Scalable high-dimensional indexing with Hadoop

Denis Shestakov, Diana Moise, Gylfi Thór Gudmundsson, Laurent Amsaleg

To cite this version:

Denis Shestakov, Diana Moise, Gylfi Thór Gudmundsson, Laurent Amsaleg. Scalable high-dimensional indexing with Hadoop. CBMI—International Workshop on Content-Based Multimedia Indexing, 2013, Veszprém, Hungary. hal-00817378

HAL Id: hal-00817378
https://inria.hal.science/hal-00817378

Submitted on 24 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scalable high-dimensional indexing with Hadoop

Denis Shestakov*
INRIA Rennes, France
Aalto University, Finland
Denis.Shestakov@{inria.fr,aalto.fi}

Diana Moise
INRIA Rennes, France
Diana.Moise@inria.fr

Gylfi Gudmundsson
INRIA Rennes, France
Gylfi.Gudmundsson@inria.fr

Laurent Amsaleg
IRISA-CNRS, France
Laurent.Amsaleg@irisa.fr

***Abstract*—While high-dimensional search-by-similarity techniques reached their maturity and in overall provide good performance, most of them are unable to cope with very large multimedia collections. The 'big data' challenge however has to be addressed as multimedia collections have been explosively growing and will grow even faster than ever within the next few years. Luckily, computational processing power has become more available to researchers due to easier access to distributed grid infrastructures. In this paper, we show how high-dimensional indexing methods can be used on scientific grid environments and present a scalable workflow for indexing and searching over 30 billion SIFT descriptors using a cluster running Hadoop. Our findings could help other researchers and practitioners to cope with huge multimedia collections.**

I. Introduction

The last decade witnessed unprecedented progresses in the soundness and the quality of high-dimensional search-by-similarity techniques [1], [2], [3]. Research literature contains many elegant contributions overall spanning a quite large spectrum of multimedia applications dealing with still images, videos, audio, sometimes considering multimodality. Furthermore, the maturity of these techniques now allows startups and major software companies to build multimedia search engines that are profitable, money wise. Good performance is here today: systems are fast and provide reasonable precision/recall.

In parallel, the scale of multimedia collections has grown faster than ever. Nowadays photo/video-enabled cellular phones, Web2.0 and UGC as well as the entire social networks sphere are massive producers of multimedia contents, 24*7, all over the planet.

Having high-dimensional indexing schemes coping with this exponential growth of multimedia collections while preserving their good performance requires researchers to turn their interests to distributed computing platforms. Clusters of computers, a.k.a. grids, are good candidates for addressing this challenge. Benefiting from great processing power is compulsory since enormous collections of raw high-dimensional descriptors must be finely analysed at index creation time. At search time, great processing power is needed to provide throughput.

Dealing with distributed and parallel programming has always been complicated, mainly because synchronisation, scheduling, load balancing and fault tolerance are hard to achieve. Several frameworks, however, have been proposed to ease this programing, such as Dryad [4], GraphLab [5] or Map-Reduce [6]. These frameworks (almost completely) transparently handle these complicated issues, leaving programmers to solely focus on their tasks, not on the plumbing required by distribution. Yet, frameworks facilitating the programming of distributed and parallel tasks impose on programmers very strict constraints sometimes conflicting with the properties of their applications. For example, Map-Reduce imposes a flow of data that makes iteration-based or graph-based search algorithms hard to implement.

It is therefore not trivial to port any of the state of the art high-dimensional indexing techniques to a distributed environment. It often requires to change the design of the indexing algorithm itself to fit in the mold as well as to finely understand and then tune the many parameters of the framework facilitating distribution. This paper precisely discusses these issues, specifically focusing on the Map-Reduce programming model used together with the Hadoop runtime environment. The first contribution is the presentation of a scalable workflow for high-dimensional indexing and searching on a cluster running Hadoop. The second contribution is a large body of experiments allowing us to discuss the effects of various parameter settings on the performance of the algorithm. The lessons drawn from this work should jump-start other researchers working in that direction.

