Parallelism and distribution for very large scale content-based image retrieval

Gylfi Thor Gudmunsson

N° d'ordre : 00000 ANNÉE 2013

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Gylfi Thor GUDMUNDSSON

préparée à l'unité de recherche IRISA – UMR6074

Parallelism and Distribution for Very Large Scale Content Based Image Retrieval

Thèse soutenue à Rennes le 12/09/2013

devant le jury composé de :

David GROSS-AMBLARD
Professeur at Rennes 1 University / *Président*

Philippe-Henri GOSSELIN
Professeur à l'ENSEA et membre du laboratoire ETIS / *Rapporteur*

Patrick VALDURIEZ
Directeur de recherche, INRIA Sophia Antipolis / *Rapporteur*

Bjorn Thor JONSSON
Maitre de Conférence à l'université de Reykjavik / *Examinateur*

Julien LAW-TO
Chef de projet, Exalead / *Examinateur*

Christine MORIN
Directeur de recherche, INRIA Rennes Bretagne Atlantique / *Examinatrice*

Laurent AMSALEG
Chargé de recherche, CNRS, IRISA, Rennes / *Directeur de thèse*

A picture is worth more than a thousand (key)words

Acknowledgments

I would like to thank Professor Philippe-Henri Gosselin and Dr. Patrick Valduriez for accepting the role of reviewers.

I would also like to thank Professor David Gross-Amblard, Dr. Björn Þór Jónsson, Dr. Julien Law-To and Dr. Christine Morin for accepting the role of examiners.

Last on this jury, but by no means least, I would like to thank Dr. Laurent Amsaleg for his patience and perseverance during the past years. His guidance and supervision has been invaluable.

Not on the jury, but still well worthy of thanking, is my other supervisor, Dr. Patrick Gros.

I am also very grateful for the financial support of the Quaero Project and would like to extend my gratitude to the partners involved. Especially Exalead, for providing us with the set of 100 million images, and Karen Sauvagnat for doing the external evaluation.

Another important factor in making this work possible is the Grid'5000, the PIM platform and the people that keep the hardware running. I would therefore like to thank all the staff involved, but especially I thank Sebastien Campion and David Margery for their assistance and hard work.

My co-workers also deserve to be mentioned. I am especially grateful to my long-time office mates, (soon to be Dr.) Julien Fayolle and Dr. Bogdan Ludusan, for their help, feedback and most of all their company.

I would like to thank my close co-workers, Dr. Diana Moise and Dr. Denis Shestakov for their contributions on the development of distributed eCP.

And I am also grateful to all the other members of the TexMex team and I would like to thank them all for our time together.

Lastly, I would like to thank my family for their support over the years.

Contents

Introduction

The popularity of digital cameras has created an extraordinary growth in the quantity of digital images produced and made publicly available. With the availability of large image collections comes also the desire to search such collections in order to discover similar images. The easy solution to finding similar images has been to use the established text-based search, where keywords or tags have been manually associated with each image, describing its contents. It is the additional text-based content that is then actually searched. The problems with keywords are numerous and take various forms. There are practical problems, like how laborious it is to manually provide tags and how difficult it is to quantify the relevance of a tag. And there are semantical issues, like language ambiguity or synonymy and the lack of consistent reproducibility due to human nature.

Content Based Image Retrieval Systems

One alternative to the text-based approach is to automatically extract features from the image that somehow capture its visual content. In addition, a similarity metric, that can quantify the similarity of two features, must be provided. The captured image features are then cataloged and indexed to form a searchable database. The queries for such a database then take the form of an image. This kind of searching is called *content-based image retrieval* or CBIR.

To achieve its goal of efficient and responsive search, the CBIR system has to solve two problems: 1) how can it automatically capture the content of an image? and 2) how does it catalog, index and store the captured content such that it can then be efficiently searched?

Many methods have been proposed to capture the content of an image. Essentially, all of them try to describe the visual content by extracting features and calculating from them one or more descriptors that typically take the form of high-dimensional vectors. The similarity of two images is then established by calculating the distance between the extracted vectors. This kind of feature extraction does not solve all the aforementioned image description problems (such descriptions have typically little semantic value). It does however provide a reliably reproducible and automatic way of describing images based on the image

content, free from the whims of the particular user.

The ability to measure similarity also means that it is possible to discover that image from a collection which is the most similar to a query image, ergo searching is made possible. The simplest most basic search algorithm would be a brute-force sequential scan, calculating the similarity of each descriptor in the collection, and keeping track of the most similar candidate descriptor(s) discovered so far.

To reduce the overhead of searching, most algorithms use some form of a divide and conquer strategy. The data is partitioned into small groups and an index can be used to quickly discard unnecessary groupings during search. Regardless of the partitioning strategy or the indexing method, the construction of the indexed database is a time consuming task that requires the processing of each descriptor, at least once. For this reason, the construction is typically done off-line.

On the other hand, the search is typically an on-line process where responsiveness is an important factor. The two most common search paradigms are the k nearest neighbor search or k-nn, where a fixed size list of the k most similar candidates found are kept, and range-queries, where all the candidates that have a similarity within a given range r are kept.

