Indexing and searching 100M images with map-reduce — Source link

Diana Moise, Denis Shestakov, Gylfi Gudmundsson, Laurent Amsaleg

Institutions: French Institute for Research in Computer Science and Automation, Centre national de la recherche scientifique

Published on: 16 Apr 2013 - International Conference on Multimedia Retrieval

Topics: Scalability, Search engine indexing and Index (publishing)

Indexing and Searching 100M Images with Map-Reduce

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

Indexing and Searching 100M Images with Map-Reduce

Diana Moise
INRIA Rennes
Diana.Moise@inria.fr

Denis Shestakov
INRIA Rennes
Denis.Shestakov@inria.fr

Gylfi Gudmundsson
INRIA Rennes
Gylfi.Gudmundsson@inria.fr

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

ABSTRACT

Most researchers working on high-dimensional indexing agree on the following three trends: (i) the size of the multimedia collections to index are now reaching millions if not billions of items, (ii) the computers we use every day now come with multiple cores and (iii) hardware becomes more available, thanks to easier access to Grids and/or Clouds. This paper shows how the Map-Reduce paradigm can be applied to indexing algorithms and demonstrates that great scalability can be achieved using Hadoop, a popular Map-Reduce-based framework. Dramatic performance improvements are not however guaranteed a priori: such frameworks are rigid, they severely constrain the possible access patterns to data and scares resource RAM has to be shared. Furthermore, algorithms require major redesign, and may have to settle for sub-optimal behavior. The benefits, however, are many: simplicity for programmers, automatic distribution, fault tolerance, failure detection and automatic re-runs and, last but not least, scalability. We share our experience of adapting a clustering-based high-dimensional indexing algorithm to the Map-Reduce model, and of testing it at large scale with Hadoop as we index 30 billion SIFT descriptors. We foresee that lessons drawn from our work could minimize time, effort and energy invested by other researchers and practitioners working in similar directions.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

Keywords

High-Dimensional Indexing, Map-Reduce, Hadoop.

1. INTRODUCTION

Multimedia collections now reach sizes that were unthinkable a few years back. Many papers published lately in multimedia research venues have experimental sections where the collections used for evaluations contain millions of images or billions of descriptors [9, 17, 12]. The quest for improving speed, scale and performance is still there, however.

The processing power of each computer has also grown, as well as the size of grids and now clouds. Architectures are now all 64bits, allowing for huge on-board RAM capacities, tens of gigabytes are not uncommon, and hundreds possible. All computers now come multi-cored and thus writing parallel programs is no longer reserved to some elite, equipped with exceptional machines.

Such architectures are appealing when processing massive collections of multimedia material, especially when *creating* high-dimensional indices. Taking a raw collection of high-dimensional descriptors and creating for it an index to allow subsequent ultra-fast searches is still a long, complex, costly, and resource-consuming task. When the raw descriptor collection is on the order of terabytes, as is the case when indexing tens of millions of real world image using SIFT [18], then indexing may take days or even weeks. Parallel and distributed architectures are also needed when *searching* indices in order to exhibit a very high query throughput.

This paper describes a complete Map-Reduce based high-dimensional indexing approach running on top of the Hadoop distributed framework. It builds on a very simple yet very effective and efficient state-of-the-art centralized indexing algorithm that is prototypical of the major trends we can observe in the indexing literature [8]. It uses *clustering* as an unstructured vectorial quantizer to create clusters grouping descriptors. At search time, one or few clusters are fetched and the descriptors they contain compared to the query vector. From this starting point, the three contributions described in this paper are:

1. **Index creation with Map-Reduce.** We propose a Map-Reduce based high-dimensional index creation scheme enabling the fast indexing of billions of descriptors, terabytes of data. This index creation technique scales very well with growth of the data collection and/or available hardware resources. Its principles are very simple and can be applied to other indexing schemes as long as they can be turned into a divide-and-conquer process.

2. **Index search with Map-Reduce.** For some applications, throughput is way more important than is the response time of individual queries. We propose a index search scheme that is geared toward throughput as it processes very efficiently large batches of queries. We show this search technique is essentially bounded

by the performance of I/Os, which leaves a lot of room for possible improvements.

3. **Very large scale experiments.** We evaluate index creation and search using an image collection containing roughly 100 million images, this is about 30 billion SIFT descriptors or about 4 terabytes of data. The extensive experiments ran on this very large-scale dataset provide a basis for a discussion on the problems raised when managing so much data with Hadoop and a grid. We thus draw several lessons we want to share.

The paper is structured as follows. Section 2 gives the necessary background to understand the remainder of this paper. Section 3 presents the high-dimensional indexing scheme we start from and that we extend to fit with the Map-Reduce paradigm. These extensions are described in details in the Section 4. Section 5 presents the context of the experiments and implementation details. Then, Section 6 gives the performance results. Section 7 concludes.

