

Prototyping a Web-Scale Multimedia Retrieval Service Using Spark

GYLFI ÞÓR GUÐMUNDSSON, Reykjavik University, Iceland

BJÖRN ÞÓR JÓNSSON, IT University of Copenhagen, Denmark and Reykjavik University, Iceland

LAURENT AMSALEG, CNRS-IRISA, France

MICHAEL J. FRANKLIN, University of Chicago, USA

The world has experienced phenomenal growth in data production and storage in recent years, much of which has taken the form of media files. At the same time, computing power has become abundant with multi-core machines, grids and clouds. Yet it remains a challenge to harness the available power and move towards gracefully searching and retrieving from web-scale media collections. Several researchers have experimented with using automatically distributed computing frameworks, notably Hadoop and Spark, for processing multimedia material, but mostly using small collections on small computing clusters. In this paper, we describe a prototype of a (near) web-scale throughput-oriented MM retrieval service using the Spark framework running on the AWS cloud service. We present retrieval results using up to 43 billion SIFT feature vectors from the public YFCC 100M collection, making this the largest high-dimensional feature vector collection reported in the literature. We also present a publicly available demonstration retrieval system, running on our own servers, where the implementation of the Spark pipelines can be observed in practice using standard image benchmarks, and downloaded for research purposes. Finally, we describe a method to evaluate retrieval quality of the ever-growing high-dimensional index of the prototype, without actually indexing a web-scale media collection.

1. INTRODUCTION

The advent of smart-phones has re-written the rulebook for media generation, while Internet-based “social” services for storing and sharing this media have completely changed media distribution. Today avid collectors can easily gather hundreds of thousands of image and video files, resulting in terabytes of media. The YFCC100M collection [Thomee et al. 2016] has almost 100 million images and close to a million videos. The Europeana collection hosts 50+ million digitised artworks and other artefacts, while DeviantArt hosts a few hundred million born-digital artworks. And, last but not least, Facebook hosts hundreds of millions of media items. This increased growth of media collection will not stop anytime soon.

The growth of media collections is accompanied by the corresponding growth of multimedia feature collections. Emerging applications shrinking the semantic gap are multimodal, which in turn increases the variety and the size of the features sets that must be stored and processed. With the popularity of deep learning, gigantic multimedia collections are needed as input for high-quality learning, and the semantic features produced by deep-learning based classifiers grow in complexity. In short, there is ever growing need for storage and computing power for multimedia applications.

Author’s addresses: G. Þ. Guðmundsson, School of Computer Science, Reykjavik University, Iceland, e-mail: gylfig@ru.is; B. Þ. Jónsson, Computer Science Department, IT University of Copenhagen, Denmark, e-mail: bjorn@itu.dk; L. Amsaleg, CNRS-IRISA, Rennes, France, e-mail: laurent.amsaleg@irisa.fr; M. J. Franklin, University of Chicago, Illinois, USA, e-mail: mjfranklin@uchicago.edu. Part of the work of G. Þ. Guðmundsson and M. J. Franklin was performed while they were at the AMPLab, University of California, Berkeley. The work of Gylfi Þór Guðmundsson was supported in part by the Inria@SiliconValley program. The research was also supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

1.1. Cloud Computing

Cloud computing addresses such needs, both in term of storage and computing power. Processing multimedia in the cloud is challenging, however, especially for media indexing and retrieval. Indexes must be partitioned across the distributed infrastructure, which is likely to be very heterogeneous, and care must be taken to prevent any storage and/or load balancing problems. In addition, such large distributed settings are failure prone, which calls for employing data redundancy. But, since multimedia collections are extremely large, such redundancy puts high pressure on storage. Furthermore, to ensure proper operation of the cluster, the many nodes involved in the distributed computation must be monitored, which is difficult and costly.

To tackle these difficulties and to fully leverage the computing power of multi-core machines, grids and cloud computing, the data management community has seen a strong trend towards “big data” technologies, such as the automatically distributed computing frameworks (ACDFs) Hadoop and Spark. Such frameworks transparently handle data redundancy, load balancing, failures, communication between nodes, and are very versatile: they have been used to implement a large variety of big data analytics tasks.

Recently, several researchers have experimented with using ADCFs, notably Hadoop, to implement a variety of multimedia-related tasks, including search and retrieval tasks against multimedia material, but mostly using small collections on small computing clusters. But what media applications are Hadoop and Spark suitable for?

1.2. Throughput-Oriented Media Services

Multimedia applications can be broadly divided into two categories depending on their primary performance metric. One category of applications focuses on the interactive *response time* of the system; such systems are typically user-facing and based on media retrieval of some sort. A typical method for implementing such system is to deal with the distributed processing by simply extending the single node approach to multiple servers (e.g., the M-tree [Batko et al. 2010]). Scaling such user-facing applications is difficult, as many machines must typically be involved in answering each query, and coordinating the machines (especially in the face of failures) is challenging.

The second category of applications focuses on the *throughput* of the processing pipeline—how many items can be processed per second—rather than the response time. Examples of such throughput-oriented service arise on web-scale content sharing sites, where many background processes could be run, such as: a service that checks copyright for detecting violations or monetizing content; a service to process and re-format content; and a service to automatically classify newly uploaded content, e.g., with face recognition or classifiers for automated tagging. And there are yet more application domains, where large-scale batch-processing of multimedia is important.

As ADCFs do not provide the response time required to implement interactive services for large multimedia collections, they are only suitable for background tasks, such as batch processing and indexing. Given the growth of media and feature collections described above, however, we believe that it is of significant interest to the multimedia community to study the engineering process and requirements for a throughput-oriented web-scale multimedia service.

1.3. Contributions

In [Moise et al. 2013a] we showed that Hadoop is fundamentally not suitable for implementing large scale search and retrieval tasks. In a nutshell, the execution model for Hadoop is too rigid and it has memory management policies that are incompatible with the needs of large scale indexing and retrieval tasks. This is extensively described below, in Section 2. The objective of this study is to evaluate how effectively the Spark framework can be used

to implement a throughput-oriented service. This paper therefore presents a prototype implementation of a large-scale copyright violation detection service using Spark. We make the following contributions:

- (1) We present detailed pipelines for prototypical index construction and query processing algorithms in Section 4. We show that the Spark pipelines required for indexing and batch processing are not overly complex, and can be easily extended for state of the art multimedia techniques.
- (2) We evaluate the performance of the Spark pipelines running on the Amazon Web Service in Section 5. We use a feature vector collection of 43 billion SIFT feature vectors, the largest experimental collection of high-dimensional feature vectors reported in the literature. Our results show that both index creation and batched retrieval scale well and excellent throughput can be achieved.
- (3) The scalability of the service raises the question of how the retrieval quality evolves as the collection and the corresponding index grow. In Section 6, we show that it is possible to evaluate the impact of the index on the quality of feature retrieval without indexing the feature collection. Such a methodology allows estimating result quality for arbitrarily large collections in a fraction of the time it would take to actually index the collections.
- (4) We have implemented a publicly available copy detection service prototype, described in Section 7, where the functionality of the Spark pipelines can be observed in practice using standard image retrieval benchmarks. All the Spark code will be publicly available via this prototype system.

This article is an extended version of a paper presented in the “Best Paper Session” at the 2017 Multimedia Systems (MMSys) conference [Guðmundsson et al. 2017]. The additions are as follows:

- In Contribution (1), we present additional details of (i) two Spark pipelines needed to implement a full-fledged service, and (ii) a BoW-based indexing pipeline available in the online prototype.
- Contributions (3) and (4) are new contributions in this article.

2. BACKGROUND

A large proportion of the growth in data creation and storage requirements consists of media files, such as images and videos. For example, Facebook claims to store more than 250 billion images, while Youtube users collectively upload more than 300 hours of video content every minute. As a result, there has been significant interest in the scalability of content-based multimedia retrieval [Chang 2011; Jégou et al. 2012; Sivic and Zisserman 2003; Jégou et al. 2011; Sun et al. 2013].

In 2011, Jégou et al. proposed an indexing scheme based on the notion of product quantization and evaluated its performance by indexing 2 billion feature vectors [Jégou et al. 2011]. Also in 2011, Lejsek et al. described a version of the NV-tree that indexes 2.5 billion local feature vectors [Lejsek et al. 2011]. In 2015, Babenko and Lempitsky used the inverted multi-index to index the BIGANN dataset, containing 1 billion features [Babenko and Lempitsky 2015]. Finally, in 2014, Amsaleg reports results using the NV-tree to index a collection of 28.5 billion local features [Amsaleg 2014]. All these approaches are centralized, however, and focus on the response time of the retrieval.