The search in the traditional CBIR systems would work as follows: It would start by identifying the query's most similar partition and then scan that partition for the query's nearest neighbors. Then, using the distance to the k^{th} nearest neighbor (or the given range r), the index would minimize the work by using mathematical analysis to exclude any partitions, that could not possibly hold any more proximate descriptors, from being scanned. Any partitions that could not be excluded in this way would have to be scanned to see if nearer neighbors could be found.

With the term "traditional CBIR systems" we refer to the methods proposed before the year 2000. At the time, the scale of the collections and the dimensionality of the descriptors were much lower than today. Image collections would typically number in the tens to hundreds of thousands of images and the dimensionality of descriptors numbered in the low tens.

Challenges and solutions for the modern CBIR system

Currently the image collections number in hundreds of thousands to billions of images. As the image collections became larger, the need for better discrimination and more powerful descriptors became apparent. To address this need, the number of features extracted and the descriptiveness of those features have increased. Both pose serious problems for traditional CBIR systems. On the one hand, the CBIR system has to scale-up to ever larger descriptor collections and on the other, the more powerful descriptors are typically represented using higher-dimensional vectors.

The traditional CBIR systems promised to find the most similar descriptor(s), or the actual nearest neighbor(s), in the database. To do so, they use the index to only exclude those parts of the partitioning that can *under no circumstances* contain a more similar descriptor candidate. As the dimensionality grows, the ability of the index to safely discard parts of the database is diminished and the search algorithms may degrade to a full scan of the entire collection. As a result, the traditional algorithms have failed to meet the modern demand to efficiently handle hundreds of millions or even billions of images.

To address this need, modern CBIR systems have turned to approximate approaches. The approximations are based on the observation that frequently the most similar descriptor(s) are found early in the search, scanning the first partition(s). In traditional systems, most of the effort is thus spent on scanning partitions that can not be excluded by the index, but rarely hold descriptors that change the result. By limiting the search to the most likely partition(s), the search cost can be kept down while still finding most of the true nearest neighbors. Thus, the approximate solutions strike a balance between response time and quality.

Relying on approximations has not been the only way to create larger and faster CBIR systems. Hardware has always played a part in the development of such systems. This is because more often than not, the design of the algorithm is trying to overcome some bottleneck that originates in the limitations of hardware. In the last decade or so this has become even more relevant as there have been many advances in hardware that greatly influence the design of current and future CBIR algorithms. Advances such as: solid state disks; size of the main memory now possible; multi-core processors and last but not least the growing availability of a large number of high-caliber machines in the form of grids and clouds. All are important factors to consider for the development of tomorrow's CBIR systems.

Contributions

The work presented in this thesis addresses the issues of web-scale CBIR. At web-scale, the image collections number in hundreds of millions or billions of images and there are demands for both responsive real-time search and high-throughput search.

A key assumption that we make is that our collections are so large that they will have to reside on secondary storage. As we shall see, a secondary storage device can be a severe bottleneck. It is thus essential to design a CBIR system with the limitations of disks in mind, right from the start.

Our algorithm, extended ClusterPruning (eCP) [GJA10, GAJ12b, MSGA13, GAJ12a, SMGA13], is based on extending a method called ClusterPruning (CP). The first contribution is a set of extensions to deal with indexing and searching

very large image collections. They include: reducing indexing cost, balancing the size distribution of clusters and efficiently accessing disks. We extensively evaluate eCP using a variety of image collections, commonly available off-the-shelf hardware and various secondary storage devices that include Solid State Disks (SSD).

The second contribution includes parallelizing eCP to harness the power of multi-core processors, both for indexing and searching. In addition, we propose a high-throughput batching search that trades response time of individual queries for throughput. The throughput is achieved by searching the database for images that are similar to hundreds or even thousands of query images at the same time. Again, we extensively evaluate our algorithm. This time using a powerful 12-core (24 logical) server with 144GB main memory and using high-performance RAID-6 configured magnetic disks. We also drastically increase the size of the image collection used for evaluation, going from a few hundred thousand images to over 25 million images.

Finally, as a third contribution, we extend the eCP algorithm again, to harness the capability of distributed computation. We do this by adapting eCP to the MapReduce programming model and implementing it on the Hadoop framework. This time, we increase the size of our image collection to 100 million images, where over 30 billion SIFT descriptors are extracted that in turn results in 4TB of data. Experiments and evaluation are run on over 100 grid-machines from three different machine-clusters.

Overall, we show: that database construction can smoothly scale-out to a vast number of cores and machines; that our simple eCP algorithm can scale-up to provide high quality results using very large collections; that high-throughput search is possible if queries can be batched and that responsive single query search would be possible if we had high-performance SSD devices with large storage capacity.

Structure

The rest of this thesis is structured as follows. We start with chapter 1, where we discuss relevant work and algorithms which we build upon, have inspired us and motivated our contributions. In chapter 2 we develop eCP and present our extensive evaluation of the algorithm. Chapter 3 contains the presentation and evaluation of the parallelized eCP as well as the batch oriented search. In chapter 4 we show how we adapt eCP to the MapReduce programming model and the Hadoop framework as well as presenting evaluation of both distributed database construction and search. Finally, we end the thesis by summarizing our accomplishments, draw conclusions from them and discuss future work.