2. BACKGROUND

This section briefly describes the recent evolutions in hardware making possible the use of Map-Reduce as well as distributed and parallel frameworks dedicated to grid and cloud computations such as Hadoop. We also briefly mention the state-of-art in high dimensional indexing techniques.

2.1 Hardware

Multi-core CPUs have been around for a long time in specialized systems. But around 2005, multi-core architectures hit the mass market and everyone could have a machine with multiple cores, by default. Today, a standard desktop has 2–4 cores, and a server has 1–4 CPUs, each with 2–24 cores.

The performance of the applications depends of course on the number of cores they use for their computations, on the nature of the computation, as well as on their access pattern to data. High-performance computing typically has complicated iterative/recursive computations using the same data, or just-created data, from one iteration to the other. Once the calculation starts from the data in memory, it proceeds and runs instructions on the data that is cached very close to the processor, requiring to access RAM only every once in a while and, when the calculation is over, to save the final result. In this case, the CPUs are kept busy while the underlying hardware is not stressed.

This is not at all the case for high-dimensional indexing and retrieval applications for which consuming data is central. All the work is data-driven, not calculus-driven. Data is fetched from disks, then installed in memory and then transfered to the processor where it does not stays that long before being moved back elsewhere. Once processed, data has to follow the inverse route at index creation time, as each data item is then typically assigned to a group of similar elements stored on disks and later used during retrieval. Data locality and the underlying hardware are therefore key for the performance of such *big data* applications.

2.2 Map-Reduce

The Map-Reduce framework is originally by Google [4] and it is a programming model for processing extremely large datasets. It exploits data independence to do automatic distributed parallelism. The developer is tasked with implementing the *Map* and the *Reduce* functions. The input data is distributed in blocks to the participating machines using the distributed Google file system GFS [6].

When a job is launched, the system automatically spawns as many Map functions as there are data blocks to process. Each mapper reads the data iteratively as a key/value pair record, processes it and, if necessary, outputs key/value pair bound for a Reduce function. All records with the same key go to the same Reduce task. The framework thus includes a copy-merge-sort data shuffle step, where data from several mappers gets directed to specific reducers depending on their key. Once enough data is locally available to reducers, they process the records and produce the final output.

The Map-Reduce run-time environment transparently handles the partitioning of the input data, schedules the execution of tasks across the machines and manages the communications between processing nodes when sending/receiving the records to process. The run-time environment also deals with node failures and restarts aborted tasks on nodes, possibly on replicated data in case of unavailability. The framework uses as little network bandwidth as possible by processing data where it resides or at the nearest available node, paying attention to the network topology and minimizing reading over machine-rack boundaries.

2.3 Hadoop and HDFS

The Map-Reduce programming model has been implemented by the open-source community through the Hadoop project.Maintained by the Apache Foundation and supported by Yahoo!, Hadoop has rapidly gained popularity in the area of distributed data-intensive computing. The core of Hadoop consists of the Map-Reduce implementation and the Hadoop Distributed File System (HDFS). Hadoop is now the de-facto reference Map-Reduce implementation.

The architecture of Hadoop consists of a single master *jobtracker* and multiple slave *tasktrackers*. The jobtracker's main role is to act as the task *scheduler* of the system, by assigning work to the tasktrackers. Each tasktracker has of a number of available *slots* for running tasks. Every active map or reduce task takes up one slot, thus a tasktracker usually executes several tasks simultaneously.

When dispatching map tasks to tasktrackers, the jobtracker strives at keeping the computation as close to the data as possible. This technique is enabled by the data-layout information previously acquired by the jobtracker. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes closer to the data (belonging to the same network rack). The jobtracker first schedules map tasks, as the reducers must wait for the map execution to generate the intermediate data. The jobtracker is also in charge of monitoring tasks and dealing with failures.

HDFS [22] was built with the purpose of providing storage for *huge files* with *streaming data access patterns*, while running on clusters of *commodity hardware*. HDFS implements concepts commonly used by distributed file systems: data is organized into files and directories, a file is split into fixed-size blocks that are distributed across the cluster nodes. The blocks are called *chunks* and are usually of 64 MB in size (this parameter specifying the chunk size is configurable).

The architecture of HDFS consists of several *datanodes* storing the data chunks and a centralized *namenode* responsible for keeping the file metadata and the chunk location. HDFS handles failures through chunk-level replication (default 3 replicas). When distributing the replicas to the datanodes, HDFS employs a rack-aware policy: the first replica is stored on a datanode in the same rack, and the second replica is shipped to a datanode belonging to a different rack (randomly chosen).

In addition to being used in cluster computing, Hadoop is becoming a de-facto standard for cloud computing. The generic nature of clouds allows resources to be purchased on-demand, especially to augment local resources for specific large or time-critical tasks. Several organizations offer cloud compute cycles that can be accessed via Hadoop. Amazon's Elastic Compute Cloud contains tens of thousands of virtual machines, and supports Hadoop with minimal effort.