The paper is structured as follows. Section II gives an overview of Map-Reduce-based high-dimensional indexing scheme we designed and implemented for the Hadoop runtime environment. Section III details the problems we encountered when implementing and testing the indexing/searching workflow in our local grid environment. Performance results and lessons drawn are described in Section IV. Section V concludes.

II. Indexing & Searching Workflow on Hadoop

This section gives a short overview of the scalable, Map-Reduce oriented, high-dimensional indexing scheme we are using in this work. More details on this algorithm can be found in [7]. The indexing algorithm uses at its core a hierarchical unstructured quantization scheme, quite similar to the approach proposed by Nister and Stewenius [8]. In a nutshell, high-dimensional descriptors are clustered around randomly picked representative points that are hierarchically organized.

Creating the index is a multi-step process that starts with preparing the high-dimensional descriptors dataset for being used with the Hadoop framework and finishes with storing the final clusters in the distributed file system. The steps are:

* – This work, funded by INRIA, was done at INRIA research labs in Rennes while the author was on leave from the Aalto University.

1) Preparing the dataset for Hadoop compatibility: For best performance, the collection of high-dimensional descriptors must be turned into *Hadoop Sequence Files*, which is a specific file type for binary data. Each raw descriptor becomes a sequence file record. A record in this file has a key (an image identifier), and a value (a high-dimensional descriptor). Hadoop scatters across many machines the blocks of the sequence file in its distributed file system, HDFS [9].

2) Creation of the indexing dictionary: The indexing starts by randomly picking from the descriptor collection C points that create the representatives of the visual vocabulary dictionary eventually created. These representatives are organized in a hierarchy of L levels for efficiency. This creates the *index tree* for indexing. This tree is saved in a file in HDFS.

3) Distributed index creation: Creating an index from a very large collection of descriptors is a very costly process. This can take days or weeks when the descriptor collection is terabyte-sized. Distributing and parallelizing index creation on a grid is key for performance. We proceed as follows. A specific node of the grid, the job-tracker, launches on each machine participating to the index creation a series of *Map*-tasks. Each such task receives at startup time (i) a block of data from the previously created sequence file and (ii) the file containing the index tree. One task can thus assign the descriptors in its data block to the correct representative using the tree. The resulting assignments are sent to *Reduce*-tasks that write to disks high-dimensional descriptors grouped by cluster identifier. This eventually creates one or more *index files* which contain clustered high-dimensional descriptors. The number of such files depends on the number of *Reduce*-tasks ran, see [7].

Searching the index is also a multi-step process. This process is geared toward throughput as it processes very efficiently large batches of queries, typically 10^4–10^7 query descriptors. The steps for running a batch search are:

1) Creation of a lookup table: All query descriptors of a batch are first reordered according to their closest representative, which is known from traversing the index tree. A lookup table is then created, allowing to easily know which query descriptors have to be used in distance calculations when a cluster identifier is given. This lookup table is saved in an HDFS file.

2) Distributed searching: At search time, the job-tracker launches on machines *Map*-tasks for searching. Each such task receives (i) a block of data from one of the previously created index files, (ii) the file containing the lookup table. One task will use the cluster identifiers stored in the received block of data to lookup the query batch in order to identify the query descriptors having identical cluster identifiers. For those, distance calculations are performed, updating k-nn tables. k-nn results are eventually emitted, then aggregated to create the final result for the query batch.

Several comments are in order. First, the indexing and searching schemes briefly sketched above does approximate k-nn searches, trading result quality against dramatic response time improvements. Second, the schemes use quite a lot of *auxiliary* data, both during the creation of the index (this is the index tree) and during the search phase (this is the lookup table). Particularly, the index tree has to be loaded when processing *each* block of (not yet indexed) descriptors at assignment time, and the lookup table has to be loaded when processing *each* block of (clustered) descriptors at search time. As it will be detailed in the experimental section, this potentially puts some overhead on the computations since auxiliary data are large compared to the size of a block of data. Repeatedly loading large auxiliary data for every single (in comparison small) data block is costly. The performance evaluation section also shows the impact of the block size on performance.