Chapter 1

Background

In this section we will discuss the challenges of large-scale modern CBIR systems in more detail. We will explain the ideas and assumptions of prior work that we build upon, as well as highlight the advantages and shortcomings of alternative solutions.

We will start by discussing how we evaluate the performance of CBIR systems. The key metrics are presented and we explain how we build a ground truth that we use to measure search quality. This knowledge is key for fully understanding the large body of experiments presented in this manuscript.

We then move on to briefly explain how the visual content is captured automatically and described with high-dimensional feature vectors. We explain how the SIFT descriptors work, the de facto standard in image and object recognition, and the descriptors we use in our work.

In the third section, we take a general view on matters related to our topic and learn important lessons about the design of tomorrow's CBIR systems. We base our lessons not only on the CBIR literature, but also on the state of the internet-addicted world we live in. Among the issues discussed are the size of today's web-scale image collections, a size that was unimaginable just a few years back, and the shift in development of computing power, from scale-up (faster and faster) to scale-out (more and more).

The fourth and last section is devoted to describing in some detail specific algorithms that are important to our discussion and the development of our own solutions to the large-scale CBIR problem.

As may be evident from reading the introduction, our goal is to develop a modern CBIR algorithm that will take full advantage of today's state of the art hardware. That includes developing both multi-threaded and a distributed version of eCP. As each implementation is quite intertwined with the relevant background material, we prefer to defer the discussion of specific background material for each topic until the beginning of each chapter. The adaptation of

eCP for parallelism is covered in Chapter 3 and eCP is extended for distributed environments in Chapter 4.

1.1 Evaluation of CBIR systems

The evaluation of a CBIR system has to take many aspects into account. First is the efficiency, like response time throughput and hardware utilization.

Second is the evaluation of the search quality. This is naturally done at the image level, it is after all similar images that we are looking for. We will present some of the commonly used indicators to gage quality.

We then move on to detailing how we build an image level ground truth for a query set, as one of the key components of evaluating search quality is to know what the correct answer to a given image query should be.

1.1.1 Metrics

To gage efficiency, we use various measurements as well as monitoring both hardware requirements and utilization. Following are some common metrics used for this purpose:

- **Time** is an important metric. The *real time* is the wall-clock time that passes between the start and end of a task. We also measure *user time*, the cumulated time that cores are reported as busy, and I/O time, where we measure time spent doing and/or waiting for I/O operations.
- **Cache miss** measures how frequently data has to be accessed at lower and more costly levels of memory hierarchy.
- **Memory footprint** is the maximum memory requirement that the algorithm requires during its execution.
- **Storage footprint** is the maximum disk space requirement that the algorithm requires during its execution.

We now move on to search quality measures. Let us assume an image query q that is searched on a CBIR system that has indexed a image collection C. The result of the search R_q is a ranked list of the most similar database images. To measure the quality of R_q, we need to know the correct ground truth ranked list G_q that should have been returned. If we know G_q, we can discover the *true positives* ($R_q \cap G_q$) and the *false positives* ($R_q - G_q$). Identifying the *false negatives* ($G_q - R_q$) is key in assessing quality. Also, as our result is a ranked list, the order of the results can also be used as a quality metric.

Following are common quality metrics derived from the above information:

- **Precision**, is the fraction of *true positives* in the result $(R_q \cap G_q)/R_q$. Precision at x, or P@x, limits the result set R_q to the x top ranked results

only. For many use-case-scenarios, the P@1 is the most important measure, i.e. "was the correct image returned as the top ranked result?".
- **Recall**, is the fraction of the ground truth returned $(R_q \cap G_q)/G_q$. Like precision, recall can also be limited to the top x values or R@x.
- **mAp**, or mean average precision, is commonly used to produce a single evaluation metric over a large set of queries. It simply calculates the average precision for each image query and finds the mean value.

The most common problem is obtaining the ranked ground truth G_q. In our example we said q was a query image, but we could just as well do this on the descriptor level. If q is a query descriptor, typically, the only way to obtain G_q is to do an exhaustive scan of collection C, comparing q to every single descriptor. In practice, the ground truth has to be obtained for a large set of queries and repeating the exhaustive scanning of C, for each q in the query set, is a daunting task.

How we obtain the ground truth at the image level is the topic of the next section.

1.1.2 Image level ground truth and search scenarios

A common way to create a ground truth is to randomly pick images from the collection and create altered versions of them (using cropping, rotation, blurring etc.), using a software tool such as StirMark [PAK98]. The StirMark tool is available for download at this web site [PAK97].

The outcome of such a process is two sets of images, the randomly picked originals and the set of altered images. Each image, both the originals and the created variants, is given a unique ID-number and the ground truth is based on a lookup-table that keeps track of what variant came from which original image.

Then, at search-time, one set of images is included in the database while the other is used as an image query set. There are thus two possible scenarios:

I) The altered images are used as queries and the originals are left in the image collection to become part of the indexed database. This is often referred to as a "the Copyright detection scenario", as it nicely simulates copyright detection. But this type of evaluation is applicable to many other problems as well.

In this case, the ground truth G_q for any given query image q is a single image, namely the original image that the query image variant was generated from. In the search result R_q for this scenario, we would thus expect the original image to be high on the rank of results, or else the search has failed. While recall is very important, what we are really interested in is to have the correct image at the top of the list (P@1).

