5.2 Flink Architecture

To meet the strict real-time requirements for analyzing mass amount of media storms and indexing the incoming descriptors within milliseconds, an in-memory strategy that keeps the data in the random access memory (RAM) all the time is necessary. As described in Section 5.1, the baseline architecture fails short to fit the demands of a stream processing architecture with low latency due to its dependency on the indexing/retrieval operation to/from the distributed databases. In contrast, current stream processing frameworks, such as Apache Flink [21], Apache Spark [22] and Apache Storm [23] provide an in-memory processing service that keeps the inbound data in the cluster's memory, and hence reduces the latency in the data transfers.

In this study, Apache Flink is selected for the in-memory stream processing service against other stream processing frameworks for the following reasons: (1) *Data Consistency*: Flink provides exactly-once guarantees to state updates, in order to avoid duplicates [60]. (2) *High-throughput*: Flink has a high-throughput engine that controllably buffers events before sending them over the network [60], [61]. (3) *Low-latency*: Flink achieves low latency by using a checkpointing mechanism based on Chandy-Lamport rather than processing data in micro-batches [62]. (4) *Message delivery guarantees*: Flink provides states in stateful operators in order to be correctly restored after a failure occurred [62]. (5) *In-memory processing*: Flink automatically allocates 70% of the free heap space so data can be retained in the random access memory (RAM) for the operators execution and the in-memory operations are maximized [63]. (6) *Scalability*: Flink utilizes the distributed databases as a secondary storage space where the least recently data are stored to overcome any memory resource limitations [63].

Apache Flink dataflow: Apache Flink is an in-memory streaming dataflow engine that provides data distribution, communication and fault tolerance for distributed computations over data streams. When submitting a program to a Flink cluster, a client compiles and preprocess the program to produce a Directed Acyclic Graph (DAG) using the Nephele execution engine [64]. A DAG is a tree representation of the operations that should be applied to the data set. Each node of the DAG represents an operator (e.g. Map, Reduce, Join, etc.) and each edge of the DAG represents the data flow over the operators. Using this DAG, Flink is able to identify which data should be retained in-memory since it will be needed by the next operator, thus reducing the additional I/O bottleneck.

Figure 3 shows the Flink architecture, where media storms are inserted to our Flink service using the Apache Kafka messaging service and processed by creating operators similar to the MapReduce paradigm described in Section 4. Specifically, in Flink job, three main steps are defined as follows: (1) *Map 1*: the dimensions of the incoming descriptors are reordered based on the priority index $\mathbf{p}$, (2) *Map 2*: the reordered image descriptor vectors are processed and compared with the already stored image descriptor vectors, creating the double linked sublists $\mathbf{L}^{(m)}$, and (3) *Reducer*: the sublists are merged defining the location of the incoming descriptors.

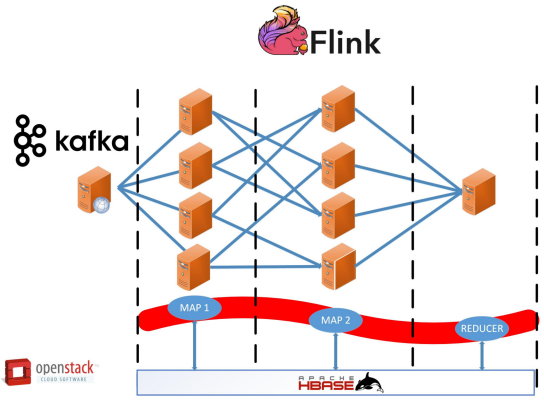

Fig. 3. Flink architecture

Flink Parameters: Apache Flink requires parameter configuration, essential for the performance of the executed operations. The most important parameters are: (1) *Number of VMs*: When building a Flink instance to a Cloud infrastructure, it is necessary to define the number of VMs that will host the Flink's master and worker nodes. This parameter is essential for the parallelization of the executed service. (2) *Number of Task Slots*: Since modern VMs provide multicore CPU resources, it is necessary to define the number of Task Slots that each VM is able to host. Thus, one VM in our cluster is able to execute more than one task at the same time and exploits all the CPU-cores that are available, reducing the total execution time of the Flink service.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

6 EXPERIMENTS

6.1 Evaluation setup

Datasets: The experimental evaluation is performed on GIST [65] of N=80M GIST descriptors of 384 dimensions and SIFT [66] of N=1B images of SIFT descriptors of 128 dimensions. To the best of our knowledge, these datasets are among the largest publicly available image collections.

Evaluation Metrics: The following metrics are used to measure the performance of the examined strategies:

- ***Indexing time***: which consists of the *CPU time* and *Latency*; where *CPU time* is the time that the media storms are processed in order to identify their position in the double linked list $\mathbf{L}$ and *Latency* is the time that the media storms spend on the messaging service and the time spend waiting for the already stored image descriptor vectors to be retrieved from the distributed databases.
- ***Search time and Accuracy***: where *Search time* is the time required to retrieve the top-k results after indexing/storing a query descriptor $\mathbf{v}_q$, and the *Accuracy* is measured according to the average results by a set of test queries $mAP = |\mathcal{R}_{seq} \cap \mathcal{R}_{ind}|/k$, with $\mathcal{R}_{seq}$ being the set of the top-k results, retrieved by the sequential search and $\mathcal{R}_{ind}$ being the set of the top-k results retrieved by the examined search strategy.