III. Working with Grid'5000

For implementing the indexing/searching scheme sketched above we have been using computers belonging to the Grid'5000 project [10]. The Grid'5000 project is a widely-distributed infrastructure devoted to providing an experimental platform for the research community. The platform including twenty physically distinct clusters for a total of about 7000 CPU cores is spread over ten geographical sites located through the French territory and one in Luxembourg. To support experimental diversity, while allowing users to adapt the environment to their needs, Grid'5000 provides a deep reconfiguration mechanism that enables users to deploy, install, boot, run and finely monitor their customized software.

To execute the indexing/searching jobs, we developed a process that automatically deploys and configures the Hadoop (version 1.0.1) environment, transfers input data to HDFS and finally runs the indexing/searching jobs. Although this automated process is necessary to facilitate experimentation on Grid'5000, it is not sufficient to overcome the challenges of running at large scale, platform and data wise. While our experiences pertain to the Grid'5000, the challenges we have encountered are also relevant to grid environments in general. We typically faced the three following problems:

1) Heterogeneity of resources: Resource heterogeneity in a grid becomes a significant limitation when setting up the Hadoop environment. Since Hadoop settings can only be specified globally and not in a per-node manner, the Hadoop deployment is configured according to the least equipped node, at the expense of wasting resources on more out-fitted nodes. To be specific, heterogeneity of resources across clusters on the grid influences the number of map/reduce slots per tasktracker, and the amount of RAM memory allocated to each task; these parameters have to be set to the lowest available values.

2) Node failures: Node failures represent the daily norm in grid environments. Even though Hadoop is designed to cope with failures in a transparent manner, machine deaths can severely impact the whole deployment. On Grid'5000, we experienced from one to five node failures during a 60 hours run, some failures requiring a complete re-deployment to exclude the failed nodes. The worst failure outcome is losing the data on the machine. To avoid this, we used a replication data factor of 2 or 3. However, this is not always possible for very large datasets; factors such as storage, replication time, add up to substantial costs.

3) Deployment overhead: The deployment overhead for each experiment is substantial when running at large scales. Since the grid is a shared tool, it operates using reservations that allow users to employ resources for a certain time slot.

Cluster	#Nodes	#CPU@Freq	#Cores /CPU	RAM	Local Disk
Cl_1	64	2 Intel@2.50GHz	4	32GB	138GB
Cl_2	25	2 Intel@2.93GHz	4	24GB	433GB
Cl_3	40	2 AMD@1.70GHz	12	48GB	232GB

TABLE I. CLUSTER CONFIGURATIONS.

Environment deployment	10 min
Hadoop deployment	5 min
Data transfer to HDFS	90 min
Index creation	170 min
HDFS optimal chunk placement	30 min
Lookup table creation	3 min
Searching	5 min
Retrieving search results	5 min

TABLE II. TIME MEASUREMENTS FOR WORKFLOW STEPS.

After the reservation expires, all deployment data and setup are deleted. Consequently, creating the experimental environment, setting up the Hadoop cluster and making the data available in HDFS have to be repeated for every experiment.

IV. EXPERIMENTAL RESULTS

To evaluate the performance of the workflow presented in this paper, we ran a large series of experiments on the Grid'5000 testbed. We employed nodes belonging to the Rennes site, spanning three clusters described in Table I. Setting up a Hadoop cluster consisted of deploying each centralized entity (namenode and jobtracker) on a dedicated machine and co-deploying datanodes and tasktrackers on the rest of the nodes; typically, we used a separate machine as a Hadoop client to manage job submissions. In next two sections we describe our data collection and provide time observed for workflow operations on the grid. We then proceed to presenting performance results of indexing and searching.

A. Dataset description

In our experiments we used the dataset provided by one of our partners in the Quaero project, http://quaero.org/. The dataset consists of around 30 billion SIFT descriptors extracted from 100 million images (resized to only 150 pixels on their largest side) harvested on the Web. This amounts to four terabytes of data. To facilitate experiments, we use not only this entire collection, but also a subset of it containing 20M images, i.e., 7.8 billion descriptors or about one terabyte on disks. To best of our knowledge, these datasets are among the largest collections presented to the image retrieval community.