II) The other possibility is to use the original images as queries and include

the altered set of images in the database. This scenario is often referred to as "similarity search" as it is a good way to simulate finding multiple similar images in a database, given a single query image.

In this case the ground truth G_q, for any given original image used as a query q, is a set of images, namely all the variants that were created from that particular original image. In the search results R_q for this scenario, we would expect several, if not all, of the variants to be among the top ranked results. The most important metric in this case would be the mean average precision (mAP) as both precision and recall matter.

How we get from results with ranked lists of vectors to ranked lists of images, is our next topic.

1.1.3 Vote aggregation, per-vector to per-image

In CBIR systems, images are represented by multiple descriptors extracted from each image. An image query will thus consist of multiple query descriptors, each generating a search result R_q that consists of a ranked vector-list, instead of a single ranked image-list. While the vector-level results do not directly measure image similarity, each vector of every vector-list belongs to some image in the database. It is thus possible to convert the ranked vector-level results into ranked image-level results. The conversion process is often called a vote aggregation, as it can be seen as an election process, where vectors cast votes for images and the popularity of images determine their rank in the image level-result.

There are many ways to hold "fair" elections and covering all of them is beyond the scope of this thesis. One of the simplest election systems is to base the ranking on the frequency of which each database image is represented by a descriptor in the ranked vector-lists. This election process uses a one-vote per vector policy, without taking any consideration neither to the rank nor the proximity information. Each vector, in each ranked result-list, casts one vote for the database image it comes from. Once all vectors have voted, all the votes are tallied and the list of candidate images is sorted, based on the number of votes each image received. The outcome is a ordered list of database images that becomes the ranked image-level result. In fact, this is the vote aggregation process used in eCP (with only a small added constraint that will be discussed later).

A more common voting policy, used in most state-of-the-art CBIR systems, is to take the vector's rank and proximity to the query into consideration, assigning weights to the votes being cast. Weighing the votes is done under the assumptions that: a) proximate vector(s) are more likely to be very relevant and b) vector(s) in the vector-list with the best relative proximity to the query are more likely to be correct. Thus, each vector's vote is weighed based on both factors, a) and b),

before it is cast. To fully understand the advantage of this, we need to understand a little bit better what actually happens in the election process.

Vote aggregation and the background noise

Let us assume we do a k-nn search and we are using the copyright detection scenario. For each query descriptor q, a k-nn ranked result-list is returned. At the image level, only one of the database images is in the ground truth. Thus, it should be clear that only 1 of the k votes from each ranked vector-list is correct. [1] The remaining $k-1$ votes are cast for random images that happen to have similar vectors. The number of votes a non-correct image receives by the random-votes is called the "noise-level". The larger the k is, the more noise is created. What the CBIR system relies on is that the correct image receive enough votes, such that it rises significantly above the "noise-level".

As the query image has been distorted, then, compared to the vectors from the original image, some new ones have been created and others are very different. Thus, out of all the extracted query descriptors, only a handful may truly be matching original descriptors. Some correct matches are very similar, at the top k, while others fall far down the list. Setting the size of k is thus a balancing act between "noise-level" and capturing the distorted but correct matches.

One of the primary goals of the weighted-voting policy is to allow the use of a large k, without increasing the "noise-level" too much. This is done using the a) and b) criteria describe before, to weigh down "bad" remote neighbors in k and boosting the "good" proximate ones. In those cases where all descriptors in the ranked vector-list k are equally remote, all descriptors get about the same voting weight.

The "noise-level" also depends on the number of unique images in the database. Many CBIR systems designed and configured for large scale may not work as expected when tested on too small image collections. The search may rely on a large number of unique images to keep down the "noise-level" and provide good results. By simply increasing the number of images, such that the collection is of adequate size, the quality of the search may improve drastically. When evaluating a CBIR system, it is important to make sure that the configuration is appropriate for the task at hand.

Our eCP search algorithm uses the simple one-vote-per-vector policy with one added restriction. eCp does not need to weigh it's voting as the search produces very accurate distance estimates (we keep the vectors and can thus do the full Euclidean distance calculation). Therefore we can set the size of the k nearest neighborhood very low, typically we use a $k = 20$. So far, we have not seen the

1. We assume that each vector in k comes from a unique image. Allowing multiple votes for the same image in the same ranked result-list can cause problems and should be prevented.

need for implementing a more CPU intensive voting scheme (like weighing the votes), although doing so would be an easy task.

The added restriction is that eCP does not allow multiple votes for the same image in the same ranked result-list R_q. We are looking for a one-to-one correspondence between descriptors, if two images are truly similar they will have many one-to-one matches and the vote will be high.

An image where all the descriptors extracted are similar to each other we call a "monotonic" image. A single such monotonic image can cause a lot of trouble if we allow multiple votes to be cast for the same database image from the same ranked result-list, i.e. we would be allowing a one-to-many correspondence between a query descriptor and image votes. The monotonic image in the database could have several of its similar descriptors accumulate many votes, allowing it to rise above the "noise-level", even if only a handful of the query descriptor (ranked result-lists) are actually behind the casting of those votes. Essentially, allowing the one-to-many correspondence can result in a tendency to always find monotonic images as good matches and make a CBIR system sensitive for attacks [DKAF12].