In 2007, Liu et al. reported the earliest distributed work indexing more than a billion high-dimensional feature vectors, but two thousand workstations were used to index the 1.5 billion vectors [Liu et al. 2007]. More recently, Sun et al. rely on the aggregation scheme proposed by [Jégou et al. 2012], indexing 1.5 billion feature vectors from as many images, but using 10 servers [Sun et al. 2013]. The first example of implementing multimedia tasks

on Hadoop is the work of Zhang et al. [Zhang et al. 2010]. Since then multiple similar systems have been proposed, but mostly working with relatively small collections (e.g., see [Gu and Gao 2012; Premchaiswadi et al. 2013; Grace et al. 2014; Yao et al. 2014]). ImageTerrier [Hare et al. 2012] used the largest collection of these, indexing 10.9 million images using BoW features based on about 10 billion SIFT feature vectors [Lowe 2004]. Such systems have also seen some use in the medical image retrieval domain, again with relatively small collections [Grace et al. 2014; Yao et al. 2014; Jai-Andaloussi et al. 2013]. The largest experiments on Hadoop indexed and searched around 100M images, or about 30 billion SIFT feature vectors, using a cluster of 100+ machines [Moise et al. 2013a].

Hadoop and Spark have also found use in other domains related to multimedia. Brandyn et al. used Hadoop to implement various computer vision tasks [White et al. 2010], experimenting with the k -means algorithm clustering about 200GB of data. More recently, Wang et al. proposed a library for Spark to improve performance of image retrieval [Wang et al. 2015]. While only k -means is described in detail, the library contains multiple algorithms for descriptor creation, image retrieval and result processing. Their experiments focus on small collections of less than 500 million descriptors. The KeyStoneML project (<http://keystone-ml.org/>) includes various machine learning algorithms implemented on top of Spark; one is a pipeline for object recognition using Fisher Vectors and SVM. In other recent projects, ADCFs were used for the training phase of deep learning processes, where massive collections feed the network to determine its parameters [Ooi et al. 2015; Moritz et al. 2015]. None of these libraries solve the fundamental problem we address, but as discussed in Section 4.4 some of the algorithms could be used in combination with the pipelines described here to add functionality to the prototype. Note also that, unlike these works, our goal is not to develop a library of algorithms, but rather to study the engineering process of constructing a large-scale multimedia service prototype.

2.1. Requirements for ADCFs

By studying the state-of-the-art in the literature and observing the needs of various multimedia services, we have gathered the following five common requirements that we believe an ADCF should meet in order to form a good basis for implementing web-scale multimedia services:

- R1: Scalability.* Ability to scale out with additional computing resources as more and more data is handled.
- R2: Computational flexibility.* Ability to carefully balance system resources as needed.
- R3: Capacity.* Ability to gracefully handle data that vastly exceeds main memory storage capacity.
- R4: Flexible pipelines.* Ability to easily implement variations of the indexing and/or retrieval process.
- R5: Simplicity.* Efficient use of implementer time through simplicity of code.

In the remainder of this paper, we discuss how well the Hadoop and Spark ADCFs satisfy these requirements.

Note that in [Guðmundsson et al. 2017], the ability to update data structures was identified as the sixth important requirement. We have chosen to omit updates from the discussion here for two reasons. First, there is a trend in the big data literature to architect analysis systems in such a way that updates to data structures are not needed; instead the data structures are always re-computed from scratch and the dynamic additions to the collections are handled separately [Marz and Warren 2015]. According to proponents of this methodology, support for updates to large-scale data structures is not important. Second, we have observed that Hadoop has no support for updates [Moise et al. 2013a] and Spark only has limited support [Guðmundsson et al. 2017]. Since we have limited space to present our work

here, we therefore choose to omit updates. Interested readers are referred to Guðmundsson et al. [2017].

2.2. Map-Reduce and Hadoop

The Map-Reduce framework was first applied to large-scale distributed computing by Google [Dean and Ghemawat 2008]. It exploits data independence through the *Map* and *Reduce* user-level functions. Input data is split into blocks stored on the participating nodes using a distributed file system such as HDFS [Shvachko et al. 2010]. The framework favors processing data locally, transparently handles the scheduling of tasks, and deals with communications when nodes send/receive records to process.

Previous work, however, has shown that while Hadoop is good for solving a particular type of big data problems, it fails at adequately solving the issues that multimedia systems raise [Moise et al. 2013a]. For example, there are no facilities for updating the feature vector collection, the pipeline is very simple and rigid, and the code required to implement even elementary retrieval processes is complicated. Thus, Hadoop does not satisfy all of the five requirements listed above: Hadoop does scale out to an extent (requirement *R1*), but it fails with all the other requirements.

Let us consider these drawbacks in more detail. Hadoop forces systems to use only a single input source: the HDFS data blocks. Many multimedia services use two sources of data, however, such as systems which need a codebook at indexing time, in addition to the data that must be indexed. Hadoop allows mappers to load distributed variables at launch time, so one such variable can store the codebook. But since mappers on the same physical node cannot share the main memory allocated for that variable, the codebook must be loaded again and again by each mapper, even when running on the same node. Hadoop thus only partially supports requirements *R1* through *R3*.

The inflexible two-step Map-Reduce architecture is also causing troubles. It is extremely difficult to run iterative or recursive processes such as a *k*-means which is often used to create codebooks. Workarounds require embedding Hadoop tasks inside high-level wrapping code repeatedly invoking Hadoop [Owen et al. 2011]. Having multiple sources or levels of codes leads to complications and poor performance. Hadoop thus does not satisfy requirements *R4* and *R5*.

2.3. Spark







Central to Spark is the notion of a *Resilient Distributed Dataset* (RDD) [Zaharia et al. 2012; Zaharia et al. 2010], a distributed data structure on disk or in memory. Spark facilitates transforming and manipulating RDDs in order to meet application needs and allows chaining operations in arbitrarily deep and complex pipelines. Spark defines many operators to manipulate the RDDs; the most common operators are listed in Table I, which also introduces a graphical notation for the operators used in the remainder of this paper.

Spark typically uses HDFS as its file system. Data in an RDD is thus typically partitioned and spread out over the computing cluster machines and Spark manipulates the data where it resides. Spark allows the programmer to choose to persist RDDs to various storage-levels (e.g., RAM or secondary storage). Keeping RDDs in RAM preserves the performance of algorithms with iterative/recursive access patterns repeatedly scanning data.

Spark uses a Master-Worker workflow where the main code base is executed on the master and the distributed executions operating on an RDD flow out to the workers. Spark uses a lazy execution model where operations on RDDs are chained together until it becomes necessary to instantiate the data. Lazy execution facilitates optimizations.

A major goal of our study is to determine how well Spark satisfies the five requirements above; this is discussed in Sections 4 and 5.

Table I. Common RDD operators in Spark.

 <i>.map</i>	The <i>.map()</i> operator is a 1-to-1 transformation of each element into a new RDD of equal size. Example use cases: type conversions (int to long or text to number) or re-keying a key-value pair RDD.
 <i>.flatMap</i>	The <i>.flatMap()</i> operator is a 1-to-many transformation as each element invocation can return a list of new elements that are flattened into a new RDD. Example use case: splitting lines of text into words.
 <i>.groupByKey</i>	The <i>.groupByKey()</i> operator invokes a shuffle of the RDD to group the elements. The <i>.groupByKeyKey()</i> operator does the same for key-value pair RDDs, returning an RDD where each value is an iterator of the key's elements.
 <i>.reduceByKey</i>	The <i>.reduce()</i> operator reduces all elements of an RDD into a single instance. The <i>.reduceByKey()</i> operator is the key-value RDD equivalent, reducing all elements with the same key into a single value. Example use case: summing numeric values.
 <i>.collectAsMap</i>	The content of an RDD can be collected as an Array from the workers to the master with the <i>.collect()</i> operator or as a hierarchical map structure with <i>.collectAsMap()</i> .
 <i>.persist</i>	The <i>.persist(storage-level)</i> operator is used to tell Spark where to keep an RDD after instantiation. The storage level defaults to in-memory.

3. DESIGN CHOICES

In this study, we are primarily interested in understanding the pros and cons of using Spark and cloud-based processing to implement (near) web-scale multimedia services. To that end, we decided to implement a multimedia task for the study with i) a feature vector collection of tens of billions of feature vectors, and ii) a workload representing both background processing and an on-line multimedia service.

3.1. Choice of Multimedia Tasks

Many throughput-oriented media tasks can be distributed relatively easily. Video encoding or compression, for example, can be scaled by assigning each video (or part of a video) to its own worker; as no coordination is needed, using ADCFs to scale these applications is straight-forward. We have therefore chosen to focus on more demanding indexing and retrieval tasks.

The only application in the literature handling tens of billions of feature vectors is the DeCP algorithm, which is a prototypical multimedia retrieval system applied to copy detection using a collection of 30 billion local feature vectors using Hadoop [Moise et al. 2013a]. This algorithm has a number of useful features for our study:

- The implementation and performance of DeCP has been studied extensively, including the impact of solid state disks, multi-core machines, and distributed processing (with Hadoop).
- It is a simple clustering and retrieval algorithm that is easily explained, understood, and implemented; yet is efficient and distributes well.
- The DeCP algorithm has both a pre-processing task (creating the clustered index) and a subsequent online task (batched image retrieval), and is thus representative of many different types of multimedia services.
- While the algorithm is simple, it can easily be extended to work with state of the art methods, such as BoW. In Section 4.4, we show how to implement such pipelines.