For evaluation of the indexing quality, we used the above mentioned data collection as a distracting dataset into which the INRIA Copydays evaluation set had been drawn [11]. Of course, images were resized as distractors were. We considered a copyright violation detection scenario, where we included 127 original images in the indexed database and queried it for 3055 associated generated variants (crop+scale, scale change+jpeg compression and manually generated strong distortions such as print-crumple-scan). We then count how frequently the original images are returned as the top result. We additionaly randomly picked a large number of images from the collection and created 49 Stirmark-based variants for each image. Overall, this series of queries added to the Copydays ones makes it possible to increase the load at query time by submitting large query batches. In this paper, we typically use one query batch solely made of the Copydays queries and another query batch that includes Copydays but contains overall 12,000 images.

Nodes	Default Hadoop(min)	Tuned Hadoop(min)
50	202	174.7
106	95	69

TABLE III. INDEXING TIME.

Parameter	Default value	Tuned value
#Map slots/tasktracker	2	8
#Reduce slots/tasktracker	2	8
Input data replication	3	3
Output data replication	3	1
Chunk size	64 MB	512 MB
JVM reuse	off	on
Map output compression	off	on
Reduce parallel copies	5	50
Sort factor	10	100
Sort buffer	100 MB	200 MB
Datanode max. receivers	256	4096
Namenode handlers	10	40

TABLE IV. HADOOP CONFIGURATION TUNING.

B. Time evaluations for workflow operations on Grid'5000

A typical experiment involving indexing and searching 1 TB of data on 50 Grid'5000 nodes requires a time frame of 5-6 hours. To better analyze this time-consuming process, we divide it into several steps and we provide the amount of time allocated to each step, see Table II.

The first step of the workflow accounts for deploying and configuring the execution environment: creating an isolated environment on Grid'5000, starting Hadoop processes, launching monitoring tools to collect information about the platform usage and job statistical data. As Table II shows, a substantial amount of time is spent on copying the data from local storage to HDFS. Note that the data is replicated 3 times to favor local execution of map tasks. The index creation process produces indexed data stored in HDFS with a replication factor of 1. In order to allow the searching step to benefit from data locality, we perform an additional step to increase the replication factor to 2 and also to change the chunk size, if needed. After searching completes, the results are collected, together with statistical information.

C. Index creation

In a first set of experiments, we focused on determining the configuration settings that best suit our workload. Given the scale of our experiments, we also investigated the tuning of Hadoop configuration parameters to improve the performance. This first round of tests was carried out on the 20 million images dataset. Table III provides the indexing time using the default Hadoop configuration as well as the tuned one. For both runs, the number of map slots per task is set to 8 and the HDFS chunk size is set to 512MB.

The results in Table III show that setting the right parameters for the Hadoop framework reduces execution time with by 27 minutes. Tuning the parameters of the Hadoop framework is a rather complex task that requires good knowledge and understanding of both the workload and the framework itself. In Table IV we provide a list of values that improved performance of the indexing process. To start with, compression of map output data led to reducing the amount of shuffled data by 30% (from 1 TB to 740 GB) which resulted in

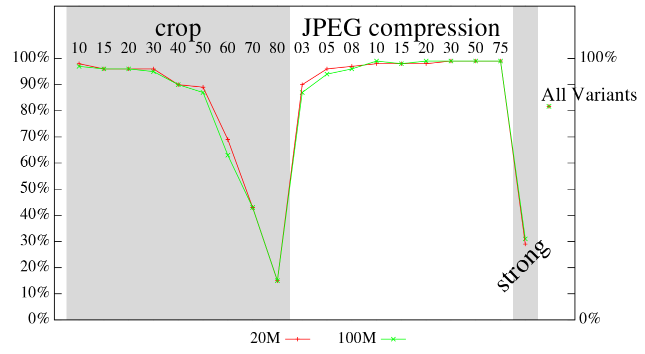

Fig. 1. Search Quality, Copydays evaluation set.