1.2 Capturing image content, SIFT descriptors

As was briefly described in the introduction, it is the captured content of the image that is actually being indexed and searched when using a CBIR system. That content takes the form of high-dimensional vectors that, together with a distance function, make it possible to quantify the similarity between images. That in turn provides the basis for the operations necessary for a CBIR system, i.e. the ability to compare images, sorting based on similarity and grouping similar images together.

There are many extraction algorithms and descriptor types in use today and new schemes are published each year. Most of the variety is due to methods that are aimed at solving specific sub-tasks, where domain specific information is used to create "better" (smaller, faster or more informative) descriptors.

The one most accepted and commonly used today is the **scale-invariant feature transform** or **SIFT** [Low04]. The SIFT descriptor was developed for object recognition in images and therefore the feature has to be tolerant of changes in the image like rotation, scale or illumination etc. Today, SIFT is used for a wide range of applications like localization and mapping in robotics, image stitching and content based search. The popularity of SIFT comes from its superior ability to reproduce similar descriptors from otherwise dissimilar images of the same visual motive.

SIFT extracts local descriptors in three steps. It starts by finding *keypoints*, at coordinate (x, y) and at a scale σ, where the Difference of Gaussian (DoG)

Figure 1.1: Example of SIFT descriptors. 1,751 descriptors are discovered in this 720x480 pixel image.

model: Ester Eir Gudmundsdottir, photographer: Rakel Edda Gudmundsdottir

response is a local extremum at various scales of the image. In the second step, for each *keypoint*, it's main orientation θ is computed based on the gradient directions locally around (x, y). We thus get a definition of a *keypoint* $= (x, y, \sigma, \theta)$. After filtering undesired *keypoints*, like low contrast and those that fall along edges, the third step is to compute the actual descriptor vector based on a support region centered on the (x, y) coordinate. The support region is divided into 16 subregions, and 8-bin quantized histograms of weighted gradient orientation of the subregions are concatenated into a 128-dimensional vector.

In Figure 1.1, we see an example of where and how SIFT features are acquired. On the left is the original 720x480 pixel image and on the right we see also the 1,751 local descriptors with orientation.

As each descriptor is calculated from a fixed neighborhood, the discovery of *keypoints* is limited to an internal area that is 20 pixels in from any image border. This restriction matters if the images are very small as only a small part of the image surface can actually be used for *keypoint* discovery. As a result, for some of images in the collections we use, where the longer edge of an image is limited to 150 pixels, few or even no descriptors can be extracted.

Although eCP can work with high-dimensional vectors from any source, we do all of our evaluations using SIFT features extracted from images. [2]

1.3 General lessons and observations

Before we discuss specific algorithms and CBIR systems, there are some general observations we would like to highlight from the literature and from the state of the online web-scale world we now live in.

2. eCP was adapted in less than one day to work with 36 dimensional audio features.

1.3.1 Size and growth of web-scale image collections

The size and availability of image collections has grown beyond anyone's expectations in the last few years. Image collections, that researchers had painstakingly been collecting over several years, where dwarfed in size almost overnight by the rise of social websites like Flickr and Facebook. Since Flickr started in 2004, it has grown to over 6 billion images and, in the year 2012, over 518 million publicly available photos were added to their collection, or 1.42 million on average per day. Facebook is even larger, with over 100 billion images and a daily growth rate of over 200 million. Unlike Flickr, the Facebook collection is not directly publicly available and thus academia has been prone to collect images from Flickr for research purposes.

The growth of video content, like on YouTube, has followed a similar trend. Recently, the mobile-oriented image hosting, such as Instagram, is also catching up quickly. In the near future, publicly available web-scale collections will not be limited to just a few sites and the demand for web-scale CBIR search will only become stronger.

The size and growth rate of web-scale collections is such that currently only the high capacity secondary storage devices are a viable option for a web-scale CBIR system. One of the primary reasons web-scale collections cannot be stored in main memory is because very large RAM is still very expensive and thus hardly affordable [GG97]. Furthermore, such collections need persistency, that can only be achieved today thanks to secondary storage devices. Making sure CBIR algorithms deal efficiently with disks is a crucial design issue.

1.3.2 Hardware developments that influence CBIR design

Since the shift in microprocessor development in 2005, from scaling-up with faster and faster cores to scaling-out to multiple cores, parallelized programming has become necessary to fully utilize the available processing power of a single machine. In addition, simultaneous multithreading, or SMT, has become very advanced and widely used. Today, SMT is typically implemented transparently, where the operating system sees two (or more) logical cores for every real core available. However, as we shall see in chapter 3, it is important to be aware of SMT and design parallelized algorithms accordingly.

Another scaling-out trend has been the growing availability of large machine clusters, or even multiple clusters, in the form of grids and clouds.

As we shall see in Chapter 2, even with a hierarchical index, an enormous amount of distance calculations is necessary to partition a large collection of descriptors. On a single core, the process can take days or even weeks. For CPU intensive applications like that, it is essential to use both parallelization and dis-

tributed computing to harness all the available processing power of the modern day infrastructures. However, it is typically not straight forward to efficiently parallelize and distribute complicated algorithms as an afterthought. It is therefore important to keep in mind the limitation of parallel and distributed computing during the CBIR's design process, adapting the solution to the environment and not the other way around.