- The DeCP algorithm has been shown to give results of high quality for the copy detection case, even at a scale of 30 billion feature vectors.

As discussed in Section 2, the Hadoop implementation of DeCP highlighted some of the shortcomings of Hadoop for multimedia tasks. Since the aim of Spark was to address some of those shortcomings, it is also interesting to implement DeCP on Spark to study whether the intended benefits of Spark materialize for this multimedia retrieval algorithm. We therefore focus on implementing DeCP on Spark.

3.2. Choice of Feature Vector Collection

As our target is to study processing of a feature collection containing tens of billions of feature vectors, the only realistic choice is to use SIFT. The SIFT feature vectors use a Difference of Gaussian (DoG) detector to find a large number of interest points in an image, and then encode the image gradients and their orientations around each of the points into a 128-dimensional histogram. While deep learning feature vectors have surpassed SIFT for some multimedia tasks, SIFT remain very competitive for partial copy detection. Most importantly, however, since a typical image yields hundreds of SIFT vectors, millions of images can yield billions of vectors. As we only have access to millions of images, this is the most important criterion for the feature vector collection.

Since the DeCP algorithm was developed in the context of the Quaero project, the feature vector collection from [Moise et al. 2013a] is not publicly available. The largest experimental image collection available now is the recently developed YFCC100M collection; fortuitously the YLI collection of SIFT feature vectors computed from the YFCC100M collection was made available just in time for our study. The YLI collection contains about 43 billion feature vectors, which require almost 7TB of storage; this is sufficient to truly exercise the capabilities of the Spark system.

As it turned out, the SIFT feature vectors for YFCC100M were extracted using an implementation of SIFT that differs from the ones used when extracting local descriptors from existing ground-truth such as Holidays [Jégou et al. 2008a], Copydays [Jégou et al. 2008b], Oxford5k [Philbin et al. 2007], Paris6k [Philbin et al. 2008] or other well established benchmarks. As a result, we can unfortunately not report quality metrics in this study. However, the DeCP algorithm has already been shown to yield results of good quality, even at a comparable scale. Furthermore, various versions of SIFT exist and have been compared. They all prove to be quite equivalent and very stable in terms of recognition capabilities across implementations. We are therefore confident that the quality results obtained with DeCP and reported in [Moise et al. 2013a] would not radically differ when using the YFCC collection as distractors, instead of the unavailable Quaero set of distractors.

Note, however, that we discuss a methodology for measuring the impact of the index on retrieval quality as the collection size grows in Section 6. For the study in Section 6, we can use a consistent set of SIFT features, based on standard benchmarks, thus avoiding the problems discussed above with the SIFT features from YFCC100M.

3.3. Choice of Experimental Environment

We are interested in using cloud processing for multimedia tasks. As the workplace of two of the authors (at the time) had an agreement with Amazon that provided Amazon Web Services (AWS) credits, it was an obvious choice to use AWS for our experiments (and, in fact, the only available choice). This choice has both benefits and drawbacks. The benefits include the fact that the YFCC100M collection and the associated YLI set of SIFT feature, were made available on AWS, reducing storage requirements.

The main drawbacks involved difficulties obtaining diverse and reliable performance measurements:

- First, since we were using the YLI collection, which was stored in the relatively small West Coast data center, we had a limited selection of machine types to choose from, which impacted our ability to study multiple machine configurations. In particular, since our data required more than 20 TB of disk space (at replication factor 3), the limited availability of local disks restricted our choices significantly.
- Second, since we had to use a low-price resource allocation policy, experiments were frequently cut short due to pricing peaks. Furthermore, as our experiments required a substantial portion of the computing capacity of the data center, running experiments sometimes generated pricing peaks, which then cut the experiments short. Based on this experience, we can recommend using a large data center for such large-scale experiments.
- Third, as the grant period for the project came to an end, the computing resources became unavailable, and further experiments could not be performed.

3.4. Choice of (No) Baseline Comparisons

Aside from the previous work on DeCP using Hadoop, there are no existing studies at this scale to compare to. As our collection is larger and the computing hardware is radically different, we can only discuss (in Section 5.4) the differences between our results and those of [Moise et al. 2013a].

3.5. Research Questions

We are primarily interested in understanding the pros and cons of using Spark and cloud-based processing to implement (near) web-scale multimedia services. To that end, we are specifically interested in the following research questions:

- (1) What is the complexity of the Spark pipelines for typical multimedia-related tasks?
- (2) How well does background processing scale as collection size and resources grow?
- (3) How does batch size impact throughput of an online service?

In the next two sections we answer these questions for the particular case of DeCP running on Spark. It is our belief that since the application is quite representative for many multimedia-related tasks, as discussed above, our conclusion will generalize to many other multimedia tasks.

4. PROTOTYPE IMPLEMENTATION

We start with a brief description of the DeCP algorithm before presenting its implementation on top of Spark. We then discuss using Spark to go beyond the prototypical algorithm and implement alternative state-of-the-art indexing approaches. Finally, we summarize the engineering effort of our project, and discuss how well Spark satisfies the requirements *R1* through *R5*.

4.1. The DeCP Algorithm

DeCP is quite representative of the core principles that are underpinning many unstructured quantization-based high-dimensional indexing algorithms and its extremely simple search procedure covers a large spectrum of existing approaches. For example, we later show how DeCP can be modified to mimic not only the seminal VideoGoogle approach [Sivic and Zisserman 2003], but also its descendants, with minimal changes to the Spark code.

DeCP uses a codebook learned over some data to guide the index construction. During the indexing phase, each image feature vector is assigned to the closest codeword(s), thereby forming clusters. The retrieval phase implements an approximate k -NN search process where only a single matching cluster (or a few) is scanned.

To improve scalability by lowering the cost of identifying the matching cluster(s), DeCP uses a multi-level hierarchy of codewords, as in many other approaches (e.g., [Nistér and Stewenius 2006]). When indexing large feature vector collections with DeCP (say, a few

billion feature vectors) best practice calls for creating a few million clusters, each grouping about 15K features vectors (to utilize IO operations well), along with a codebook hierarchy of 3 to 5 levels (to minimize CPU cost). The wider and deeper the index becomes the harder it gets to find the correct cluster at search time. What we have is essentially a three-way balancing act between 1) CPU cost of scanning, 2) CPU cost of traversing the index and 3) search quality. As many other state of the art approaches, DeCP may use soft assignment and multi-probing to increase the quality of results (see [Philbin et al. 2008]). Finally, DeCP can easily process large batches of queries, searching up to a few million query vectors at the same time, maximizing throughput at the expense of response time.

DeCP has been shown to return approximate results of good quality at very large scale. It has been used to execute large batches of queries against extremely large image feature collections (up to 30 billion SIFT feature vectors). The codebooks used were up to 6 million clusters wide and 5 levels deep but to deal with a truly web-scale dataset of over 1 billion images (400+ billion features) the width of the index would have to be in the tens of millions. Such indices are studied in Section 6.

DeCP’s design principles, its architecture, some implementation details and extensive performance results have been published in various venues [Moise et al. 2013a; 2013b; Shestakov et al. 2013].

4.2. DeCP Index Construction

The feature vector collection must be stored on HDFS before the index construction can start. The codebook for DeCP is created by randomly sampling the feature vectors using the *.take()* command. The sampled vectors are organized top-down into a multi-level tree, which is then pushed to persistent storage by writing it as a serialized object file.

The pipeline for quantizing, i.e., assigning feature vectors to codewords, is a rather straightforward chain of RDD operators, shown in Figure 1. The codebook is first broadcast to all workers, as shown using the dashed arrow in the figure. The RDD with the feature vector collection is then chained to a *.map()* operator which i) reads the input data and ii) traverses the codebook to determine the codeword that is the closest to each input feature vector. This first *.map()* operator produces a new RDD made of key-value pairs, where each pair is composed of the *codewordID* as the key and the current feature vector as the value, thus representing the assignment of each feature vector to a cluster. That RDD is then chained to a *.groupByKey()* operator which groups the pairs according to the key, the *codewordID* in this case, which essentially shuffles the data across the network in order to prepare for the final cluster formation. As a technical detail, the *.groupByKey()* operator produces a new key-value RDD where the value is an iterator, so a final *.map()* operator is needed to convert the iterators into arrays such that the RDD can be serialized and stored to disk. At that point, the feature vectors have been assigned to clusters which are in turn distributed using HDFS.