Roadmap: In Section 6.2, we present the media storm generator to evaluate the performance of the indexing mechanisms. Next, three set of experiments are conducted: *Experiment#1:* Section 6.3 demonstrates the performance of the baseline and Flink architectures (Section 5) using the EMR (Section 4.2) and AMR (Section 4.3) indexing mechanisms. *Experiment#2:* Section 6.4 analyzes the impact of the Flink parameters on the performance of the proposed indexing mechanism. *Experiment#3:* Section 6.5 compares the media storm indexing mechanism with the state-of-the-art approaches in terms of indexing, search time and mAP accuracy.

6.2 Media Storm Generator

Media storms are measured in terms of ***Incoming Image Rate (IIR)***, which is defined as follows [11]:

Definition 6 (IIR). *"Incoming Image Rate (IIR) is the number of incoming images per second."*

To evaluate the performance of the proposed media storm indexing mechanism, 80% of each dataset were initially preprocessed, and the remaining 20% is considered as the media storm size, corresponding to 200M and 16M descriptors for SIFT and GIST, respectively.

The incoming image rates vary in real-world applications, where bursts of incoming image rates affect the indexing performance of our mechanism. Hence, a media storm generator is implemented by varying IIR according to the: (i) Exponential, (ii) Logarithmic, (iii) Random, and (iv) Normal distributions. The reason for selecting these distributions is their different characteristics, such as different IIR bursting frequency and magnitude. Also, the storming time frame is varied, that is, the media storm duration, in 1, 2, 3, 4 and 5 minutes. This way, different indexing workloads were generated to test the limits of the proposed media storm indexing mechanisms on the baseline and Flink architectures.

Figure 4 shows the generated indexing workloads for the different distributions and the media storm durations. Table 2 presents the average IIR and the standard deviations of both the SIFT and GIST datasets for different distributions and storm durations. The Normal distribution generates a more intensive media storm workload at every storming time frame, as our indexing mechanism receives more descriptors on every second than the other distributions. Specifically, the highest IIR picks/bursts, presented in Figure 4, of the Normal distribution in a 1-minute storming time frame is 16×10^6 for SIFT (11×10^5 for GIST respectively). In contrast, the highest IIR picks/bursts of the Exponential distribution is limited to 15×10^6 for SIFT (7×10^5 for GIST respectively), whereas for the Logarithmic distribution is limited to 3.8×10^6 for SIFT (3.2×10^5 for GIST respectively) and for the Random distribution is 9.9×10^6 (4.6×10^5 respectively).

TABLE 2
IIR FOR DIFFERENT DISTRIBUTIONS AND STORMING TIME FRAMES IN THE SIFT AND GIST DATASETS.

Dataset	Storming Time Frame (min)	Distribution: Exponential	Logarithmic	Random	Normal
SIFT	1	3.33 ± 4.52	3.33 ± 0.45	3.33 ± 2.21	3.33 ± 5.18
	2	1.66 ± 2.42	1.66 ± 0.37	1.66 ± 1.75	1.66 ± 2.54
	3	1.11 ± 1.53	1.11 ± 0.27	1.11 ± 0.31	1.11 ± 1.67
	4	0.83 ± 1.12	0.83 ± 0.18	0.83 ± 0.14	0.83 ± 1.25
	5	0.66 ± 0.89	0.66 ± 0.09	0.66 ± 0.20	0.66 ± 1.02
GIST	1	2.66 ± 2.42	2.66 ± 0.45	2.66 ± 0.93	2.66 ± 3.72
	2	1.33 ± 1.36	1.33 ± 0.35	1.33 ± 0.36	1.33 ± 1.93
	3	0.89 ± 0.98	0.89 ± 0.26	0.89 ± 0.22	0.89 ± 1.23
	4	0.66 ± 0.79	0.66 ± 0.18	0.66 ± 0.20	0.66 ± 0.94
	5	0.53 ± 0.56	0.53 ± 0.09	0.53 ± 0.20	0.53 ± 0.77

6.3 Comparison of Underlying Architectures

In this study, the benefits of the proposed media storm indexing mechanism are evaluated using the two architectures described in Section 5:

- ***Baseline-AMR***: is the disk-based architecture built on the cloud infrastructure (Section 5.1) that implements the AMR indexing mechanism (Section 4.3).
- ***Flink-AMR***: is the in-memory architecture (Section 5.2) built on the Flink stream processing framework that implements the AMR indexing mechanism (Section 4.3).
- ***Flink-EMR***: is the in-memory architecture (Section 5.2) built on the Flink stream processing framework that implements the EMR indexing mechanism (Section 4.2).

The implementations of our media storm indexing mechanisms using both the baseline and the Flink architectures are made available at:

https://github.com/stefanosantaris/StormLibrary

Cloud Configuration: As presented in Section 5, the OpenStack cloud computing software platform for the real cloud infrastructure supports the default five different types of VM instances, that is, tiny, small, medium, large and xlarge, depending on the amount of resources that each VM reserves [67]. In our experiments, 62 distinct VM instances (Table 4) were used. These instances were deployed on 11 Dell PowerEdge servers, each with 2 Intel Xeon X5660 (24 cores), 40GB of RAM and 2TB of SSD storage. We used