For a long time, 32bit addressing limited the size of main memory to 4GB. Today, 64bit addressing has taken over and the machines with tens or even hundreds of GB of RAM are available. Machines with very large RAM capacity still remain very expensive. This development has brought new life to in-memory CBIR systems. But, as we have already discussed, we do not consider in-memory algorithms as a viable long term option at web-scale. The memory footprint and the efficient use of the memory is however very important to any CBIR system.

The last development we will mention here is recent advances in secondary storage, namely solid state disk technology (SSD). For decades, the magnetic disks have been the only option for permanent storage. The only drastic development has been in the form of increased storage capacity. Due to mechanical limitation of moving parts, performance of random access is very poor. A lot of design and software logic is thus spent on grouping I/O requests such that the disk(s) are accessed sequentially. This is a form of batching I/O requests. The new SSDs are both much faster and, as they have no moving mechanical parts, not limited to sequential access patterns. Currently, the only true limitation of SSDs is their still relatively small capacity. As we will show in chapter 2, I/O bound algorithms like the eCP search can greatly benefit from using SSDs.

1.3.3 The failure of the traditional CBIR algorithms

In this section we will discuss why the early CBIR algorithms have failed to handle the scale of image collections and the descriptive high-dimensional features used today.

Uri Shaft and Raghu Ramakrishnan published an article in 2006 where they showed that almost none of the indexing structures at that time could cope with increased vector dimensionality [SR06]. This was a serious problem, as the answer to the demand for better and more descriptive features had been found in increasing the dimensionality of the descriptor vectors. Making matters even worse was the size of the descriptor collections. Not only were there more images in the collections, but the use of multiple local descriptors to describe each image could easily hundredfold, or even thousandfold, the number of descriptors to index.

The millstone for the traditional algorithms, like the K-D tree [Ben75], R*-tree [Gut84], SS-tree [WJ96] and the M-Tree [CPZ97], was that they would guarantee to find "the" nearest neighbor(s). To provide that guarantee, the search

had to scan every partition that could possibly contain a nearer neighbor than the ones already found, regardless of how unlikely that extra work was to actually change the results.

Around the turn of the millennium, several new approximate solutions were proposed where that correctness guarantee was sacrificed for more responsive methods. We will discuss some of the proposed solutions later in this chapter. The key observation that the new systems build on, is that in the traditional approach, most of the true nearest neighbors are found early in the search process and that most of the subsequent effort is spent on scanning partitions that rarely influence the final outcome. The approximations proposed, give up the notion of finding "the" nearest neighbors and instead focus on finding "a set" of near neighbors that is likely to contain as many truly nearest neighbors as possible, but with minimal effort.

1.3.4 Strategies to divide and conquer

Most recent approximate indexing algorithms use some form of quantization to divide the vector collection into small partitions. Indexing is the process of assigning to the appropriate partition(s) every vector of the collection. Searching is the process of identifying the most profitable partition(s) to scan in order to discover the nearest neighbor(s) of the query.

Quantization associates each high-dimensional vector with a representative value. There are much fewer such representative values than there are vectors in the collection. If the representative values are *numbers*, then the quantization process is called *scalar quantization*. Random projections are typically used in approaches doing scalar quantization of high-dimensional vectors. Examples of CBIR systems that adopt this type of approach are LSH [GIM99] and the NV-tree [LAJA06] that is described in Section 1.4.2.

If the representative values lie in a high-dimensional space, then the quantization process is called *vectorial quantization*. Some of these approaches are *unstructured* as they determine the representative values such that they best fit the real distribution of data over the high-dimensional space. With regard to quality, the best performing CBIR systems are based on unstructured vectorial quantization.

We will now move on to discussing a few of the approximate CBIR algorithms in more detail.

1.4 Algorithms

Following our discussion of partitioning strategies, the first algorithm we will discuss is the k-means algorithm. It is a very popular unstructured vectorial

quantizer that is used for a wide range of applications, including several CBIR systems we will discuss shortly.

We then turn our focus on modern CBIR algorithms. We choose to split them into two categories, "in-memory" and "disk oriented". "In-memory" does not mean the algorithms cannot be extended to use secondary storage. In fact, most of them have been. It simply means that disks were not part of the initial design, and thus the cost of using disks is dealt with more as an afterthought. The algorithms covered in the "in-memory" category are Bag-of-Features (a.k.a. Bag-of-Visual Words), the Vocabulary tree and ClusterPruning. And the NV-tree algorithm is in the "disk oriented" category.

We then end this section and the chapter with a discussion of the pros and cons of the various algorithms. But we start with k-means.

k-means

k-means iteratively partitions datasets into k clusters, minimizing each cluster's internal variance, until a locally optimal distribution is found. It starts by picking k vectors randomly as the initial representatives. Then it proceeds to assigning each vector to its most similar representative, forming Voronoi-cell shaped clusters. During the assignments, a new representative value is calculated for each cluster. Then, at the end of assignments, the current representatives are replaced by the newly calculated ones. The process is then iterated again and again, replacing the k representatives after each round, until they have stabilized at a local optima.