2.4 Image retrieval

Content-based image retrieval systems can now manage collections having sizes that could not even be envisioned years back. Most systems can handle several million images [16, 12], billions of descriptors [17, 13], or address web-scale problems [5, 1]. The ImageTerrier platform [9] uses Hadoop. The scale of their system is smaller since they index about 10 million images with a bag of feature approach.

Overall, most high-dimensional indexing schemes use some form of partitioning where the data is split into little groups at indexing time, and one or few such groups are loaded and analysed at retrieval time. All indexing schemes differ by the technique they use to create groups. They typically rely on scalar quantization (such as LSH [7, 3]) or on vectorial quantization (this is VideoGoogle [23] and all its derivatives, including [9]). The algorithm we use in this paper belongs to this second category, as it is presented next.

3. EXTENDED CLUSTER PRUNING AS A STARTING POINT

We decided to build on top of the extended Cluster Pruning (eCP) algorithm [8] for several reasons we highlight at the end of this section. eCP is a centralized high-dimensional indexing strategy. eCP is very related to the well-known k-means approach. As k-means, eCP adopts an unstructured quantization scheme to create clusters containing similar descriptors. eCP is designed to be I/O friendly as it assumes the data collection is too large to fit in memory and must reside on secondary storage.

3.1 Indexing and Searching with eCP

eCP randomly picks C points from the collection that are used as the representatives of the C clusters the algorithm will eventually build. C is determined from having set the average number of data points each cluster should contain. This number is called the TargetSize, ts, and $C = N/ts$ where N is the number of points in the collection. eCP then organizes the C representatives in a multi-level hierarchy composed of L levels. The points from the data collection that remain are read one after the other, traverse the tree of representatives and are eventually assigned to the closest cluster representative at the bottom of the tree. The multi-level hierarchy allows to assign points with a logarithmic complexity. Once all the data collection has been processed, then eCP has created C clusters as well as a tree of representatives, all this being stored on disk. Note the tree of representative is rather small and can fit in main memory.

Searching with eCP requires to navigate down the tree of representatives by following the path indicated at each level by the representative that is the closest to the query point. Then, the corresponding bottom cluster is fetched, and the distances between the query point and all the points in that cluster are computed to get the k-nearest neighbors.

eCP compensates its somewhat brutal clustering by adopting ideas from various state-of-art indexing schemes. It uses a form of soft-assignment [2, 21] while building the tree of representatives. With soft-assignment, each representative is not solely assigned to its closest parent representative, but it is assigned to its a closest representatives. Note a applies only to the tree of representatives, not to the data stored in the clusters. It also uses a form of multi-probe approach at search time as more than one cluster can be searched, as it has been proposed for LSH [19, 15]. eCP can probe the b clusters that are the closest to the query point.

3.2 Motivation for eCP

We port eCP to Map-Reduce and not another state-of-the-art indexing solutions for the following reasons:

- eCP is quite representative of the core principles underpinning many of the unstructured quantization-based high-dimensional indexing algorithms that perform very well [23, 11].

- eCP is not iterative by nature while traditional indexing schemes based on k-means are. At every iteration of a k-means process, and in order to eventually converge, new representatives must be computed based on the previous round. Distributing k-means or any other algorithm that needs rounds to converge is costly as a global state must be reconstructed and propagated to all participants. Having no such rounds with eCP was a strong motivation for using this algorithm, as distributing and parallelising it were greatly simplified.

- eCP pre-calculates a representative hierarchy that is used to significantly speed-up the assignment of points to clusters. This is key for performance when data collections are terabytes sized in order to have a indexing approach usable *in practice.*

- eCP proved to return good quality result despite the crude process it uses to create clusters. This is shown in the experimental section of this paper.

- Due to the extreme simplicity of its search procedure, eCP indeed covers a large spectrum of existing indexing approaches. A behavior very similar to the one of VideoGoogle [23] can be obtained if instead of computing the distances to all the points in the fetched cluster(s) eCP simply returns the cluster identifier, as each cluster is indeed a visual word. It can also behave quite similarly to the vectorial variant of LSH when it processes the contents of the clusters [20]. eCP is also compatible with the best indexing solutions that are known today and that rely on some form of smart descriptor aggregation [12].

4. MAP-REDUCING eCP

In this section we present how can eCP be adapted to fit with the Map-Reduce paradigm. We first describe the index creation and then move to describing the search process.

4.1 Index Creation with Map-Reduce

The index creation process of eCP can be split into two main phases. During Phase #1, the creation of the index tree, cluster representatives are picked from the collection and organized in a in-memory tree. During Phase #2, vectors are assigned to clusters.

Obviously, Phase #2 is the prime candidate for parallelization and distribution. It is clear that chopping the entire data collection into independent parts assigned to physically distinct nodes is going to speed up the whole process.