When this pipeline is executed, the underlying distribution of the feature vector data on HDFS is inherited by the initial RDD instantiation. Subsequent RDDs, resulting from shuffling or other Spark operators, may be distributed differently. Recall that RDDs are

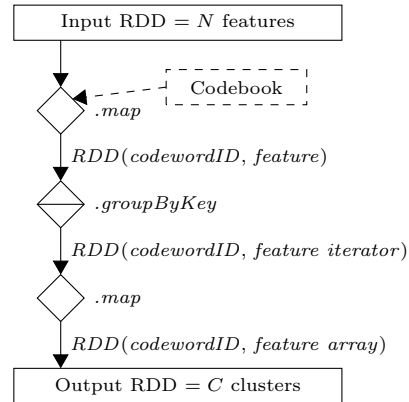
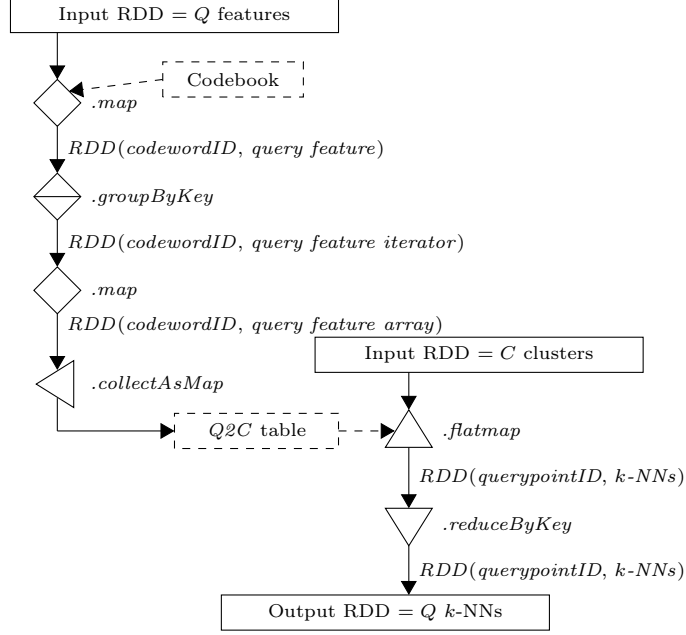


Fig. 1. Spark pipeline for indexing (quantization).

Fig. 2. Spark pipeline for batch k -NN search.

instantiated only when needed and are transient, unless persisting operators are used explicitly, so the whole pipeline is executed at the same time.

4.3. DeCP Batch Retrieval

The DeCP query pipeline starts by creating an RDD from all the query feature vectors in the query batch. The codebook of the index is then loaded to identify the codeword that is most relevant for each query vector. Once this is done, the queries can be grouped according to their *codewordID*, which shows which query vector requires data from which cluster. This is, again, the same pipeline as the one used in the initial steps of the index creation, albeit with one difference. As many other state-of-the-art multimedia retrieval systems, DeCP may use multi-probing at search time [Philbin et al. 2008]. With multi-probing, multiple codewords are involved per query and multiple clusters must therefore be scanned, possibly on different machines, each resulting in intermediate k -NN lists that must be merged and consolidated to eventually form the final k -NN of the query.

Figure 2 shows the batch search pipeline. The start of the pipeline creates an RDD that associates *codewordID* to *queryID*, resulting in an RDD which contains the query-to-codeword table (*Q2C*, top half of Figure 2). Note that the RDD might contain several entries for each *queryID* due to multi-probing. Entries in the *Q2C* table are grouped using *.groupByKey()* according to the values of *codewordID*. This table is collected and broadcast to all nodes (dashed arrow).

The bottom half of Figure 2 shows the remaining part of the pipeline. A *.flatMap()* operator reads the indexed RDD, as well as the broadcast *Q2C* table, and determines the k -NN of each query point for each codeword that is concerned by this query batch. This is the main operation of the pipeline, where neighbors are matched with query features. This creates another RDD of pairs with *queryID* as the key and the k -NN as the value. Here again, multiple pairs with the same *queryID* key may exist due to multi-probing, so the next step in the pipeline is to merge and consolidate the multiple intermediate k -NN

lists that exist for each query. This is done using a `.reduceByKey()` operator that produces a unique k -NN result list per query vector.

Note that Hadoop could not support multi-probing as there was no efficient way to handle the two reduction steps that are required. To implement multi-probing in Hadoop, it is required to push to HDFS all the intermediate k -NN lists, and then initiate a second full MapReduce job to load these lists, perform the merge and consolidate the lists. In addition to the costly launch overhead of Hadoop, the performance would suffer seriously due to writing and then reading data on secondary storage. In contrast, this process is easy and efficient with Spark as transformations of RDDs can be chained; RDDs remain in main memory; and the whole pipeline is a single job.

4.4. Beyond DeCP: Advanced Pipelines

Multimedia retrieval systems in real life typically include components that are not part of the DeCP approach described above. Furthermore, state-of-the-art systems may use techniques that slightly differ from the ones used in DeCP, which has been designed as a *prototypical* retrieval system. In particular, a fully functional multimedia retrieval system must extract feature vectors from the media file and we first show how external Computer Vision libraries can be linked to Spark. We then describe how secondary similarity measures can be added to DeCP in order to re-rank candidate images. This allows, e.g., the implementation of a voting process or weak geometry verifications when dealing with local feature vectors. Finally, we show how DeCP can be extended to imitate the high-dimensional indexing strategies derived from the seminal Bag-Of-Words (BoW) paradigm [Sivic and Zisserman 2003].

4.4.1. Extracting Feature Vectors from Media. So far we assumed that feature vectors were already extracted from the images and ready for indexing and querying. It is possible, however, to extend the pipelines sketched above to include a feature extraction step. First, the multimedia material must now be stored in HDFS. The next step is to run the feature extraction code, in a distributed manner, saving the resulting high-dimensional vectors into a new RDD using `.map()`, and then we do a `.flatMap()` to label each feature and flatten the structure. This step, shown in Figure 3, must be added to the pipelines of Figures 1 and 2, but the pipelines are otherwise identical to what was described before.

Connecting a Java library (e.g., BoofCV) to Spark is trivial, but it is more complicated to connect a C/C++ library. Spark can invoke legacy code with the `.pipe()` operator but that only passes text via std-in and std-out, which is not suitable for large volumes of high-dimensional features. Another alternative is relying on JNI to wrap the legacy library, allowing invocations from `.map()` or `.flatMap()` operators. Traditional computer vision libraries can thus be utilized (such as OpenCV or VLFeat), allowing the extraction of state-of-the-art feature vectors such as MFCC (for audio), SIFT, SURF, or the more sophisticated VLAD features [Arandjelovic and Zisserman 2013]. This is, for example, done in the ML-Lib vision pipeline, where a JNI wrapper allows using VLFeat to extract dense SIFT features [Vedaldi and Fulkerson 2010].

4.4.2. Secondary Similarity Measures. Various motivations have driven researchers to use secondary similarity measures in multimedia retrieval systems, including removing false positives, trying to defeat the curse of dimensionality, and compensating for the asymmetry

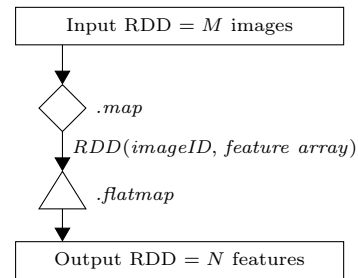


Fig. 3. Integration of feature extraction in Spark. The resulting feature collection must be the input to the indexing and search pipelines of Figures 1 and 2. If needed, it could also be stored on disk using a `.persist` operator.

of NN-based similarity. Typically, the traditional primary k -NN similarity creates a list of candidates, which is then processed using a secondary similarity measure to re-rank the candidates; the top elements of the re-ranked list are then returned to the user. It is possible to equip DeCP on Spark with such secondary similarity measures and in the following we give two examples that are representative of techniques found in the literature.

The first example is a voting-based secondary similarity measure, for example needed when using local features such as SIFT [Lowe 2004]. With local features, each image, including the query, is described using many feature vectors. The similarity is established by first identifying the k -NN of each query local feature vector and then making each identified local feature vector “vote” for the image it belongs to. Once all the query local feature vectors have been used to probe the index, the images with the most votes are returned as the most similar.

The vote aggregation process in Spark requires adding operators to the search pipeline in order to transform the RDD that contains the consolidated k -NN lists. This RDD groups the lists according to their *queryID*. It is necessary to read this RDD and reshuffle its data according to the *imageID* using a *.reduceByKey()* followed by a *.map()* that will count the number of votes each image receives from the collection. The resulting pipeline is shown in Figure 4. Note that all the experiments described in the next section use this vote aggregation secondary similarity measure.