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

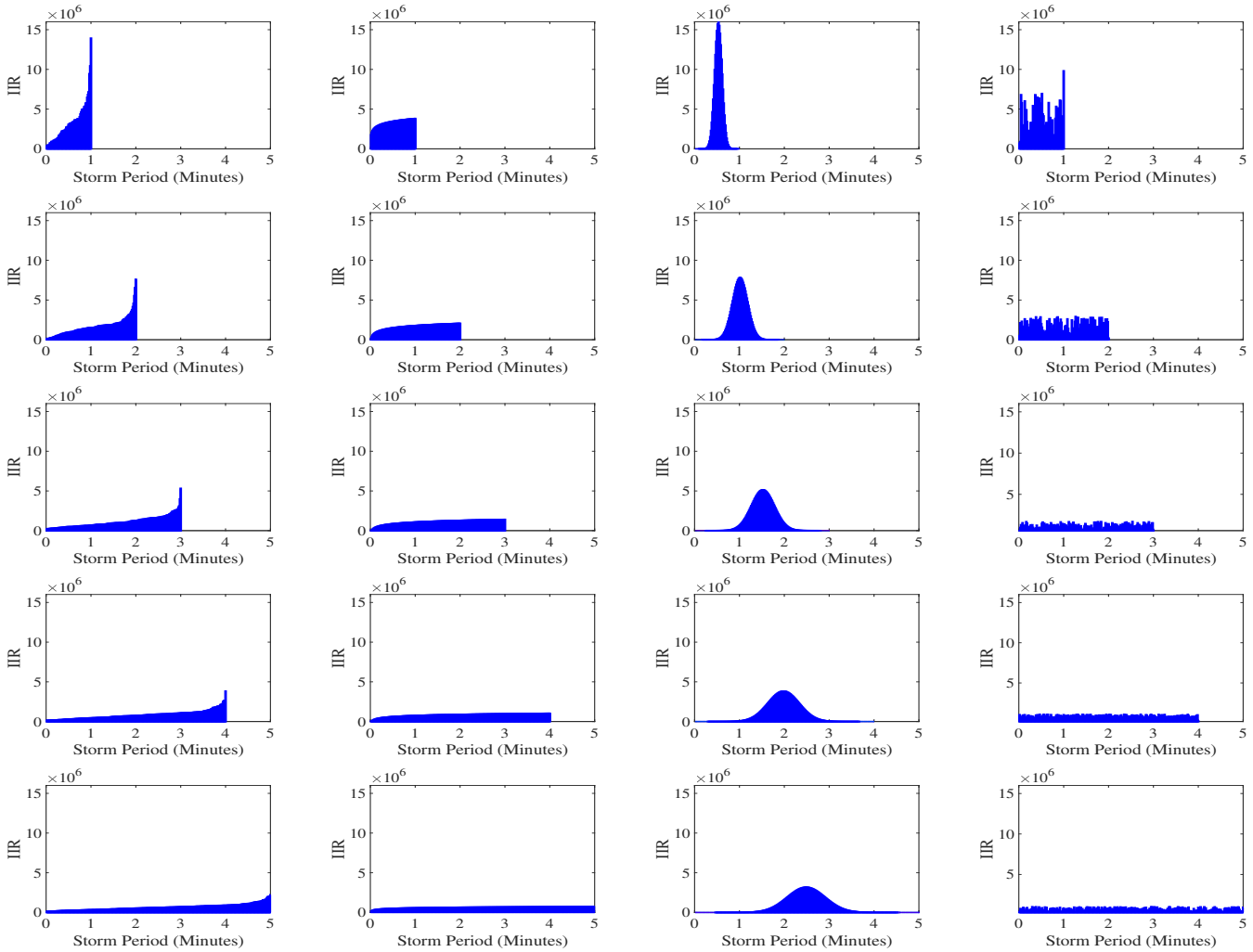

Fig. 4. IIR in SIFT-1B ($\times 10^6$ incoming images per second). For constrained storming time frames the indexing workloads are significantly increased.

TABLE 3
TOTAL INDEXING TIME FOR DIFFERENT DISTRIBUTIONS AND STORMING TIME FRAMES.

Dataset	Storming Time Frame(min)	Architecture	Distribution			
			Exponential	Logarithmic	Random	Normal
SIFT	1	**Baseline-AMR**	2,322	1,843	1,547	2,743
		Flink-EMR	1,039	988	958	1,057
		Flink-AMR	528	507	503	621
	2	**Baseline-AMR**	2,034	1,689	1,449	2,349
		Flink-EMR	910	784	805	909
		Flink-AMR	489	442	429	548
	3	**Baseline-AMR**	1,886	1,518	1,314	1,804
		Flink-EMR	715	595	734	820
		Flink-AMR	404	361	402	506
	4	**Baseline-AMR**	1,128	1,127	1,193	1,627
		Flink-EMR	592	496	695	638
		Flink-AMR	387	343	386	407
	5	**Baseline-AMR**	847	897	831	1,041
		Flink-EMR	518	447	556	566
		Flink-AMR	344	323	312	381
GIST	1	**Baseline-AMR**	1,323	1,214	1,128	1,562
		Flink-EMR	951	911	1,006	1,030
		Flink-AMR	489	426	410	531
	2	**Baseline-AMR**	1,301	1,023	962	1,434
		Flink-EMR	734	722	896	912
		Flink-AMR	435	408	385	482
	3	**Baseline-AMR**	1,261	913	804	1,122
		Flink-EMR	637	613	764	750
		Flink-AMR	393	345	339	399
	4	**Baseline-AMR**	1,084	847	759	1,023
		Flink-EMR	477	431	701	619
		Flink-AMR	329	331	321	375
	5	**Baseline-AMR**	752	646	697	853
		Flink-EMR	450	415	661	521
		Flink-AMR	314	309	306	334

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

TABLE 4
CLOUD CONFIGURATION.