In contrast, Phase #1 is computationally cheap and requires no distribution. Therefore, picking C random points and building the in-memory hierarchy is done on a single machine once ts, a and L have been set. The resulting hierarchy of representatives is then sent to the various nodes involved in the construction of the index. Each node will use this hierarchy to assign its subset of the data collection to clusters. Results will be consistent across nodes as the hierarchy is identically replicated everywhere.

Map tasks do the assignments. Each mapper loads the representative hierarchy and clusters the data by reading-assigning-emitting every descriptor in it's block of data. The key emitted is the identifier of the cluster the descriptor is assigned to.

Reduce tasks receive records grouped and sorted on their cluster identifier from the shuffle. All reducers do is to propagate to disk the data they receive to form the bottom level of the index, i.e., the clusters themselves. Note some bookeeping is needed to keep track of cardinalities, etc.

4.2 Search with Map-Reduce

The Map-Reduce abstraction is geared towards efficiently streaming multi-megabyte blocks of data to map tasks. Such behaviour is quite opposite to the one of an indexing system that rapidly returns the points from the collection that are the most similar to the query point. In this case, a very small percentage of the indexed data is read from disks, one or few clusters at most. This, in turn, optimizes the response times of each query. Other applications need throughput, where sacrificing individual query response time is acceptable so that multiple queries can be run simultaneously. Copyright infringement applications typically need throughput.

Being a batch processing framework, Map-Reduce is not designed for answering individual queries, but is well suited for processing massive batches of queries. A batch is typically in the range of 10^4–10^7 query descriptors extracted from the associated images. It is easy to reorder query descriptors in a batch according to the cluster identifiers into which each query points falls, clusters being subsequently used for searching for the k-nearest neighbors. Such scheme minimizes the disk I/Os: one cluster fetched in memory is read only once and used to answer all the query descriptors in the batch that match with this cluster.[1]

All mappers receive the whole query batch with query descriptors ordered by cluster identifiers, and start to process their blocks of data. All blocks are read, but their contents is processed only if points they contain belong to the clusters needed by at least one query descriptor from the batch. Batching consumes RAM at each mapper since they have to maintain several k-nn tables for all the query points concerned with the current cluster under analysis. Tables can be deallocated when map tasks cross cluster boundaries, and a series of records are emitted. Special care must be taken when a single cluster spans more than one data block.

Overall, the cost for processing a batch is either entirely dominated by the cost of reading all data blocks, or dominated by the CPU for distance computations if the batch is really large or if there are very many points in each cluster.

[1]Hadoop *reads only once all the data blocks*, so grouping query descriptors per cluster is compulsory in this work.

5. CONTEXT OF THE EXPERIMENTS

The experiments were carried out on the Grid'5000 [14] testbed. The Grid'5000 project is a widely-distributed infrastructure devoted to providing an experimental platform for the research community. The platform is spread over ten geographical sites located through the French territory and one in Luxembourg. We could get access to the machines belonging to the Rennes site only.

5.1 Datasets, Queries and Ground-Truth

The dataset we used in our experiments has been created for the Quaero project.[2] One of the Quaero partners, Exalead, collected roughly 100 million images by harvesting the Web. To limit the size of data and to facilitate sharing among the partners in the Quaero project, images have been resized to only 150pixels on their largest side. SIFT descriptors were then extracted from these images, resulting in about 30 billion descriptors, i.e. 300 SIFT descriptors per image on average. To best of our knowledge, this image collection is one of the largest collections encountered in the content-based retrieval literature.

To evaluate the quality of indexing, we used that data collection as a distracting dataset into which we have drown the well studied INRIA Copydays evaluation set [10]. We resize Copydays images to the same size as our distractors and then use a copyright violation detection scenario, where we include 127 original images in our indexed database and use the associated 3055 generated variants (crop+scale, scale change+jpeg compression and manually generated strong distortions such as print-crumple-scan) as the queries. We then simply count how frequently the original images are returned as the top result. Many query images are visually such that only a very small number of SIFT descriptors can be extracted from their contents, e.g., 1% of the images have less than 8 descriptors. Finding the original images from their modified versions is therefore sometimes very challenging. Getting 100% accuracy is impossible as some image variants have zero SIFT descriptors (too dark e.g.).

5.2 Implementation Details

Several implementation details must be clarified to know how eCP works with Map-Reduce and Hadoop.

Preparing the dataset. The descriptors extracted from the image set are stored as binary files comprising records of 132 bytes; each record defines a descriptor and consists of a 4-byte integer for the image identifier followed by the 128 bytes of the actual descriptor. Overall, the 30 billion descriptors occupy just below 4 TeraBytes on disks. We first implemented a conversion mechanism creating *SequenceFiles* from binary data. A SequenceFile is a Hadoop-specific data

[2]Quaero is a research and innovation program adressing automatic processing of multimedia and multilingual content.

file employed for dealing with binary data. It consists of a header and one or multiple records. The header contains metadata that HDFS uses to parse the records. The records in a SequenceFile are fixed-sized and are defined as a single key-value pair. Several features (such as support for block compression and *sync markers* allowing to seek to the boundary of a record) make SequenceFiles an optimal choice for processing binary data with Hadoop. The descriptor conversion to SequenceFiles in HDFS creates records with the image identifier as the key, and the descriptor as the value.