The complexity of the k-means algorithm is NP-hard in the general case, but if k and the dimensionality of the vectors d are fixed, the complexity is:

$$O((Ndk + 1) * \log N)$$

where N is the number of vectors in the dataset and $\log N$ is the number of iteration it typically takes k-means to stabilize.

Despite the high complexity, k-means has been very popular and several extension, approximations and heuristic versions of it have been proposed [AV07, Elk03]. An extensive overview can be found in [Jai08].

When k-means is used in a CBIR system, the centroids of the k-means algorithm are used as an index to quickly prune off those regions of the dataset that do not have to be scanned against a query. How strict the criteria of what is necessary to scan will depend on weather the search algorithm is exact or approximate.

The main drawback of using k-means in a large-scale CBIR system is that the number of clusters k must be large and thus the complexity of indexing is very high. Using a small number of clusters k is not feasible as that will hurt the

search process: With a small k, each cluster will contain a lot of points and that hurts the response time of the search.

A large k also affects the search, as the search must scan so many cluster representatives to find the one(s) most similar to a given query. As k-means with a large k is so costly, developers have turned to more approximate solutions that build hierarchical index structures.

k-means is also prone to create an imbalanced cluster-size distribution. This is because its only objective is to find stable clusters, regardless of each cluster's size. It will be perfectly happy with a majority of the data residing in few dense clusters while others are all but empty. As the queries' descriptors are likely to follow the same distribution as the descriptors of the database, an imbalanced cluster distribution will become a big problem at search-time. The largest clusters will be both more frequently requested and more costly to scan, causing degradation in response time and overall performance.

In short, a CBIR system that is based on the original k-means algorithm will not fare well on today's large collection sizes. However, as well shall shortly see, optimized and/or altered versions of k-means are frequently used in state-of-the-art CBIR systems.

We now move on to describing the modern in-memory CBIR algorithms.

1.4.1 CBIR: data-in-memory design

Bag-of-Features and the Vocabulary tree

The largest breakthrough in the last decade was the so called Bag-of-Features or BoF algorithm [SZ03], also known as Bag-of-visual Words or BoW, described by Sivic and Zisserman. A good overview of BoF can be found here [OD11].

BoF was inspired by the inverted indexing technique designed to efficiently index vast amounts of very high-dimensional but sparse word histograms derived from text documents. The similarity measure of text documents is based on how many words they have in common. For every word, the inverted index database creates a list of all documents that contain it. At search-time, for every word that is in the query text, the list of all documents in the database containing that word is returned. Then, the most similar documents to the query text are found by finding the most common documents in all the returned lists.

Turning to images, the basic idea of BoF is to use a quantizer, called the *codebook*, to transform local descriptors extracted from an image into *visual-words* and then directly apply the *visual-words* on the inverted indexing schema. The *codebook* is typically some efficient variant of k-means, where each cluster plays the role of a *visual-word* and thus each local descriptor assigned to that cluster becomes an instance of that *visual-word* in the clusters inverted index. The main advantage of this approach is that ultimately the descriptor vector will not have

to be kept and no costly scanning is done during the search process.

At search-time, the *codebook* is used to discover all the *visual-words* for a query image. Then, each *visual-word* list (i.e. cluster) is retrieved and the inverted index schema derives similarity based on how many *visual-words* the query and the database images have in common. As the similarity of images in BoF is based on how many *visual-words* they have in common, there is no need to do any costly scanning of clusters and therefore there is no need to keep the bulky descriptor vectors either, since no distance calculations are needed.

Another way to view the inverted index schema is to imagine it as a vote aggregation, where each cluster is an unranked list of descriptors and every descriptor in the cluster gets to cast a vote for the image it comes from.

As was shown in [JDS08], the size of the visual-vocabulary (the number of clusters in the *codebook*) needs to be very big, or else there is not sufficient discriminative power to filter the result set to only a few most similar database images. The larger the visual-vocabulary, the more costly it becomes to use k-means as the quantizer. To address this issue, David Nistér and Heinrik Stewéníus proposed a hierarchical quantizer that they call the **Vocabulary tree** [NS06].

The goal of the Vocabulary tree is to have a large and discriminative *codebook* that still keeps the indexing and search overhead low. The hierarchy is defined by a branching factor br and a maximum number of levels L. The construction process of the hierarchy starts by doing a k-means with k=br of a sufficiently large training set. Subsequent levels of the tree are then built by dividing each created cluster into br new subsets, again and again, until the desired depth L is reached and a total of br^L clusters have been created. Each subdivision is thus only defined by the distribution of the descriptors assigned to the parent Voronoi-cell and so directly defines the vocabulary of the *codebook*. Once the Vocabulary tree has been built, it is a static structure that is used to quantize all the remaining data, and future queries.

The indexing cost of the large *codebook* is however not BoF's only problem. Let us summarize the disadvantages and then discuss them:

1. It is hard to set the number of clusters k in the *codebook* just right. If they are too few, and there is not enough discriminative power and if they are too many, similar descriptors risk falling in different clusters and correct matches are lost;
2. To handle large collections many clusters are needed. Even if an efficient variant of k-means is used, indexing can be costly.
3. As the vectors of the descriptors are not kept, no internal cluster discrimination between descriptors is possible.
4. As a result of 3), search expansions into multiple near clusters are not feasible.