# of VMs	Type of VM	# of Cores	RAM	Disk	Usage
30	Large	4	8 GB	80 GB	Computation
2	Small	1	2 GB	20 GB	Queue
30	Large	4	8 GB	80 GB	Storage (HBase)

the OpenStack Folsom Release, while the architecture for the setup was similar to the Amazon Web Services (AWS), that is, Elastic Compute Cloud for computing nodes; Simple Storage Service for storage; Amazon Machine Images for image services; and Elastic Block Store for block storage. The HBase distributed data storage was installed in 30 large VM instances to retrieve and store the data in a distributed manner. In addition, two small VM instances were utilized to install the Apache Kafka service. The computational power was distributed in 30 large VM instances, corresponding to the M=30 computational nodes of CDVC. Hence, a total of 242 Cores, 484 GB RAM and 4.84 TB Hard Disk Storage were utilized. To run our experiments on the Flink distributed streaming framework, a Flink cluster was deployed on the same 30 large VM instances as the M=30 computational nodes. Large VM instances were used for the M computational nodes in order to increase the parallelization on each VM instance, with each large VM instance having four CPU cores. To monitor the indexing times (milliseconds), several timestamps were added on each incoming descriptor. Moreover, to calculate the total indexing time on Flink, the Monitoring Rest API was utilized [68].

In Table 3, the total indexing time is presented when different distributions and storming time frames are used. We can make the following observations:

– ***Performance on different time frames and distributions***: In all architectures, the total indexing time is decreased for larger storming time frames, as IIR is reduced. Moreover, it is observed that the total indexing time is higher when Normal distribution is applied, as it has the higher indexing workload than the Logarithmic, Random and Exponential distributions.

– ***Comparison of Baseline-AMR against Flink-AMR***: The total indexing time of Flink-AMR presents 3.72, 3.55, 3.18 and 3.8 speedup factor against Baseline-AMR for the SIFT dataset and for the Exponential, Logarithmic, Random and Normal distributions, respectively (2.92, 2.53, 2.45, 2.8 for the GIST dataset). This happens because Baseline-AMR utilizes a disk-based strategy as it retrieves an already stored descriptor every time a new descriptor is indexed (Remark 1), while Flink-AMR maintains the descriptor in memory (Remark 3), thus reducing the additional I/O latency.

– ***Comparison of Flink-AMR against Flink-EMR***: When comparing Flink-EMR with Flink-AMR it is observed that the total indexing time is decreased by 70.66%, 73.94%, 50.64%, 79.72% when Flink-AMR is applied for the SIFT dataset and for the Exponential, Logarithmic, Random and Normal distributions, respectively (56.44%, 42.2%, 7.38%, 44.57% for the GIST dataset). Moreover, Flink-AMR outperforms Flink-EMR, as the incoming descriptors $\mathbf{v}_q$ in Flink-AMR are compared only with the respective root descriptor $\mathbf{v}^*_{pk}$, instead of the subset $\mathcal{V}_{pk}$ of Flink-EMR (Remark 4). Thus, Flink-AMR has lower *Latency* and *CPU* time.

– ***CPU times on different time frames and distributions***: As the storm time frame increases, the total indexing time between the Flink-EMR and Flink-AMR mechanism is decreased by a 61.03% on average for the Exponential, Logarithmic, Random and Normal distributions for the SIFT dataset (62.92% on average for the GIST dataset). This occurred as Flink-AMR is parameter free, that is, it is only affected by the IIR. Therefore, each incoming descriptor $\mathbf{v}_q$ in Flink-AMR is compared only with the root descriptor $\mathbf{v}^*_{pk}$, and thus the CPU time equals 4.105 and 8.091 milliseconds for the SIFT and GIST datasets, respectively. In contrast, Flink-EMR, which compares the incoming descriptor $\mathbf{v}_q$ with the whole set $\mathcal{V}_{pk}$, not only is affected by the IIR but also by the number of comparisons between the incoming descriptors and the set $\mathcal{V}_{pk}$ that are accomplished during the media storms. Thus, the average CPU time for indexing each incoming descriptor is 112.172 and 183.123 milliseconds, for the SIFT and GIST datasets, respectively. Increasing the time storm frame, Flink-EMR performs lower number of comparisons every second and the total indexing time is reduced significantly. We would like to mention that in both Flink-AMR and Flink-EMR the average CPU time for each independent indexing is not affected by the distribution used to generate the indexing workload, since each new descriptor is compared with the same number of already stored descriptors, that is one (with the root descriptor $\mathbf{v}^*_{pk}$) and $|\mathcal{V}_{pk}|$ for Flink-AMR and Flink-EMR, respectively.

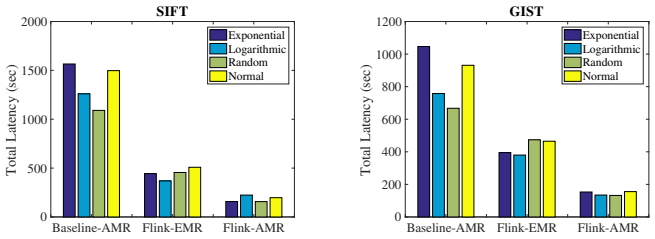

Fig. 5. Total latency using a 3 minute storming time frame.