The second example re-ranks the candidate images according to an estimate of the degree of geometric consistency of angles and scales between the query and the candidate images [Jégou et al. 2008a]. Pushing consistent images up in the ranking, and inconsistent images down, is easy as geometry and scale information is integrated into the feature vectors. Unlike most systems, a costly re-extraction of the feature vector information from the candidate images is not needed as we can simply include the original feature vectors in the search pipelines RDD with very minimal code changes. The actual post-process of re-ranking can then be appended to the pipeline using the necessary RDD transformations; for example, *.flatMap()* to compare randomly chosen sets of feature vectors in the result to the corresponding sets of original query features (already in memory in the *Q2C* table), and *.reduceByKey()* to gather the geometric results per image and output the final re-ranked results. The Hamming embedding approach discussed in [Jégou et al. 2008a] can also be implemented in a similar fashion.

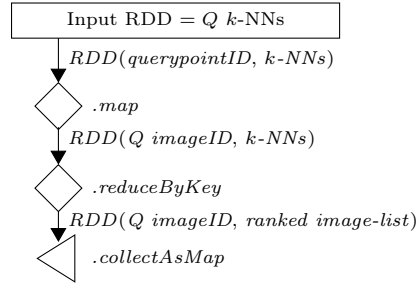


Fig. 4. Integration of vote aggregation in Spark. The input RDD is the output of the search pipeline of Figure 2.

4.4.3. Imitating BoW. The Bag-Of-Words (BoW) approach was originally proposed by Sivic and Zisserman [Sivic and Zisserman 2003]. Many extensions and improvements have subsequently been proposed, making the BoW approach a seminal contribution to the field of high-dimensional indexing. BoW basically applies to images textual information retrieval techniques where the words of the document collections are recorded in a vector model with a cosine-based metric and a *tf-idf*-based secondary similarity measure. With images, the local image features are turned into “visual words” using a visual codebook and feature vectors are clustered with their closest codeword. Each local query feature vector votes for *all* the images assigned with the closest codeword(s), so scanning the clusters to search for the closest feature vectors is in fact not needed.

This impacts the pipeline as some Spark operators can be removed and the code simplified. Early in the pipeline, it is not necessary to create an RDD where the features from the collection to index are kept; instead it is only necessary to keep track of which *featureID* gets assigned to which *codewordID*. The resulting RDD is much smaller (as the components of the feature vectors are not needed) so it can potentially remain in (the distributed) main memory, thus enhancing performance. Stripping the *featureID* from an existing indexed dataset can be done using a *.map()* operator. The subsequent *.flatMap()* of the search pipeline in Figure 1 is also simplified, as the code for scanning the selected cluster(s) is not needed anymore. The resulting pipeline is represented in Figure 5.

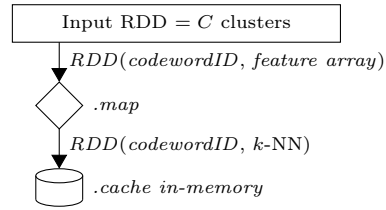


Fig. 5. Index creation for BoW search. In contrast with DeCP indexing, the resulting index is small and can be cached in memory.

4.5. Discussion

In this section, we have described the engineering process for a prototypical (near) web-scale multimedia service implemented on top of Spark. We have described how we were able to take advantage of the flexibility built into Spark, both with respect to the advanced resource management and the flexible pipeline construction.

As an example of the former, Spark is able to use main memory very effectively, meaning that with Spark the scalability of DeCP is only bound by the amount of RAM per machine and not by the amount of RAM per core, as was the case in Hadoop. We believe that the requirements of scalability, computational flexibility and capacity, *R1* through *R3*, can all be satisfied quite well by Spark.

As an example of the latter, we have shown how Spark’s flexibility and deep pipelines provide the tools necessary to implement a full-feature system seamlessly, and we have also described how some common post-processing steps, such as re-ranking, can be added with minimal overhead and code changes. None of this was considered remotely feasible with Hadoop [Moise et al. 2013a]. This discussion thus shows that the requirements for flexibility and simplicity of pipelines, *R4* and *R5*, are satisfied very well. The support for the last two requirements is perhaps best articulated by the following three observations: a) we are able to build a full-featured system with relatively easily explained pipelines; b) we can often propose more than one way to solve the same task; and c) we have proposed numerous extensions to implement more complex pipelines with very modest code changes.

We also learned that working with collections at this scale is a very time-consuming and expensive process. As one example, converting the feature collection from text format to binary format took weeks. This conversion process required using machines with large memory, as the vectors were stored in very large compressed text files and Spark does not allow partial decompression of such files. Renting these machines was very expensive, and hence only a few of them were used. These machines were not connected to large disks, however, so remote writes were mandatory to store the output of the compression. It is that combination of costs, storage and network capacities that caused this phase to be so daunting. As another example, storing the resulting compressed collection would cost hundreds of dollars per month, and therefore the collections were removed after experimentation was completed.

However, given the relative ease of implementing these pipelines, as well as the results reported below, we conclude that Spark has very strong potential for implementing various families of large-scale multimedia services.

5. EXPERIMENTS AND RESULTS

All experiments reported in this section were run on Amazon Web Services (AWS), using C3.8xl nodes. Each node has 60GB of RAM, 640GB of SSD storage, an E5-2680 CPU at 2.8GHz with 32 virtual cores (vCores). Intel hyper-threading technology is used for half of these vCores; this is known to perform worse than a true core. We have in all cases used 51 of these C3.8xl nodes to create the Spark cluster. The cluster is thus composed of 1 master and 50 slaves, for a total of 1600 vCore workers. This configuration allows using 2.8 TB of RAM and 30 TB of HDFS SSD storage in total. Recall, from Section 3.3, that we were using a relatively small data center; as a result this was the smallest configuration that could fit our data set on local disks, and due to difficulties with pricing peaks we could not run larger configurations.

Our feature vector collection is the recent publicly available YLI corpus, which consists of 42,949,150,170 SIFT feature vectors derived from 96,560,779 Creative-Commons-licensed Flickr images [Thomee et al. 2016]. We converted the features to a binary format stored as an RDD in our S3 bucket; in total the collection requires about 7TB of disk space. To facilitate running experiments at various scales, we partitioned the 43 billion feature vector collection into five roughly equals parts. Each of the five parts, referred to as *a*, *b*, *c*, *d* and *e* respectively, contains approximately 8.5 billion SIFT vectors and occupies about 1.4TB of disk space. We believe that we are reporting the first experiments using the full SIFT collection of the YLI corpus, and hence the largest feature vector collection ever used in the literature.

To index the collection (and the various sub-collections) we created a 5-level codebook that defines 20 million codewords. We have intentionally used a large codeword hierarchy which can accommodate much more than the YLI collection. With 43 billion feature vectors, only about 2,100 feature vectors are assigned to each codeword on average (or about 425 for each of the five parts *a* through *e*), while it is good practice to fit between 15 and 50 thousand feature vectors per codeword [Moise et al. 2013b]. The codebook hierarchy used here could thus gracefully, *with absolutely no change*, scale to indexing 500 billion to a few trillion vectors with only a linear increase in the time required for quantization. Note that the overhead for traversing this hierarchy and managing so many codewords negatively impacts the index creation and search times that we report, as a 4-level codebook defining 2 to 4 million codewords would have been more appropriate for indexing 43 billion feature vectors; the overhead is of course even worse when indexing smaller sub-collections.

Note that aside from basic Spark parameter tuning, we have not performed any optimization. We run the Spark framework “out-of-the-box” and none of the authors are experts in either Java or Scala. Furthermore, we deploy our cluster in AWS where accurate monitoring of the environment is limited due to virtualization and concurrency.

5.1. Experiment #1A: Scaling Out Resources for Indexing

The first large scale experiment observes the ability of Spark to scale out; how the execution time decreases in proportion to the increase in hardware resources. We focus on the index construction (the most CPU intensive pipeline) of a relatively small sub-collection (*a*). To measure the ability to scale out, we set a Spark configuration parameter (`spark.cores.max`) to limit the number of vCores to 400, 800 or all 1600 vCores. Recall that the first 800 vCores are true hardware cores while the last 800 vCores are hyper-threading cores.

The performance of the index creation pipeline against these three AWS configurations is summarized in Table II. Its second line is when hyper-threading cores are not used.

Table II. Experiment #1A: Scaling out index creation.

vCores	Indexing Time (s)	Relative Scaling Observed / Optimal
400	5,931	— / —
800	3,510	0.59 / 0.50
1600	3,287	0.55 / 0.25

Doubling the number of vCores in use nearly divides in half the time it takes to run the index creation. The added overhead of using 800 true cores instead of 400 is 18% above the optimal, see the Observed/Optimal column.

Above 800, the added cores are hyper-threading vCores. While the number of vCores is doubled, the quantization time is only reduced by about 7% (third line of Table II). This is caused by the poor performance of hyper-threading cores as indicated by Intel’s guide-lines and observed in work on Hadoop [Moise et al. 2013a; Shestakov et al. 2013].

5.2. Experiment #1B: Scaling Up Collection for Indexing

This second experiment is intended to observe the ability of the system to scale up; how the execution time evolves when indexing larger and larger collection with the same hardware. For this experiment, we use the full 1600 vCores of the 50 C3.8xl AWS worker nodes. We measure the wall clock time for running the index creation pipeline against five feature collections of increasing size, ranging from a to the full collection of 43 billion descriptors.