significantly reduced network traffic. Another factor with a significant impact on performance is the HDFS chunk size. However, since the indexing process is highly CPU-intensive, adjusting this parameter did not impact the execution time. Other low-level parameters refer to the shuffling phase, when the data is sorted on the mapper's side, copied to the reducer and then merged and sorted on the reducer node. The last two lines of Table IV show options configured on the HDFS nodes: the maximum number of chunk requests (read/write) each namenode can simultaneously handle, and the number of connections the namenode can serve at a time. A configuration parameter not included in the table, refers to *rack-awareness*. This parameter enables Hadoop to map its deployment to the network topology; this mapping is used in replica placement and task scheduling. For our experiments, we did not configure racks, since the nodes we employed are all connected to the same switch.

Given the performance gains obtained, tuning the parameters listed in Table IV is worth-while; nevertheless, the values delivering best performance are highly dependent on the workload, and thus, tweaking them in order to discover the best values is advisable.

In the second phase, we index the whole 4 TB dataset, using the configuration settings previously discovered. Because of RAM limitations, we decreased the number of map slots to four and reduce slots to two. With this settings, the indexing on 100 nodes took 8 hours 27 minutes to complete.

D. Searching

We divided our grid experiments on searching the data collection with batches of query images into five parts. First, we evaluated the quality of search results by searching the query batch across the full dataset (100M images, 4TB size) and 20% of it (20M, 1TB size). We then evaluated the performance of search on 20M dataset. Our specific goals were to test the search performance depending on the number of nodes available (i.e., batch search scalability) and to analyze the batch search execution. Next we experimented with the best parameters for storing the indexed collection on the grid (particularly, influence of HDFS block size on search performance). Finally, we evaluated the throughput of our batch search.

1) Exp #1: Quality Measurement: The first experiment we show here reports quality measurements results proving the indexing technique returns high-quality results. To this

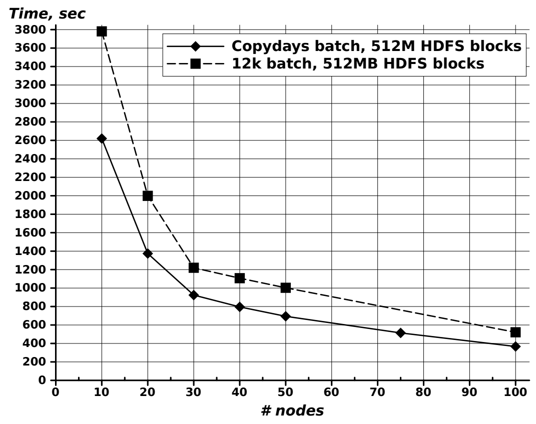

Fig. 2. Batch search scalability for Copydays and 12k batches.

end, we used the Copydays dataset presented above drawn in two distracting collections comprising 20M and 100M random images from the Web.

Figure 1 shows the quality results of the search. This Figure plots for every family of variants the percentage of original images found at rank 1, for the two distracting collections. It also plots the average percentage across all variants at the far right end of the Figure. From the Figure, it is clear that our indexing scheme returns high quality results, except for some severely attacked images such as when 80% of the image is cropped+rescaled to its original size or when strong manual distortions are applied. It is interesting to observe search quality does not significantly degrades when the size of the distracting dataset increases. Overall, 82.68% of Copydays variants are found when drowning them in 20M images, and we find 82.16% of them when drown in 100M images. This is a clear assessment that our indexing technique is very viable.