Figure 5 presents the impact of the latency in the proposed indexing mechanisms using a 3 minute storming time frame. The following observations were noted:

- Baseline-AMR presents higher latency than both the Flink-AMR and Flink-EMR, as it is a disk-based architecture. Since Baseline-AMR is not designed as an in-memory execution job, 83% of the total indexing time is spent in latency, such as transfer the media storm through the Apache Kafka service and waiting the already stored root descriptor to be retrieved from the HBase distributed database.
- As Flink is an in-memory processing architecture, the already retrieved image descriptor vectors are retained in the VMs RAM, reducing the additional latency of the HBase read operation for the next descriptor. Thus, the network latency constituted the 62% of the total indexing time of Flink-AMR and Flink-EMR.
- Flink-AMR presents lower latency than Flink-EMR. This happens because Flink-AMR requires the retrieval of only the root descriptor $\mathbf{v}^*_{pk}$ from the distributed database, thus reducing the latency to 39% of the total indexing time on average.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

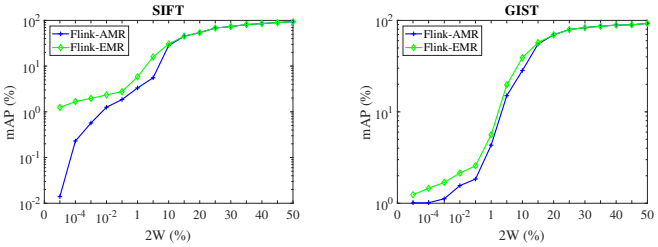

Fig. 6. Search accuracy mAP by varying the search radius $2W$.

In Figure 6, the search accuracy for both the Flink-AMR and Flink-EMR indexing mechanisms is presented, varying the search radius $2W$. Figure 6 shows that Flink-AMR has lower mAP than the exact approach when the search radius $2W$ is extremely low. However, as the search radius increases, the Flink-AMR indexing mechanism presents similar search accuracy with Flink-EMR. This happens because the incoming descriptors of media storms are relatively close to their exact position in the double linked list $\mathbf{L}$ that Flink-EMR would assign, for a larger search radius $2W$.

6.4 Flink Parameter Sensitivity

Next, we evaluate the performance of Flink-AMR by varying the following Flink parameters: the number of VMs and the number of task slots on each VM (Section 5.2).

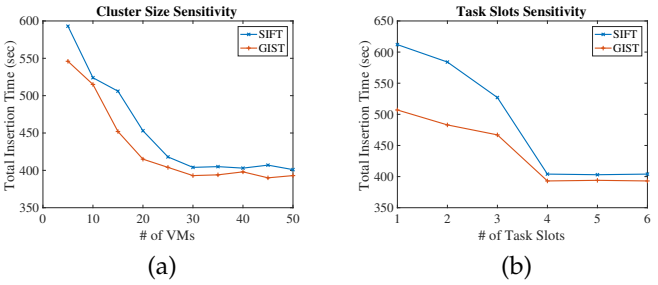

Fig. 7. Flink parameter sensitivity on media storm indexing mechanism

Figure 7(a) presents the total indexing time when different number of VMs are used for the Flink-AMR indexing mechanism. As the number of VMs increases, the total indexing time is linearly decreases. However, it is obvious that increasing the number of VMs over 30, the total indexing time of Flink-AMR is not further decreased. This occurs due to the grouping process that is applied in the Map 2 step based on the primary key pk of each descriptor. Specifically, each VM in the Map 2 step retrieves the subset $\mathcal{X}_{pk}$ of descriptors with the same primary key pk. Thus, the number of VMs depends on the number of primary keys that exist in the respective media storm. Each VM is a multi-core instance with 4 CPU-cores, able to receive four different subsets of $\mathcal{X}_{pk}$. Increasing the number of VMs more than 30 computational nodes has no effect on the performance of the media storm indexing mechanism, as the primary keys of the reordered descriptors from the Map 1 step are less than 120 and no additional computational node is needed.

Figure 7(b) reports the total indexing time by varying the number of task slots on each VM computational node. The number of task slots defines the number of jobs that each VM can execute in parallel. It is observed that increasing the number of task slots in the multi-core VMs more than four does not provide any substantial improvement to the total indexing time, as each VM has four CPU-cores (Section 6.3).

6.5 Comparison with State-Of-The-Art

In this set of experiments, the following methodologies are compared:

- ***Flink-AMR***: (Section 4.3): the $2W$ search radius was varied as in the experiments of Section 6.3.
- ***Flink-EMR***: (Section 4.2): similarly, the search radius was varied as in Flink-AMR.
- ***Multi-Index Hashing (MIH)***: M=8 hash tables were set, corresponding to the number of CPU cores and MLH as the default hashing method to compute the hash bits. The number of bits was varied in 8, 16, 32 and 64 because the implementation of MIH caused memory overflows for larger number of bits.
- ***FLANN***: the randomized KD-trees algorithm was used with a 2^7 branching factor, by varying the number of randomized KD-trees in $(2^5, 2^6, 2^7, 2^8, 2^9, 2^{10})$ and setting M=8 CPU-cores to search in parallel over the constructed KD-tree.
- ***Inverted Multi-Index (IMI)***: the list length was varied in $(2^8, 2^{12}, 2^{16})$ using a codebook with size 2^{14}.

As MIH, FLANN and IMI are not implemented to work in a cloud infrastructure, for fair comparison, all the experiments of the aforementioned methodologies were conducted on a single 8-core machine of 3.3 GHz CPU with 18 GB main memory, while the descriptors were stored in HBase similar to our approaches. In this set of experiments, only the CPU time excluding the latency times is reported. Although MIH, FLANN and IMI's implementation support parallel search, the indexing of the incoming descriptors is performed in a sequential manner. Therefore, in this set of experiments, a smaller media storm of 1K incoming descriptors/test queries is considered which have to be indexed in a batch.