Building the tree of representatives. We developed a Java implementation of the creation of the index tree containing cluster representatives. This tree of representatives is built outside Hadoop and serialized to a file subsequently used for clustering the data collection.

Creating the index: clustering. The clustering process assigning points to cluster representatives uses the index tree to efficiently discover the cluster each point belongs to. The Hadoop application consists of a map function that loads the tree of representatives and then reads the block of data it has to process. Each point from this block traverses the index tree until its closest cluster representative is known. Then the map task emits a *(cluster-id, point)* and loops. The reduce function simply outputs in SequenceFiles the records received from the mappers. It is key to realize the index tree is loaded by each map task at startup time. The tree will thus be loaded as many times as there are map tasks needed to complete an entire Hadoop job execution.

Searching batches of queries. A batch contains a very large number of query descriptors. Before being used to search the index, the query descriptors are reordered according to the identifier of the cluster each query points falls into. To do this, each query descriptor traverse the tree of representatives until it hits the bottom level, at which point the cluster identifier is known. To keep track of these identifiers for every query descriptor we build a lookup table. This table is created outside Hadoop and sent to every map task when a batch search is fired.

When spawned, mappers start by loading the lookup table.[3] A mapper then receives its block of data. It then finds in its block the records having any of the cluster identifiers existing in the lookup table. Only those records are subsequently used for distance calculation. It is possible that not all records of a block are used because (i) it is unlikely all query points of one batch will fall into distinct clusters, (ii) there are typically much more clusters than query points in a batch, (iii) a block typically contains several clusters. Mappers emit k-nn results.

6. PERFORMANCE RESULTS

After having presented the experimental setup, this section gives the performance results for running the index creation process on Hadoop. We then move to the performance of the batch search.

6.1 Experimental setup

We could have access to 129 nodes belonging to our local grid infrastructure. The nodes form three clusters, each composed of identical machines, as it is reported in Table 1.

[3]When a batch is very large, then this table consumes a lot of memory. Partial loading of the table is work in progress.

Cluster id	#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 1: Cluster Configurations.

%age	#Imgs.	#Desc.	Data Size	C	L	Index Size
10%	10M	3.3×10^9	0.5TB	652K	4	193MB
20%	20M	7.8×10^9	1.0TB	1.5M	4	461MB
100%	100M	30.2×10^9	4TB	6M	5	1.8GB

Table 2: Index Configurations.

While each cluster has a highly connective internal network, inter-cluster bandwidth is limited. In practice, some of 129 nodes may be down at any given point of time. The Hadoop framework was deployed as follows: the namenode, jobtracker and the job client are each on a dedicated machine, while the other nodes serve as both datanodes and tasktrackers.

At the level of HDFS, we use the default replication factor of 3 for the input data. In addition to facilitating the tolerance of faults, data replication favors local execution of mappers and minimizes the number of remote map executions, this being key for performance. We however typically set the output replication factor to 1 only. A larger value adds a substantial overhead to the running time because one replica goes to a remote rack. That cost becomes significant given the size of our data set.

To facilitate the experiments as well as to get a better understanding of the scalability issues, we indexed images subsets containing roughly 10% and 20% of the entire collection in addition to indexing the full 100M images. Details on the resulting configurations are reported in the Table 2. In all cases, $a = 3$ when soft-assigning the representatives in the index tree; this does not apply to data in clusters, see Section 3.1. The value of ts is also the same when indexing each (sub-)set, it is set $ts = 5,000$. This gives clusters containing 5,000 points, occupying 645KB, on average. On the one hand this creates quite a lot of clusters, on the other hand, each is quick to analyze at searching time. ts and the number of descriptors in each set to index give C, and L is set such that the cost of traversing the tree of representatives stays roughly the same across the configurations.

6.2 Index Creation

This section reports the performance results for running the index creation process on Hadoop, while increasing the data set from 10M to 100M images. For these experiments, the chunk size at the level of HDFS was set to 128 MB, as recommended by Hadoop, when dealing with large data.

6.2.1 Exp. #1: 10M images, 3.3B Descs., 0.5TB

The first experiment indexes 10% of the image collection, i.e., about 10 million images, 3.3 billion descriptors, 0.5 TeraBytes of data. This corresponds to the first line of Table 2. With this setting, 3,478 map tasks are to run—this is determined by Hadoop from the number of data blocks needed to store the raw descriptor collection. For this experiment, we configured Hadoop such that each node run simultaneously up to 8 mappers and 2 reducers. We used 20 to 50 processing nodes belonging to the Cl_1 cluster.