2) Exp #2: Scalability of batch search: We conducted series of batch searches over 20M dataset (stored in HDFS using 512MB blocks) on varying number of nodes, from 10 to 100 nodes. We used two different batches: with 3055 query images (Copydays) and with around 12,000 images (12k). Due to node/cluster availability at the time of experiments, only nodes in reservations with 40 and 50 nodes were from the same Grid'5000 cluster, Cl_1 (see Table I). While in 10-, 20- and 30-machine reservations nodes belonged to clusters Cl_2 and Cl_3, nodes of 75- and 100-node reservations were from three Grid'5000 clusters. Particularly, some nodes in 10-, 20-, 30-, 75- and 100-machine reservations were expectedly underperforming as the Hadoop had to be configured according to the nodes with minimal RAM, number of cores, etc (see Section III). The results are shown in Figure 2. In general, despite node heterogeneity, the proposed searching scheme scales well for both average-size and large-size batches: searching the same 1TB collection took approximately 7.2 times faster on 100 nodes than it took on 10 nodes.

3) Exp #3: Understanding batch search execution: We continued with experiments to reveal how the Hadoop framework executes the batch search job by obtaining basic information (such as start time, finish time, duration, node name) on every mapper.[1] We performed search for 12k batch over 20M

[1]Hadoop job history logs supporting Exp #3 and Exp #4 are available at http://goo.gl/e06wE.

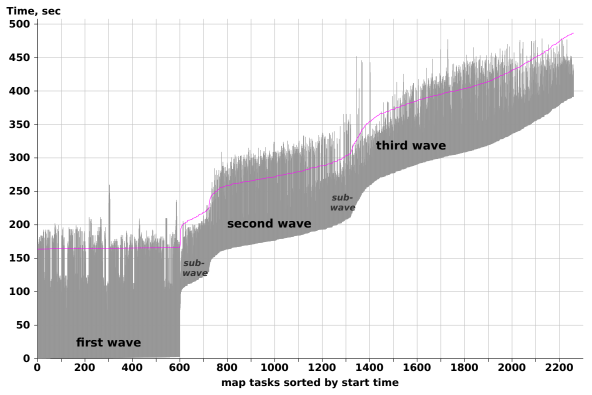

Fig. 3. Time progress of map tasks for searching 12k batch over 1TB (in 512MB blocks).

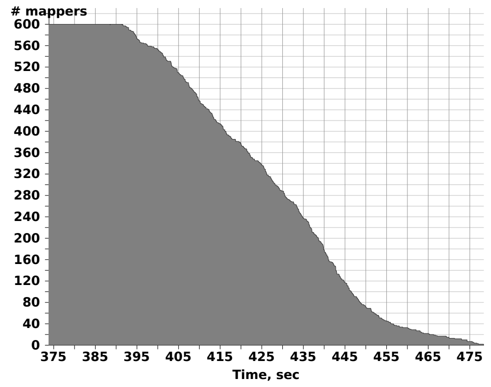

Fig. 4. Number of running map tasks during searching 12k batch over 1TB (in 512MB blocks) from 375s to 478s.

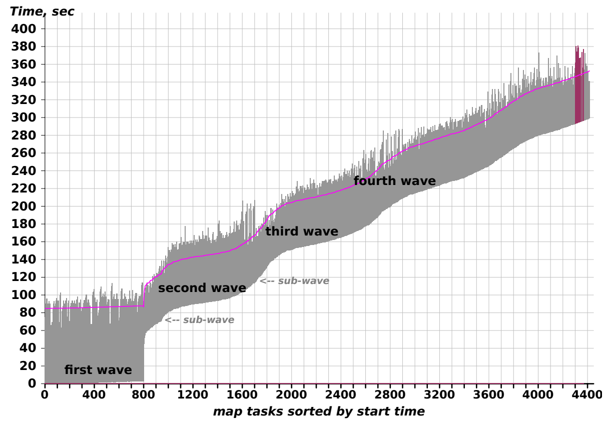

Fig. 5. Time progress of map tasks for searching Copydays batch over 1TB (in 256MB blocks).