Table 5 presents the total indexing time of the aforementioned methodologies for the 1K test queries. It is observed that Flink-AMR achieves 51.99, 8.89 and 25.43 speedup factor against IMI, MIH and FLANN for the SIFT dataset, respectively (24.51, 6.26, 40.82 for the GIST dataset). This happens because IMI, MIH and FLANN provide no parallelization during the indexing of media storms. In contrast, Flink-EMR and Flink-AMR not only provide parallelization but are also parameter-free in terms of indexing. Flink-AMR achieves 2.39 and 2.35 speedup factor against Flink-EMR, as Flink-EMR compares the test query $\mathbf{v}_q$ with the subset $\mathcal{V}_{pk}$ of the already stored descriptor, while Flink-AMR requires only one comparison of $\mathbf{v}_q$ with the root descriptor $\mathbf{v}^*_{pk}$.

In Figure 8, the search accuracy is reported after the media storms have been indexed. This is to evaluate if the descriptors have been properly indexed. With respect to the online query processing performance, Flink-AMR achieves high mAP in low search time. It can also be observed in Figure 6, for small values of the search radius $2W$, the mAP accuracy is lower than Flink-EMR. However, as the search radius increases, Flink-AMR maintains equal search

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2697441, IEEE Transactions on Big Data

TABLE 5
TOTAL INDEXING TIME IN SECONDS FOR THE 1K TEST QUERIES. BOLD DENOTES THE BEST METHOD.

Method	Parameter	Dataset SIFT	Dataset GIST
Flink-AMR	-	**35.123**	**42.192**
Flink-EMR	-	84.037	99.275
IMI	**List Length**		
	2^8	1826.283	1034.313
	2^{12}	1937.937	1376.348
	2^{16}	2384.762	1772.183
MIH	**# of bits**		
	8	312.383	264.317
	16	367.274	298.193
	32	426.487	327.192
	64	887.294	674.391
FLANN	**# of KD-Trees**		
	2^5	893.217	1722.201
	2^6	1078.402	2006.233
	2^7	2344.492	2228.017
	2^8	3567.451	3479.182
	2^9	4006.411	3614.726
	2^{10}	5232.114	5278.273

accuracy to Flink-EMR. Both Flink-AMR and Flink-EMR present higher search accuracy in lower search time against the competitive similarity search strategies. For instance, despite the fact that the search time of MIH is low, MIH preserves the limited mAP accuracy of the MLH hashing method. FLANN searches the complex structures of randomized KD-trees in parallel, making thus the search time high for many randomized KD-Trees.

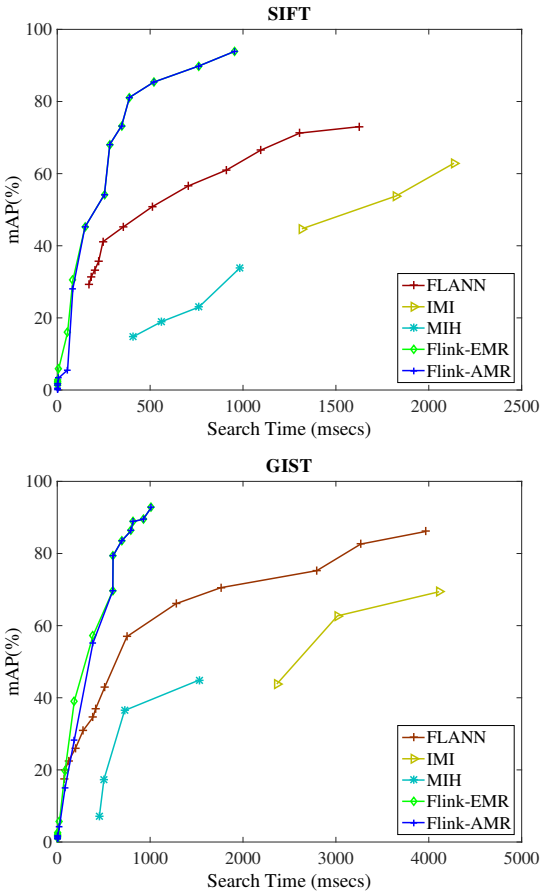

Fig. 8. Comparison of Flink-AMR against baselines in SIFT and GIST.

6.6 Discussion

The experimental results showed that it is essential to provide an efficient indexing mechanism in order to index a massive amount of media storms generated by the modern multimedia applications. To evaluate the performance of EMR and AMR, large-scale experiments were conducted on a real-cloud environment using two image datasets, which are among the largest publicly available, with 80M and 1B images. Media storms with high volume and different velocity were generated at the scale of 10^5 and 10^6 incoming images per second. Moreover, two different architectures were used in order to identify the benefit of the in-memory processing against the disk-based processing. This means that for the disk-based processing, the stored descriptors can be retrieved from the distributed database when a new descriptor is received; while for the in-memory processing, the most frequent descriptors are maintained in the memory of each VM computational node. These experiments showed that the in-memory processing using the Flink distributed stream processing framework accelerates the total indexing time, achieving a speedup factor equal to 3.12 on average. Moreover, since we retain the most frequent descriptors in the memory, while the rest of the descriptors are stored in the distributed databases, our media storm indexing mechanisms are able to scale, overcoming the memory resource limitations of the distributed environment.

Meanwhile, the EMR mechanism presents *high latency* and *high CPU* cost because of the number of descriptors which have to be retrieved from the distributed databases and the number of required comparisons to index the incoming descriptors. In contrast, the AMR mechanism overcomes the *high latency* and *high CPU* issues of EMR, by following an approximate strategy. AMR minimizes the number of descriptors retrieved from the distributed databases and compared with the incoming descriptor, achieving a speedup factor of 2.37 on average in the two evaluation datasets.