For all execution rounds, we checked the logs created during runs and observed that most of the map tasks were executed locally (only 20-40 out of 3,478 map tasks read remote blocks of data); this is a consequence of having set the replication factor to 3 for the input data, enabling Hadoop to favor most of the time local task execution. Replication is important for performance. We observed a 10% increase of the response time when setting the replication factor to 1 for the input data.

We now turn to the time it takes to complete the creation of the index. The measurements are reported in Table 3. The second column of this table gives the average time it takes to create the index when varying the number of nodes. Of course, the more nodes, the faster each terminates. The third column shows the total work where the times for all nodes are summed up. Several comments are in order. First, having no increase is a sign of having a global system that scales. Second, it is rather surprising to observe the work decreases as the number of nodes increases. This is a direct consequence of some of the Hadoop architectural design decisions colliding with the specific characteristics of our application: (i) the total number of map tasks to run is determined from the data collection size divided by the size of a block (128MB here, resulting in running 3,478 map tasks); (ii) the total number of map tasks is totally independent from the number of nodes used to run the entire job; (iii) a new map task is spawned every time a new block of data is to process; (iv) at spawning time, a map task has to load whatever auxiliary information it needs to correctly process the data in its block (in our case, the tree of representatives, 193MB to load every 128MB of data to index!).

It thus results that every map task has to load the tree of representatives. Spawning a mapper thus includes a fixed overhead for reading the tree of representatives. This tree is loaded again and again, even by mappers running on the same node. Overall, a large fraction of the differences in running time can be explained by considering the number of loads of this tree that can happen in parallel. With 50 nodes, more parallelism is possible, thus the overhead is less prevalent and the work diminishes compared to 20 nodes. It is also likely this data gets better cached when used that frequently with 50 nodes.

Lesson #1. Performance are hurt when two conditions are met: (i) the data collection occupies *many* blocks, hence many map tasks have to be run, and (ii) each map task need to load *a lot of auxiliary information* at startup time. It is key to reduce as much as possible the overhead payed by each map task at spawning time. One possible option is to increase the size of the blocks of data to a value that is significantly larger than the ones recommended by Hadoop, typically 64MB or 128MB. Setting this to 512MB or few GB in turn reduces the number of map tasks to spawn and thus reduces in proportion the time wasted when each map task starts. Note, however, that big data blocks may cause some nodes to run out of disk space as the temp area buffering the data produced by mappers and consumed by reducers fills up faster when blocks are big. We kept using 128MB blocks for this reason. It is also useful to compress as much as possible that auxiliary information to reduce its load time and to generously replicate it across the system to avoid disks/network hot-spots.

#Nodes	Time(min)	Work(min)
20	149.3	2,986
30	95.7	2,871
40	61.8	2,472
50	45.2	2,260

Table 3: Indexing 10%, varying number of nodes.

6.2.2 Exp. #2: 20M images, 7.8B Descs., 1.0TB

The second experiment indexes 20% of the full set, that is 1TB of data, about 20 million images and 7.8 billion descriptors. This gives 8,178 map tasks to run. We extended the deployment setup and used 57 nodes from Cl_1, 15 from Cl_2 and 36 from Cl_3, for a total of 108 nodes. Here again, 3 nodes are dedicated to managing the system, leaving 105 tasktrackers nodes. The system is again set to using, per machine, at most 8 slots for mapping and 2 for reducing.

Aside the obvious differences in the hardware and the size of the data set used here, we must highlight a key difference this experiment has with respect to the previous one indexing 10% of the dataset. Here, the tree of representatives used to guide and do the assignment of points is much larger. It uses about 1.5M representatives. Not only this occupies a lot more RAM (461MB), but it takes longer for each mapper to load from disks that tree in memory and to create the data structure for subsequent assignments. It also means more distance calculations are needed to assign a descriptor as there are more representatives eventually guiding to a larger number of clusters. For these reasons, the overall work for clustering this dataset is significantly larger than it is in the case of the previous experiment.

Here, with 108 nodes, it takes 71 minutes to complete the indexing and the total work amounts to 7,455 minutes. This increased amount of work is also in part caused by the uneven distribution of the representatives in the tree, from one level to the other. Therefore, a large fraction of the data traverses rather dense branches of the tree of representatives, which, in turn, requires to do more distance calculations to find the closest representative guiding to the next lower level.

6.2.3 Exp. #3: 100M images, 30.2B Descs., 4TB

The third experiment indexes the full dataset using 108 nodes. With this configuration, the tree of representative is large as it uses more than 6 million data points to accommodate with the 30.2 billion descriptors to cluster. The tree occupies roughly 1.8GB in RAM.[4] This forced us to reduce the number of map tasks per machine to 4 only as otherwise not enough RAM was available for each mapper.