dataset stored in HDFS using 512MB block sizes on 100 nodes, each configured to run six map tasks simultaneously. Figure 3 shows execution duration for every mapper, where all, 2260, map tasks are sorted by their starting times with their durations depicted as vertical lines plotted on Y-axis based on their absolute starting and finishing times. For convenience of identifying map tasks outliers we added the line showing average finishing times for mappers of the first and subsequent waves. One can see that the Hadoop jobtracker took 2.8s to start execution of the first 'wave' of mappers (where a wave capacity is defined by $N_{nodes} * n_{mappers}$, i.e., 600 map tasks per wave in this experiment). Overall this batch search job had four waves of mappers, 3x600 mappers plus the fourth wave with 460. However, only the first three are easily recognizable in Figure 3. Since mappers run on nodes with different performance (see Table I) and, on top of that, the amount of computations performed varied for every map task, there is a significant variance in map duration times which eventually led to the degradation of mapper waves. In addition, variations in node performance explain 'sub-waves' (created by mappers on the same cluster's nodes), two of which are also identifiable on the plot (see Figure for mappers at range 600-750 and 1200-1400). For instance, performance of mappers at 600-750 is due to fast nodes (of the Cl_2 cluster, see Table I) that finished processing some of their first six data blocks earlier than others and hence were first to start running the second wave of mappers. Spikes at around 300, 600, 1350-1400 are of particular interest as these represent the longest mappers in this search execution – they are caused by the worst-performing nodes of Cl_1 that were processing blocks with maximum distance calculations required. Comparing map tasks in the first and the three subsequent waves reveals another important aspect: all mappers of the first wave as being first had to load the lookup table by reading it from the HDFS, while the next waves' mappers have this table cached in the memory. In this way, we observe a substantial difference among average duration of map tasks in the first and three subsequent waves, namely 164s and 95s respectively.

For more careful analysis of this batch search, we plotted overall number of mappers running from 375s to 478s in Figure 4. As every node was configured to run six mappers at most, the maximum number of map tasks cannot exceed 600. Starting from 392s we observe stable decreasing in number of running mappers from 600 to 1 as all map tasks had been already assigned to nodes and thus a node that just finished a map task had nothing to run next. The time range from 392s and till finishing time of very last mapper, 478s, is the under-performing period, i.e., within this time fast nodes became idle first and stop to contribute to the overall search performance.

For better illustration of map wave concept we built another plot (see Figure 5) showing map task execution duration for Copydays batch search over 20M dataset (in 256MB blocks). Similar to 12k-512MB-search workload four (out of six) mapper waves are easily recognizable: in this workload there are 800 mappers in first five waves and 417 mappers in the last wave. The computational workload of every mapper, however, significantly decreased as not only there are approximately two times less points to be processed in each block due to two times smaller block size, but also the number of query descriptors in the batch became substantially smaller. Unlike 12k-512MB-search workload, last mappers of Copydays-256M-search are responsible for a delay in overall execution time: indeed, while the slowest mapper of last wave (among marked mappers at 4200-4400 in Figure 5) started 5.3s seconds earlier than the very last mapper, it finished the last, 34 seconds later than expected on average.

4) Exp #4: Most profitable HDFS block size: We continued with experiments on defining storage parameters, particularly, what HDFS block size may lead to better search performance. On the one hand, since processing of every HDFS block requires a corresponding map task to load the entire lookup table,

Batch	HDFS block size, MB	# blocks	**Search time, s**	Map tasks duration, s			
				avg	min	max	median
Copydays	256	4417	413	59	15	111	57
12k			863	106	24	390	72
Copydays	512	2260	382	99	18	291	99
12k			521	113	30	258	114
Copydays	640	1832	406	125	21	213	132
12k			545	130	30	325	126
Copydays	1024	1186	398	174	21	337	192
12k			554	187	33	340	186

TABLE V. SEARCH FOR COPYDAYS AND 12K BATCHES OVER 1TB WITH DIFFERENT HDFS BLOCK SIZES.