Regarding the search accuracy, the proposed AMR media storm indexing mechanism indexes the incoming descriptors efficiently in order to preserve the high search accuracy of EMR. Instead of using a subset of descriptors with the same primary key, the approximate mechanism of AMR uses a root descriptor and indexes the incoming descriptors in a relative position that is close to the position that the exact mechanism of EMR indexes. As a result, AMR presents lower search accuracy than EMR when the search radius $2W$ is extremely low, while they have similar search accuracy for high search radius $2W$.

However, this study showed that both the EMR and AMR mechanisms outperform the baseline methodologies, in terms of indexing, search time and accuracy. For example, despite the fact that IMI, MIH and FLANN support the indexing of incoming descriptors, they present extremely low performance when new descriptors are coming as media storms since they perform sequential indexing. It was observed that AMR outperforms the baseline methodologies and achieves an average speedup factor of 11.09 in the total

indexing time for the two evaluation datasets.

Real-world applications have to index a vast amount of multimedia content; consequently, to account for the fact that real-world multimedia systems have to scale, it is worth to examine the implementation of AMR to index media storms. For example, social networks like Flickr [12] and Instagram [69], where users upload photos following different distributions according to the popular events, can benefit from AMR by efficiently indexing the millions of new images that are generated per day in a streaming mode and with limited latency. Moreover, with the proliferation of the IoT where billions of sensors, web cameras, and so on produce a vast amount of multimedia content at an unexpected rate, AMR as an IoT service can provide a scalable solution to index the generated media storms [14].

7 Conclusion

This article presented an efficient approximate media storm indexing mechanisms with MapReduce (AMR) that overcomes the high CPU and latency issues of the exact media storm indexing mechanism (EMR). Moreover, the EMR and AMR indexing mechanisms were implemented using two architectures, a baseline architecture that uses a disk-based strategy and an in-memory strategy using the Flink stream processing framework. As far as literature is concerned, this is the first study to propose a media storm indexing mechanism using the in-memory strategy. This study showed that the Flink-AMR proposed mechanism achieves a 3.12 speedup factor in the total indexing time against the Flink-EMR mechanism, for different indexing workloads. The experimental evaluation also demonstrated that the Flink-AMR mechanism maintains high search accuracy against state-of-the-art approaches in low time, after the media storms have been indexed.

Interesting topics for future work are: i) to monitor the resources allocated by the CDVC components and automatically provide elasticity [70]; ii) to follow a data intensive workload optimization strategy for multi-query execution [71].

References

[1] D. Moise, D. Shestakov, G. T. Gudmundsson, and L. Amsaleg, "Terabyte-scale image similarity search: Experience and best practice," in *Proceedings IEEE International Conference on Big Data*, 2013, pp. 674–682.

[2] H. Wang, B. Xiao, L. Wang, and J. Wu, "Accelerating large-scale image retrieval on heterogeneous architectures with spark," in *Proceedings of the 23rd ACM International Conference on Multimedia*, 2015, pp. 1023–1026.

[3] M. Muja and D. G. Lowe, "Scalable nearest neighbour algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[4] M. Raginsky and S. Lazebnik, "Locality-sensitive binary codes from shift-invariant kernels," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds., 2009, pp. 1509–1517.

[5] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, June 2008, pp. 1–8.

[6] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds., 2009, pp. 1753–1760.

[7] H. Jegou, M. Douze, and C. Schmid, *Computer Vision – ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part I*, 2008, ch. Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search, pp. 304–317.

[8] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua, "Ldahash: Improved matching with smaller descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 1, pp. 66–78, Jan 2012.

[9] G. Shakhnarovich, P. Viola, and T. Darrell, "Fast pose estimation with parameter-sensitive hashing," in *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ser. ICCV '03, 2003, pp. 750–.

[10] W. Zhu, P. Cui, Z. Wang, and G. Hua, "Multimedia big data computing," *MultiMedia, IEEE*, vol. 22, no. 3, pp. 96–c3, 2015.

[11] D. Rafailidis and S. Antaris, "Indexing media storms on flink," in *Big Data (Big Data), 2015 IEEE International Conference on*, Oct 2015, pp. 2836–2838.

[12] "Flickr," www.flickr.com, accessed Jun. 06, 2016.

[13] "Flickr images per day report," www.flickr.com/photos/franckmichel/6855169886/, accessed Jun. 06, 2016.

[14] R. Want, B. N. Schilit, and S. Jenson, "Enabling the internet of things," *IEEE Computer*, no. 1, pp. 28–35, 2015.

[15] D. Novak, M. Batko, and P. Zezula, "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, 2008.

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

[17] D. Novak, M. Batko, and P. Zezula, "Large-scale similarity data management with distributed metric index," *Information Processing and Management*, vol. 48, pp. 855–872, 2012.

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

[19] M. Zhu, D. Shen, Y. Kou, T. Nie, and G. Yu, "An adaptive distributed index for similarity queries in metric spaces," *Web-Age Information Retrieval, Lecture Notes in Computer Science*, vol. 7418, pp. 222–227, 2012.

[20] S. Antaris and D. Rafailidis, "Similarity search over the cloud based on image descriptors' dimensions value cardinalities," *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 11, no. 4, p. 51, 2015.

[21] "Apache flink," http://flink.apache.org/, accessed Jun. 06, 2016.

[22] "Apache spark," http://spark.apache.org/, accessed Jun. 06, 2016.

[23] "Apache storm," http://storm.apache.org/, accessed Jun. 06, 2016.

[24] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, 2015.

[25] E. Tiakas, D. Rafailidis, A. Dimou, and P. Daras, "Msidx: Multi-sort indexing for efficient content-based image search and retrieval," *IEEE Transaction on Multimedia*, vol. 15, no. 6, pp. 1415–1430, 2013.