The wall clock time for the indexing (quantization) is reported in Table III. It takes 3,287 seconds to complete the index creation pipeline when indexing the a sub-collection. Indexing the full 43 billion features takes 19,749 seconds, or about six times longer. As the last column of Table III indicates, the system scales up quite well with larger collections.

The indexing time for the largest collection is about 5.5 hours, which can be decomposed into about 2.5 hours for assigning each feature vector from the collection to its appropriate codeword and about 3 hours to achieve the shuffling process grouping the feature vectors into clusters. Note that despite using as many as 1600 vCores, only 50 machines were used and they had to shuffle about 7TB of data, which severely stresses the communication links.

5.3. Experiment #2: Scaling Batch Retrieval

This last experiment examines the performance of searching the full scale collection with batches of query images. We have built batches of queries by sampling the collection, resulting in up to 80,000 images in a single batch, each having about 400 query features, for a total of up to 32 million query features in a single batch.

We use the 51 C3.8xl nodes. The codebook is the 5-level hierarchy organizing 20 million codewords. Each experiment is an end-to-end batch-search job, where the wall clock running time of the entire job is measured. Note that the wall clock time includes the time required to load the codebook at job launch time, about 550 seconds, which is something a live system would only do once.

In this experiment, multi-probing is not used (as we are not studying retrieval quality, and turning multi-probing off emphasizes the network connections) and the number of neighbors collected for each query point is set to $k = 20$. In this discussion we focus on the time to process the batch and the corresponding throughput.¹

The time it takes to entirely process batches of queries are given by Figure 6. It takes about 1,000 seconds to process the smallest batch, which contains 2,500 images, while it takes close to 1,500 seconds to process the largest batch of 80,000 images. Multiplying the size of the batch by 32 thus increases the response time only by a factor of 1.5: larger batch requires relatively little more disk activity to read clusters, while utilizing CPUs much better.

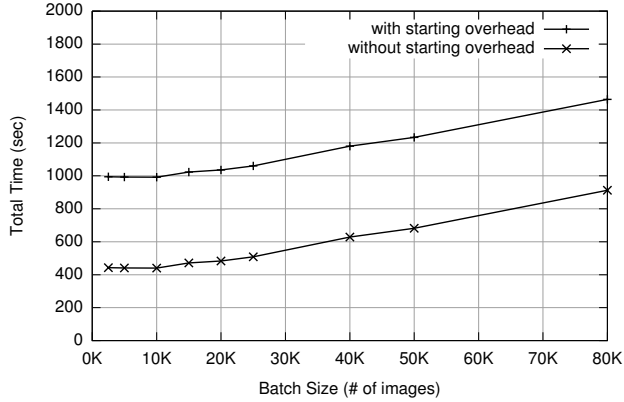
Table III. Experiment #1B: Scaling up index creation.

Collection	Descriptors	Indexing Time (s)	Relative Scaling
a	8.5B	3,287	—
$a - b$	17.2B	5,030	1.53
$a - c$	26.0B	11,943	3.63
$a - d$	34.5B	14,192	4.31
$a - e$	42.9B	19,749	6.00

¹As discussed in Section 3, we do not report quality indicators here, as a) no ground truth is available for the YLI feature collection and b) the DeCP algorithm has been shown to return good results at scale.

Figures 7 and 8 show the time per image and throughput, respectively, both with and without the time required to load the codebook. As the figures show, the time required per image initially drops very fast as the batch size is increased, with a corresponding increase in throughput. Furthermore, the figures show that as the batch size increases, the impact of loading the codebook becomes smaller. The time required to process one query image from the batch drops from 0.39 seconds with the smallest image batch to 0.018 seconds when processing the largest batch. This suggests that with small batches most of the CPUs sit idle, waiting for data to arrive. With larger batches, in contrast, the CPUs are more effectively used and the throughput of the system improves.

Fig. 6. Experiment #3: Total running time.



5.4. Discussion

One of the primary limitation of implementing even a basic multimedia service in Hadoop [Moise et al. 2013a] was that scalability was bound by the RAM per core. As the collection grows, larger data structures are typically needed for managing the collection (the cluster index, in the case of DeCP), which in turn require more RAM memory. This is not an issue with Spark, and in our experiments we even used a significantly larger index than needed to emphasize this difference.

Also, in contrast to the Hadoop implementation reported in [Moise et al. 2013a], we implemented a full-featured system using Spark. This was not originally planned, as it was not until we started working with Spark that we realized how the various features of the framework made it easy for us to accommodate the more complex pipelines. This is a clear testament to the simplicity and flexibility of the Spark framework.

Our experimental results reinforce our conclusion that Spark supports large-scale throughput-oriented multimedia services very well. We have investigated the performance of index construction and batch search, and shown that Spark scales both out and up. Using a grid of a hundred machines, DeCP on Hadoop needed more than half a second per image; in these experiments, however, a large batch results in an average time per image of less than 20 milliseconds! This is despite the fact that we are running experiments on heavily under-sized clusters, due to the over-sized index.

6. EVALUATING RETRIEVAL QUALITY AT SCALE

An important question, when extending systems and applications towards web-scale, is how the result quality evolves as the collection and the corresponding index grow. There are two important use-cases. First, indexing a very large collection once is time-consuming enough (and expensive, when using commercial cloud services), so repeatedly indexing the collection with different index configurations is completely infeasible. Second, we may wish to understand the evolution of result quality for collections much larger than those currently available. In this section, we show that for a copy-detection application it is possible to evaluate the impact of the DeCP index structure on quality, without actually indexing the entire collection, thus reducing the work of quality evaluations significantly. We first outline our approach, and then use it to evaluate the quality of the DeCP index as the size of the

Fig. 7. Experiment #2: Search time per image.

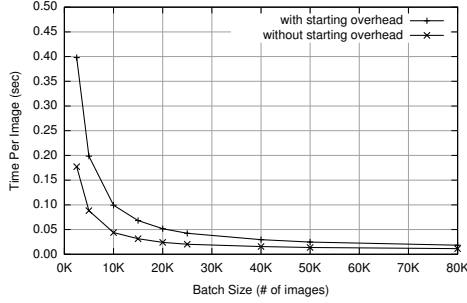
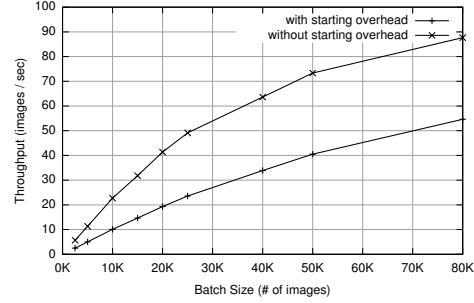


Fig. 8. Experiment #2: Batch search throughput.



indexed (hypothetical) feature collection grows from roughly 10 billion features to 1,000 billion features.

6.1. Quality Evaluation for Single Feature Retrieval Benchmarks

Consider a ground truth for single feature retrieval, generated using a sequential scan over a fixed collection, where the outcome is a set of query features f_i^q , each with a matching nearest neighbour f_i^n that the approximate retrieval process should find. In an approximate index, there are two possible reasons why a query for f_i^q would not find the correct f_i^n within the k -nearest neighbours. First, the retrieval process might take a wrong turn inside the index structure and arrive at a cluster (or leaf) that does not contain f_i^n . When this happens, the analysis of the cluster is guaranteed to be in vain. Second, when the index guides search to the correct cluster, the analysis of its content might still be too approximate and miss the feature f_i^n .

For an algorithm, such as DeCP, that stores the entire feature within the cluster and runs a full nearest neighbour search within the cluster, however, any retrieval for f_i^q that finds the cluster containing f_i^n is then guaranteed to locate the correct answer. The central observation behind our method is that to evaluate how well the index guides the retrieval of f_i^q towards f_i^n , we only need the index hierarchy, not the actual clusters: if the index hierarchy guides the search for both f_i^q and f_i^n to the same cluster, then the index structure has done its job.

Note that this observation also extends to multi-probing, where a query retrieves $b > 1$ clusters: if the first cluster found for f_i^n is one of the b clusters found for f_i^q , then the cluster index has done its job. And with soft assignment, where a feature is inserted into $a > 1$ cluster during indexing, it would be sufficient to find any of the top a clusters for f_i^n within the top b clusters for f_i^q , to guarantee that a query for f_i^q would retrieve f_i^n . This method for quality evaluation also applies to other approximate methods, such as BoW, although for methods that use approximate processing inside the cluster (or leaf) the quality given by our method represents an upper bound.

6.2. Quality Evaluation for Image Retrieval Benchmarks

While the quality of single feature retrieval is important, most of the available benchmarks are defined at the image level. In an image benchmark, a set of query images I_j^q and “original” images I_j^o are given. The query image is typically generated from the original image through some standard image manipulations (cropping, resizing, compression, etc), but sometimes the query image is in fact a different image with similar content. The originals can be embedded in a large collection of “distracting” images, to make the problem harder. The

Table IV. Index structures evaluated.