increasing HDFS block size and hence decreasing the number of HDFS blocks should lead to faster searches. On the other hand, increasing block sizes slows down mappers (as they need to read more data and perform more distance calculations). This, together with decreasing total number of mappers, can result in non-optimal[2] map task assignments by the Hadoop jobtracker. We make the deployment on 100 nodes and cloned the 20M collection in HDFS using four different block sizes, namely 256MB (recommended by Hadoop for large datasets), 512MB, 640MB, and 1024MB. We then searched through each collection-clone with Copydays and 12k batches. The results are summarized in Table V, where the third column indicates the total number of blocks used for storing a collection-clone on HDFS, and the fifth column contains average, extreme (fastest and slowest) and median durations of map tasks. Fastest mappers are those that processed smallest blocks[3] – since our indexed dataset consisted of 200 files, around 200 blocks were smaller than the defined HDFS block size and hence were processed significantly faster on average. Life-span of the slowest mapper defines a lower bound for overall search run time for a given HDFS block size – disregarding the number of nodes available, for search completion one of the nodes should run the slowest map task. In the worst case, the slowest map task is started among the mappers of last wave and severely delays the overall search time.

5) Exp #5: Batch search throughput: Finally we evaluated the throughput of our batch search, i.e., average processing time per image for different batch sizes. As a lookup table is loaded entirely into memory by every mapper, a big batch limits the number of mappers that can run in parallel, which, when $n_{mappers} < 0.75 \times n_{cores}$, increases search times (and hence decreases the throughput rate). At the same time, search runs are dominated by I/Os and, thus, big lookup tables can be preferable until a certain size when CPU work on distance calculations surpass I/Os. We searched two batches, Copydays and 12k, over 100M dataset on 87 nodes (of three clusters, see Table I). Search times suggest that 12k batch provide a stable throughput, around 210ms per image (more than 2 times better than the observed throughput for Copydays search, 460ms per image), for searching over 100 million images. The limitations for throughput when processing the 100M dataset is linked to the lowest amount of RAM available among all nodes. As 18 (out of 87) nodes had only 24MB RAM, we had to set the maximum number of mappers, namely, six for 12k batch, based on these nodes. As a result, other nodes (with 32 and 48MB RAM), which could handle at least eight simultaneous mappers rather than six, were under-performing and unable to improve the throughput.

[2]Ideally, the last wave of map tasks on all nodes should finish almost simultaneously. Achieving this is easier for the job scheduler if lots of mappers have short life-spans.

[3]The last block of a file in HDFS is of varying size and smaller than the configured HDFS block size. For example, with HDFS block size set to 256MB, the size of a last block of 520MB file is 8MB.

V. CONCLUSIONS

This paper presented a scalable workflow for high-dimensional indexing and searching on a cluster running Hadoop. Besides its scalability benefit, the proposed scheme not only achieves good search quality but also allows to search 100M image collection with a stable throughput of around 210ms per image. We particularly focused on the issues essential for porting a high-dimensional indexing technique to a distributed grid platform. We described a wide collection of experiments and the practical lessons we have drawn from our experience with the Hadoop environment. Our specific recommendation here is to tune Hadoop configuration parameters to a specific workload as it is very beneficial performance-wise. In addition, analyzing Hadoop job execution pinpoints the steps for further improvement: e.g., we plan to avoid loading the index tree for each mapper, by implementing multi-threaded map tasks that can utilize the full processing power at lower RAM usage. On the searching part, RAM usage can also be improved by a partial loading of the lookup table. As future work, we plan to conduct experiments with Hadoop deployments spanning multiple Grid'5000 sites. Although network bottlenecks are inevitable, the extra computing power will allow us to test scalability beyond 100 nodes.

ACKNOWLEDGEMENTS

This work was partly achieved as part of the Quaero Project, funded by OSEO, French State agency for innovation. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed (see http://www.grid5000.fr/ for details).

REFERENCES

[1] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. on PAMI*, 2011.

[2] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISAPP*, 2009.

[3] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, 2007.

[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, 2012.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, 2008.

[7] D. Moise, D. Shestakov, G. Gudmundsson, and L. Amsaleg, "Indexing and searching 100M images with Map-Reduce," in *ICMR*, 2013.

[8] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *MSST*, 2010.

[10] R. Bolze, F. Cappello, E. Caron *et al.*, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *Int. J. HPC Appl.*, vol. 20, no. 4, 2006.

[11] H. Jégou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *ECCV*, 2008.