With this setting, it took about 10 hours to cluster the entire data set. A careful analysis of the logs shows that 99% of the reduce tasks where completed after 520 minutes, and the remaining 1% reduce tasks completed after 80 additional minutes. The reason behind this behavior is in part the uneven distribution of points to clusters.

But there is another explanation to this response time. Finely analyzing the data collection, we discovered that it contains hundred thousands of *identical* distracting images that turn out to come from a small set of explicit web sites having different URLs redirecting to a unique point. This

[4]Note 1.8GB of auxiliary info have to be loaded every 128MB of data to cluster! This encourages using significantly larger block sizes, see Lesson #1 above.

is unfortunate, but it is a good example of what happens in the real world when indexing images. It would have been possible to filter these images but this would have required a specific ad hoc process we will integrate in the future. The direct impact of so much duplicates is that there is a small set of clusters into which the descriptors of these images accumulate, creating very large, unbreakable clusters, and writing them to disks takes a lot of time.

Lesson #2. Hadoop's map tasks are completely independent and each require to load the tree of representatives. When this auxiliary information is large, then each map task consumes a significant portion of the RAM available on a node. In turn, it means map tasks are unable to run inside every available core in a node, because there is not enough RAM. It is unfortunate to waste some of the processing power leaving cores idle because there is no way to share data, even *read-only data* (as is the tree of representatives) between map tasks running on the same node. This observation suggests for application programmers to implement multi-threaded map tasks. This is way more complicated to program but it is one option for using all the processing power of nodes while circumventing Hadoop's inflexible architecture. With multi-threaded map tasks, a single task would load the auxiliary data only once and then would process its block of data faster thanks to its multiple threads running on multiple cores. In the case of this experiment with the full data set, one single map task could then use up to 6 threads processing data in parallel on Cl_3, overall keeping the 24 cores constantly busy, instead of using only 4 cores now.

6.3 Batch searching

This section reports the performance results obtained when searching the full data collection with batches of query images. The images in the batch are the 3,055 variants from the Copydays evaluation set. The results are expressed both in terms of response time and search quality. Response time wise, we record the time it takes to complete the query batch using the 110 nodes, almost as in Exp. #2 and #3. Quality wise, we search for the 20 nearest neighbors of each query point computed from the query images. There are just below 1M query points in the batch. Each nearest neighbor votes for the image from the indexed collection it belongs to, and the votes are aggregated to eventually return the identifiers of the most similar images. We have a rather strict success criterion for searching: the search succeeds if and only if the original image identified from its query quasi copy has rank 1; the search fails otherwise. The percentages given when discussing quality thus correspond to counting the number of times original images are ranked first.

The lookup table built from the descriptors in a batch (see Section 5.2) is stored as an HDFS file read by all search mappers when they are spawned. It takes about 3 minutes to build this lookup table on a single core outside Hadoop. The lookup table file is replicated three times to reduce contention when mappers access it. The block size for the indexed data is 128MB, with hence 33,483 mappers to run. To avoid remote reads, we replicated the indexed data in HDFS using a replication factor set to two.

6.3.1 *Exp. #4: Time and Quality, 100M images*

Searching the entire batch took 1,623 sec. on average, or just over 27 minutes. This gives an average processing time per image of under 530ms. 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. It also plot the average percentage across all variants at the far right end of the Figure. Note for comparison we also measured the quality when indexing 20% of the data collection (see Exp.#2). From the Figure, it is clear that eCP returns high quality results, except for some severely attacked images such as when 80% of the image is cropped and then it is rescaled to its original size, or when strong manual variants are applied. Note, we count as search failures the cases when no descriptor can be computed on the query images (this happens for 6 variants). 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 eCP is a very viable indexing technique.

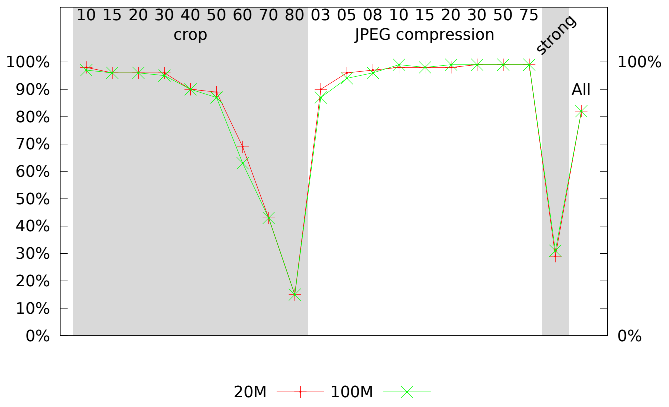

Figure 1: Search Quality, Copydays evaluation set

6.3.2 *Discussion*

Aside the experiment described above, we have done many other performance evaluations varying several parameters. We briefly report the key results we found.