Clustering	Index Size (clusters)	Collection Size (billion features)	Index Shape	
			(depth)	(branching)
HKM-200K	200,000	3–10	3	58
DeCP-200K	200,000	3–10	3	59
DeCP-2M	2,000,000	30–100	4	38
DeCP-20M	20,000,000	300–1,000	5	29

goal of the image retrieval system is then to query for the query images I_j^q and retrieve the “originals” I_j^o .

Using image benchmarks complicates the evaluation of quality for unknown feature collections. For a pair of images (I_j^q, I_j^o) , there is no longer a clear connection between the features generated from each. First, the images may be so different that some features from the original have no counterpart of the query image, and some features in the query image have no counterpart in the original image. Second, even when features do correspond, they may be so different that even a sequential scan would not identify the original in a k -nearest neighbor retrieval for the original query feature f_i^q . Third, two features from different parts of the images may be very similar by pure happenstance. All of this makes it difficult to produce (f_i^q, f_i^n) pairs of ground truth features, and an approximation is needed.

To evaluate the quality of the index structure, using our methodology defined above, we propose to select, for each feature f_i^q from the query image I_j^q , the most similar feature f_k^n from the original image I_j^o :

$$f_i^n := f_k^n | \min_k (d(f_i^q, f_k^n))$$

Note that this is in fact exactly the feature from the original image that a sequential scan would be most likely to find.

6.3. Experiment

In this experiment we evaluate the (inevitable) quality loss that occurs due to the increased index size (both index depth and branching factor) as the collection grows towards web-scale. Using the SIFT1B feature collection as the source of centroids, we have built three progressively larger indices for DeCP, with the largest designed to index 300–1,000 billion features (15–50K features per cluster, on average). We have also used 2 million features to create an index using the well known hierarchical k -means algorithm (HKM) from the publicly available VLFeat library,² to serve as a baseline. Note that creating the HKM index took 5 hours, while even the largest DeCP index could be created in minutes. Table IV presents these indices in more detail.

We then used the evaluation methodology described above to evaluate the retrieval quality for the well-known CopyDays benchmark, which is a traditional copy-detection benchmark.³ It is known that a majority of the query features in CopyDays do not find a match, even at a smaller scale, due to the sometimes extensive transformation of the images. At the image level these matches add up, however, and the correct match is typically returned for 80–95% of the query images.

Figure 9 shows the quality for the three different DeCP index configurations, when retrieving $b = 1 \dots 5$ clusters. Due to the implementation of the VLFeat library, we could only retrieve $b = 1$ cluster using HKM (and only for the smallest hypothetical collection, as noted in Table IV) and the quality of HKM is therefore shown as a flat line.

²<http://www.VLFeat.org>

³<https://lear.inrialpes.fr/~jegou/data.php>

Figure 9 shows that, for the smallest index, DeCP yields nearly the same result quality for the first cluster as HKM, with a further 10% recall added for $b = 5$ clusters. Note that previous results show that detailed clustering algorithms, such as HKM, typically have some very large and dense clusters which are frequently retrieved as the first cluster, leading to relatively high cost of reading the first cluster and a poorer quality over time trade-off than the simple DeCP algorithm [Tavenard et al. 2011]. This is thus an excellent result for DeCP, as its index is built in minutes compared to hours for HKM.

Figure 9 also shows that most of the matches are found in the first cluster. As DeCP only selects additional clusters from the bottom branch the failure must lie higher in the structure. In future work, we plan to study the effect of soft-assignment in more detail.

As expected, there is some degradation of quality as the DeCP index structure grows deeper and wider. Overall, for $b = 5$, the 2M index retains 76.4% of the matches that the 200K index retrieved, and the 20M index retains 78.5% of the matches that the 2M index retrieved. We expect that HKM would show a similar degradation of quality for 2M and 20M indices. Due to its prohibitive running time and memory requirements, however, we could not confirm that expectation.

Considering that the 20M index has two additional levels in the index hierarchy, and can fit 100 times more data, this is actually a surprisingly good result. Overall, the results show that even with a collection that is roughly 20-30x larger than the one indexed in our prior experiments with DeCP, the index retains more than half of its result quality. Most importantly, however, we could make this estimate without the need to actually generate and index a collection of 1,000 billion SIFT features.

7. OPEN SOURCE PROTOTYPE

To facilitate applying our results in research and industry, we make the following contributions: (i) make publicly available all the code necessary to implement the pipelines described in this article; and (ii) create a virtual machine with DeCP search engine and web-interface pre-installed. Both available at <https://github.com/elgerpus/DeCP>

7.1. The DeCP Search Engine

The prototype search engine is an end-to-end system taking images as its input and producing text-based ranked results. SIFT features are extracted with the open-source Java-based BoofCV library (see Section 4.4). When the engine is started it is possible to temporarily add new images to the indexed collection, by utilizing Spark’s ability to join two RDDs (see [Guðmundsson et al. 2017]). Note that the new images will not become a permanent part of the index unless the new RDD is explicitly stored to disk.

To maximize flexibility, the search engine does not integrate the web-API but opts for a text-based input/output system. The input folder is monitored for new requests in the form of a *.batch* file. In such a batch file, the first line indicates colon-separated search parameters, while the remaining lines contain paths to query images. Special input files can be used as well, to halt the system a *halt.batch* is used and to save the current indexed RDD a *save.batch* files is used.

The search results are also presented as text files written to a specific output folder. For each query batch, a new sub-folder is created, containing a *batch.res* file with information on search settings and search duration in the first line, while the remaining lines are paths to

Fig. 9. Index retrieval quality for the CopyDays benchmark.

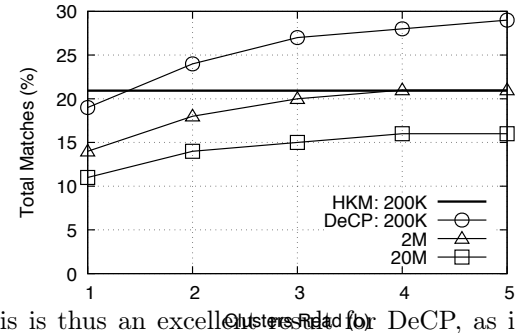
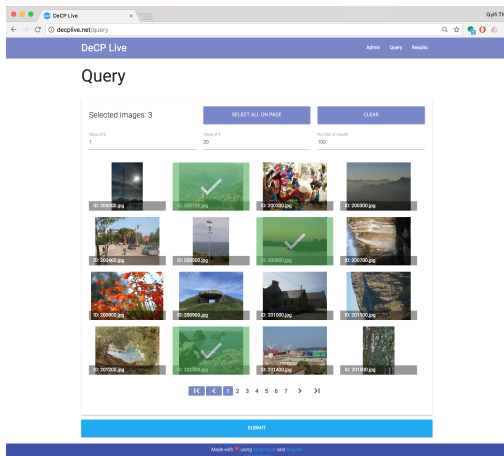
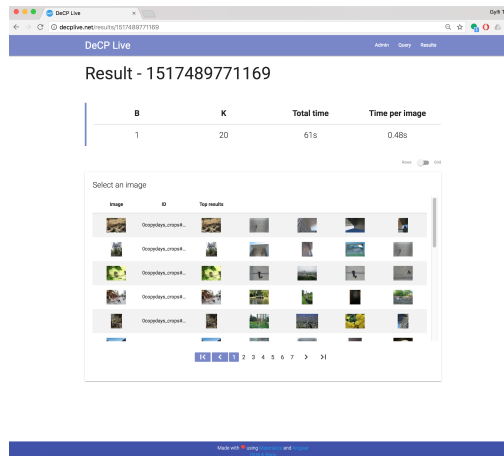


Fig. 10. Creating a batch in the web-interface.

Fig. 11. Listing the batch results in *Rows*-mode.

individual search results for each query image in the batch. The result file for a query image contains a path to the query image, and the number of SIFT features extracted from it, in the first line. The remaining lines then contain a ranked list of the most similar database images, along with the number of votes they received.

7.2. Browser-Based Search Interface

Figure 10 shows the web-interface for submitting batch queries. The user can supply arbitrarily many query images, as well as supply run-time settings for the search parameters k (neighbors returned for each query feature), b (clusters examined for each query feature), and the number of results per query image. Upon submitting the query images, a *.batch* file is written to the input folder where it will be picked up by the search engine. As soon as the search is done, the search results are written to the output folder, which in turn is monitored by the web-application. As soon as new results are detected, regardless of how the queries were submitted, the web-interface will notify the user that new search results are available.

Through the web-interface the user can browse all search results in the engines output folder, even those submitted by other means. In Figure 11 we see the interface after selecting a batch from the list of available batch results. We can browse in two modes, *Rows*- or *Grid*-mode, and the in the *Rows*-mode shown in the image we notice that in addition to the expected batch information we get a sneak preview the top-five results for each query image. The *Grid*-mode setting shows 16 query images per page.