[26] M. Batko, D. Novak, F. Falchi, and P. Zezula, "Scalability comparison of peer-to-peer similarity search structures," *Future Generation Computer Systems*, vol. 24, no. 8, pp. 834–848, 2008.

[27] M. Aly, M. Munich, and P. Perona, "Distributed kd-trees for retrieval from very large image collections," in *Proceedings of BMVC'11*, 2011.

[28] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast exact search in hamming space with multi-index hashing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 6, pp. 1107–1119, 2014.

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

[30] J. Song, Y. Yang, X. Li, Z. Huang, and Y. Yang, "Robust hashing with local models for approximate similarity search," *IEEE Transactions on Cybernetics*, vol. 44, no. 7, pp. 1225–1236, July 2014.

[31] K. Jiang, Q. Que, and B. Kulis, "Revisiting kernelized locality-sensitive hashing for improved large-scale image retrieval," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[32] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International*

Conference on Very Large Data Bases, ser. VLDB '99, 1999, pp. 518–529.

[33] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, 2004, pp. 253–262.

[34] Y. Gong and S. Lazebnik, "Iterative quantization: A procrustean approach to learning binary codes," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, June 2011, pp. 817–824.

[35] W. Liu, J. Wang, and S. fu Chang, "Hashing with graphs," in *In ICML*, 2011.

[36] W. Liu, J. Wang, R. Ji, Y. G. Jiang, and S. F. Chang, "Supervised hashing with kernels," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, June 2012, pp. 2074–2081.

[37] F. Shen, C. Shen, W. Liu, and H. T. Shen, "Supervised discrete hashing," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 37–45.

[38] A. Gionis, P. Indyk, and M. R., "Similarity search in high dimensions via hashing," in *Proceedings of International Conference on Very Large Data Bases*, 1999, pp. 518–529.

[39] A. Babenko and V. Lempitsky, "The inverted multi-index," in *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3069–3076.

[40] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00514462

[41] J. Kwak, E. Hwang, T. K. Yoo, B. Nam, and Y. R. Choi, "In-memory caching orchestration for hadoop," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 94–97.

[42] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed., 2015.

[43] "Machine learning library (mlib)," http://spark.apache.org/docs/latest/mllib-guide.html, accessed Jun. 06, 2016.

[44] "Sparking water," http://www.h2o.ai/product/sparkling-water/, accessed Jun. 06, 2016.

[45] "Flinkml - machine learning for flink," https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/batch/libs/ml/index.html, accessed Jun. 06, 2016.

[46] "Apache samoa," https://samoa.incubator.apache.org/, accessed Jun. 06, 2016.

[47] "Gelly: Fling graph api," https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html, accessed Jun. 06, 2016.

[48] "Openstack," http://www.openstack.org, accessed Jun. 06, 2016.

[49] "Apache kafka," accessed Jun. 06, 2016.

[50] "Apache hbase," http://hbase.apache.org, accessed Jun. 06, 2016.

[51] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[52] "Cassandra," http://cassandra.apache.org, accessed Jun. 06, 2016.

[53] "Redis," http://redis.io/, accessed Jun. 06, 2016.

[54] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1219–1230, 2014.

[55] H. Wang, J. Li, H. Zhang, and Y. Zhou, *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 4th and 5th Workshops, BPOE 2014*, 2014, ch. Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra, pp. 71–82.

[56] "Rabbitmq," https://www.rabbitmq.com, accessed Jun. 06, 2016.

[57] "Apache activemq," http://activemq.apache.org/, accessed Jun. 06, 2016.

[58] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, Dec 2014, pp. 69–78.

[59] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1654–1655, Aug. 2015.

[60] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, Dec 2015, pp. 797–802.

[61] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015.

[62] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[63] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A survey of open source tools for machine learning with big data in the hadoop ecosystem," *Journal of Big Data*, vol. 2, no. 1, pp. 1–36, 2015.

[64] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: A programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010, pp. 119–130.

[65] "Tiny images dataset," http://horatio.cs.nyu.edu/mit/tiny/data/index.html, accessed Jun. 06, 2016.

[66] "Texmex," http://corpus-texmex.irisa.fr/, accessed Jun. 06, 2016.

[67] "Openstack flavors," http://docs.openstack.org/openstack-ops/content/flavors.html, accessed Jun. 06, 2016.

[68] "Flink monitor rest api," https://ci.apache.org/projects/flink/flink-docs-release-1.0/internals/monitoring_rest_api.html, accessed Jun. 06, 2016.

[69] "Instagram," https://www.instagram.com/, accessed Jun. 06, 2016.

[70] R. Mian and P. Martin, "Executing data-intensive workloads in a cloud," in *Proceedings IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 758–763.

[71] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann, "Shared workload optimization," *PVLDB*, vol. 7, no. 6, pp. 429–440, 2014.

Stefanos Antaris received the B.Sc. and M.Sc degrees from the Department of Informatics of the Aristotle University of Thessaloniki, Greece in 2011 and 2013, respectively. He is a student member of IEEE and his current research interests include Big Data stream processing, Distributed Systems, data analysis and social media mining.

Dimitrios Rafailidis received the B.Sc., M.Sc., and Ph.D. degrees from the Department of Informatics of the Aristotle University of Thessaloniki, Greece, in 2005, 2007, and 2011, respectively. He is a Post-Doctoral Research Fellow with the Department of Informatics, Aristotle University of Thessaloniki. His current research interests include multimedia information retrieval, social media mining, and Big Data platforms.