Lesson #3. Most of the mappers read the data locally, only about 1% of remote reads were observed. This is again the case for setting the replication factor to a value above 1. We have not seen any major performance improvement for any value > 2. In contrast, maintaining a single copy of the indexed data causes roughly 8 to 10% remote reads, hurting performance. Rack awareness and replication are good for performance, and not only for coping with failures. Note, we experienced several nodes failures and happily observed that Hadoop re-ran tasks, eventually completing the runs.

Lesson #4. We have re-blocked the indexed data before running the search to see the impact of using larger blocks, as suggested earlier. When setting the block size to 512MB, we observed a dramatic response time improvement when using batches containing roughly 12,000 images (roughly 3.7M descriptors). Times decrease down to roughly 1,500 seconds (instead of 3,500 when blocks are 128MB), simply because there are only 8,808 mappers loading the lookup table (its size is roughly 1GB) instead of more than 33,000. This is really making a case for using larger blocks.

Lesson #5. Hadoop is architectured such that all the blocks of data are read. Therefore, search runs are totally dominated by I/Os until the batch has enough points to keep the CPU extremely busy doing distance calculations. Using the 20% index configuration and 512MB blocks, we have ran batches containing only one query image (312 query descriptors), and this takes 323 seconds while the Copydays batch (1M query descriptors)runs in 388 seconds. The difference is quite small (15%), especially because it includes an I/O bounded step (loading the batch) in addition to the extra CPU work, likely almost entirely hidden by the I/Os.

Observation #6. We also used this index configuration to check the impact of using a varying number of nodes on the time it takes to run the Copydays batch. With 30 nodes, it takes about 1,048 seconds, 744 seconds for 40 nodes and 388 seconds with the 100 nodes. The total work is stable when using 30 and 40 nodes (respectively 31,440s and 29,760s) but jumps to 38,800s when using all the nodes. Part of the extra cost comes from the fix time it takes to launch Hadoop, it is always about 20 to 40 seconds, which becomes a significant part of a running time that is around 400 seconds.

7. CONCLUSIONS

This paper presents a Map-Reduced based implementation of an high-dimensional indexing algorithm that uses clustering to build small groups of data, subsequently searched. The performance of this algorithm have been demonstrated in terms of speed gains when using more hardware. It has also been demonstrated in terms of quality as we show it correctly identifies about 82% of the images from a state-of-the-art evaluation set drown in 100M distracting images. Several lessons can be drawn from this work such as the case for using data blocks of a size larger than the one Hadoop is recommending, or the case for implementing multi-threaded map tasks to fully use the processing power of cores while avoiding RAM issues.

Overall, Hadoop is helpful for achieving scalability. Properly setting its parameters is not trivial, however, as described in the paper. We observed that very often, running an experiment that creates and/or searches an index is not what consumes the largest amount of time. It is rather copying the data to HDFS before being ready to launch experiments, or getting out the indexed data from the grid for being used elsewhere. Feeding a grid/cloud with all the required data through limited bandwidth is a very practical problem.

This work was partly achieved as part of the Quaero Project, funded by OSEO, French State agency for innovation.

8. REFERENCES

[1] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubský, and P. Zezula. Building a web-scale image similarity search system. *Multimedia Tools Appl.*, 47(3), 2010.

[2] F. Chierichetti, A. Panconesi, P. Raghavan, M. Sozio, A. Tiberi, and E. Upfal. Finding near neighbors through cluster pruning. In *PODS*, 2007.

[3] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, 2004.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[5] M. Douze, H. Jégou, H. Singh, L. Amsaleg, and C. Schmid. Evaluation of gist descriptors for web-scale image search. In *CIVR*, 2009.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

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

[8] G. Gudmundsson, B. T. Jónsson, and L. Amsaleg. A large-scale performance study of cluster-based high-dimensional indexing. In *VLS-MCMR Workshop with ACM MM*, 2010.

[9] J. S. Hare, S. Samangooei, D. P. Dupplaw, and P. H. Lewis. Imageterrier: an extensible platform for scalable high-performance image retrieval. In *ICMR*, 2012.

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

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

[12] H. Jégou, F. Perronnin, M. Douze, J. Sánchez, P. Pérez, and C. Schmid. Aggregating local image descriptors into compact codes. *IEEE Trans. on PAMI*, 2011.

[13] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011.

[14] Y. Jégou, S. Lantéri, J. Leduc, and all. Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. *Intl. Journal of HPC Applications*, 20(4), 2006.

[15] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *MM*, 2008.

[16] H. Lejsek, F. H. Amundsson, B. T. Jónsson, and L. Amsaleg. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Trans. on PAMI*, 2009.

[17] H. Lejsek, B. T. Jónsson, and L. Amsaleg. NV-Tree: nearest neighbors at the billion scale. In *ICMR*, 2011.

[18] D. Lowe. Distinctive image features from scale invariant keypoints. *IJCV*, 60(2), 2004.

[19] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[20] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 2010.

[21] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In *CVPR*, 2008.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[23] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.