Then, as the last result view, we have the result for each individual query image. In this view we see the query image at the top of the page and how many SIFT features were extracted from it. This view default to the *Grid*-mode with the larger thumbs. However, to provide maximal information, we show both the image ID and the #votes in an overlay on each ranked result image.

The web-interface also has an administrative window where the user can submit the special *halt.batch*, to halt the search engine, or the *save.batch* that will save the current indexed RDD.

8. CONCLUSIONS

In this article, we have argued the advantages of using automatically distributed computing frameworks (ADCFs) to implement throughput-oriented multimedia services, in order to cope with today's very large and ever growing multimedia collections. We have identified

five requirements that such ADCFs should satisfy, in order to effectively support multimedia services: Scalability; Computational flexibility; Capacity; Flexible pipelines; and Simplicity.

To the best of our knowledge, we have engineered the first (near) web-scale multimedia service running on Spark: a full-featured off-line copy detection system (with multi-probing, search-expansion and post-process re-ranking). We have detailed the Spark pipelines for index creation, batch search, and index maintenance, and also discussed how to implement many advanced CBIR approaches and extensions using Spark. We have then measured the performance of the prototype by conducting some of the largest experiments reported to date, using 43 billion SIFT descriptors from the YFCC100M collection. Finally, we proposed a new methodology for studying result quality at much larger scales, without actually creating and indexing the feature collection.

Our experiments have shown that Spark satisfies all five requirements identified for a high-throughput web-scale multimedia service. We therefore argue that designers of scalable multimedia services should strongly consider using Spark (or subsequent frameworks with similar capabilities) as the basis for their systems. In order to support that work, we have made all our code publicly available through a code repository, an open prototype, and a ready-made virtual machine.

REFERENCES

- L. Amsaleg. 2014. A Database Perspective on Large Scale High-Dimensional Indexing. Habilitation à diriger des recherches, Université de Rennes 1.
- R. Arandjelovic and A. Zisserman. 2013. All About VLAD. In *Proc. of the IEEE Int. Conf. on Computer Vision & Pattern Recognition*.
- A. Babenko and V. S. Lempitsky. 2015. The Inverted Multi-Index. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 37, 6 (2015).
- M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubsky, and P. Zezula. 2010. Building a web-scale image similarity search system. *Multimedia Tools and Applications* 47, 3 (2010).
- E. Y. Chang. 2011. *Foundations of Large-Scale Multimedia Information Management and Retrieval: Mathematics of Perception*. Springer, Berlin, Germany.
- J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008).
- R. K. Grace, R. Manimegalai, and S. S. Kumar. 2014. Medical Image Retrieval System in Grid Using Hadoop Framework. In *Proc. of the Int. Conf. on Computer Science and Computational Intelligence*.
- C. Gu and Y. Gao. 2012. A content-based image retrieval system based on Hadoop and Lucene. In *Proc. of the Int. Conf. on Cloud and Green Computing*.
- G. P. Guðmundsson, L. Amsaleg, B. P. Jónsson, and M. J. Franklin. 2017. Towards Engineering a Web-Scale Multimedia Service: A Case Study Using Spark. In *Proc. of the ACM Multimedia Systems Conf.*
- J. S. Hare, S. Samangooei, D. P. Dupplaw, and P. H. Lewis. 2012. ImageTerrier: An Extensible Platform for Scalable High-performance Image Retrieval. In *Proc. of the ACM Int. Conf. on Multimedia Retrieval*.
- S. Jai-Andaloussi, A. Elabdouli, A. Chaffai, N. Madrane, and A. Sekkaki. 2013. Medical content based image retrieval by using the Hadoop framework. In *Int. Conf. on Telecommunications*.
- H. Jégou, M. Douze, and C. Schmid. 2008a. Hamming embedding and weak geometric consistency for large scale image search. In *Proc. of the European Conf. on Computer Vision*.
- H. Jégou, M. Douze, and C. Schmid. 2008b. The Copydays image dataset. <http://lear.inrialpes.fr/people/jegou/data.php/#copydays>. (2008).
- H. Jégou, M. Douze, and C. Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 33, 1 (2011).
- H. Jégou, F. Perronnin, M. Douze, J. Sánchez, P. Pérez, and C. Schmid. 2012. Aggregating local image descriptors into compact codes. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 34, 9 (2012).
- H. Lejsek, B. P. Jónsson, and L. Amsaleg. 2011. NV-Tree: Nearest Neighbours at the Billion Scale. In *Proc. of the ACM Int. Conf. on Multimedia Retrieval*.
- T. Liu, C. Rosenberg, and H. A. Rowley. 2007. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *Proc. of the IEEE Workshop on Applications of Computer Vision*.

- D. G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *Int. Journal on Computer Vision* 60, 2 (2004).
- N. Marz and J. Warren. 2015. *Big Data: Principles and best practices of scalable real-time data systems*. Manning Publ. co, Shelter Island, NY, USA.
- D. Moise, D. Shestakov, G. P. Guðmundsson, and L. Amsaleg. 2013a. Indexing and Searching 100M Images with Map-Reduce. In *Proc. of the ACM Int. Conf. on Multimedia Retrieval*.
- D. Moise, D. Shestakov, G. P. Guðmundsson, and L. Amsaleg. 2013b. Terabyte-scale image similarity search: experience and best practice. In *Proc. of the IEEE Int. Conf. on Big Data*.
- P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. 2015. SparkNet: Training Deep Networks in Spark. *arXiv:1511.06051*. (2015).
- D. Nistér and H. Stewénus. 2006. Scalable Recognition with a Vocabulary Tree. In *Proc. of the IEEE Int. Conf. on Computer Vision & Pattern Recognition*.
- B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K.H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. 2015. SINGA: A Distributed Deep Learning Platform. In *Proc. of the ACM Int. Conf. on Multimedia*.
- S. Owen, R. Anil, T. Dunning, and E. Friedman. 2011. *Mahout in Action*. Manning Publ. co, Shelter Island, NY, USA.
- J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. 2007. Object retrieval with large vocabularies and fast spatial matching. In *Proc. of the IEEE Int. Conf. on Computer Vision & Pattern Recognition*.
- J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. 2008. Lost in quantization: Improving particular object retrieval in large scale image databases. In *Proc. of the IEEE Int. Conf. on Computer Vision & Pattern Recognition*.
- W. Premchaiswadi, A. Tungkatsathan, S. Intarasema, and N. Premchaiswadi. 2013. Improving performance of content-based image retrieval schemes using Hadoop MapReduce. In *Proc. of the Int. Conf. on High Performance Computing and Simulation*.
- D. Shestakov, D. Moise, G. P. Guðmundsson, and L. Amsaleg. 2013. Scalable high-dimensional indexing with Hadoop. In *Int. Workshop on Content-Based Multimedia Indexing*.
- K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *Proc. of the IEEE Symp. on Mass Storage Systems and Technologies*.
- J. Sivic and A. Zisserman. 2003. Video Google: A Text Retrieval Approach to Object Matching in Videos. In *Proc. of the IEEE Int. Conf. on Computer Vision*.
- X. Sun, C. Wang, C. Xu, and L. Zhang. 2013. Indexing billions of images for sketch-based retrieval. In *Proc. of the ACM Int. Conf. on Multimedia*.
- R. Tavenard, H. Jégou, and L. Amsaleg. 2011. Balancing clusters to reduce response time variability in large scale image search. In *Int. Workshop on Content-Based Multimedia Indexing*.
- B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. 2016. YFCC100M: The New Data in Multimedia Research. *Commun. ACM* 59, 2 (2016).
- A. Vedaldi and B. Fulkerson. 2010. VLfeat: An Open and Portable Library of Computer Vision Algorithms. In *Proc. of the ACM Int. Conf. on Multimedia*.
- H. Wang, B. Xiao, L. Wang, and J. Wu. 2015. Accelerating Large-scale Image Retrieval on Heterogeneous Architectures with Spark. In *Proc. of the ACM Int. Conf. on Multimedia*.
- B. White, T. Yeh, J. Lin, and L. Davis. 2010. Web-scale Computer Vision Using MapReduce for Multimedia Data Mining. In *Proc. of the Int. Workshop on Multimedia Data Mining*.
- Qing-An Yao, Hong Zheng, Zhong-Yu Xu, Qiong Wu, Zi-Wei Li, and Lifen Yun. 2014. Massive Medical Images Retrieval System Based on Hadoop. *Journal of Multimedia* 9, 2 (2014).
- M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the Symp. on Networked Systems Design and Implementation*.
- M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster computing with working sets. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing*.
- J. Zhang, X. Liu, J. Luo, and B. Lang. 2010. DIRS: Distributed image retrieval system based on MapReduce. In *Proc. of the Int. Conf. on Pervasive Computing and Applications*.