5. As a result of 1), 3) and k-means tendency for imbalanced size distribution of clusters, BoF typically requires several correctly matched *visual-words* to find the truly most similar database images; If resolution of the images is small and/or the query image is severely altered, the BoF search may fail where others CBIR systems can still succeed.
6. For image collections with billions of images, even BoF will run out of memory.

To fully understand the issues above, we should recall our discussion on vote aggregation from Section 1.1.3. Specifically, the discussion about copyright detection, vote aggregation of the ranked lists and the "noise-level" created by random votes.

As described before, we can imagine the inverted index filtering as a vote aggregation. One of the key problems is the high "noise-level" that is created by allowing all the descriptors of a cluster to cast a vote. Large clusters are both more likely to be found and cast a lot of random votes. Contributing to this problem is k-means tendency to create imbalanced clusters. Because of this, BoF has a tendency to need many correct votes, so that the truly most similar images can rise above the noise.

In Section 1.1.3, we described how a near neighbor search can use it's ranked results to influence the "noise-level" either by changing the size of the neighborhood and/or by using the weighted voting policy. As the inverted index lists of the *visual words* are not ranked and no information is kept to create such a ranking, BoF can do neither. The only solutions available to BoF are to either start storing additional information or to increase the expressive power of the clusters by making each one smaller and the visual-vocabulary larger. That is, smaller clusters cast less random votes and therefore "make less noise".

This brings us to the second part of issue 1), setting the size of the *codebook*. When the size of the k-means becomes very large, not only is it more costly to process, but the risk of similar descriptors falling into different clusters is also increased.

This is a common problem in many partitioning strategies and the classical way to address it is to compensate by widening the scope of the search. The two most common methods for that are called "query expansion" [CPS+07, PCI+08] and "soft-assignments" [PCI+08]. Query expansion involves allowing the search to widen its scope to examine multiple similar partitions, instead of just one. The soft-assignments allow each descriptor to be assigned to multiple partitions, instead of just a single most similar one (called the "hard-assignment" policy). In [PCI+08], there is a detailed discussion and evaluation on applying both search extensions to the large scale CBIR task.

Unfortunately, BoF has no good way of doing such expansions without raising the "noise-level" excessively. Remember that without ranking, every descriptor

in the cluster casts a vote. Therefore, increasing the number of descriptors considered, either by larger clusters or multiple ones, is bound to also add a lot of "noise". The point of having large k was to keep the clusters small such that they do not create too much noise. It would then counter-productive, and ironic, to compensate for large k by increasing the size of the clusters.

Because of the dilemma described above, many of the extensions to the original BoF algorithm choose to sacrifice some memory to keep additional information that makes it possible to rank the *visual word* lists (i.e. to do a near neighbor search of the cluster). Examples of such approaches are Hamming embedding [JDS08] and VLAD descriptors [JDSP10], just to name a few.

In any case, setting the size of the *codebook* (i.e. the number of clusters) is a balancing act between discriminative power, search quality and indexing cost. Solving the indexing overhead with a hierarchical index is a vast improvement, but does not solve the whole problem.

Note that post processing procedures, like geometrical verification [PCI+07], can be used to remove *false positives* and are often used to re-rank the returned results.

With a hierarchical index, Bag-of-Features is a very fast method that works well when many local descriptors are available from each image. It has received a lot of attention from the research community and has inspired a whole family of similar algorithms that try to address the issues raised above.

ClusterPruning

ClusterPruning or CP [CPR+07] is an algorithm proposed by Chierichetti et al. and was initially intended for indexing and searching 400,000 dimensional vectors derived from text documents.

CP's approximate search algorithm is based on scanning a single cluster of data for the k-nearest neighbors of a query. Unlike BoF, CP keeps the full vectors in its clusters to be able to discover a query vector's k-nn accurately. The main focus at search-time, is to minimize the number of distance calculations necessary to populate the query vectors k-nn.

Like BoF, CP uses a quantizer to partition the data into small groups, and then builds an index over the groupings. At search-time, the index is used to quickly identify a single most likely partition and then that partition is fully scanned for the query's k-nearest neighbors.

The base case for CP is to build a single level flat index, where the number of calculations during search is split even between scanning the index (for rapidly finding the single most similar partition) and scanning that partition (to determine the k-nearest neighbors of the query). The even split is achieved by having $\sqrt{N}$ clusters, each containing $\sqrt{N}$ entries, where N is the size of the dataset.

The quantizer proposed for CP borrows the initial step of k-means. For a given collection of $C = v_1, ..., v_N$ vectors, the base case is to create the $\sqrt{N}$ partitions by randomly picking $l = \sqrt{N}$ vectors as *leaders*. Then, each v_i is assigned to the most similar *leader* in l. Unlike k-means however, CP does not iterate. It simply stops after that first and only round of assignments.

Once all N vectors have been assigned and the clusters created, a cluster *representative* is chosen for each cluster that is used in the final index. Chierichetti et al. proposed three options for the *representative*: to keep the randomly selected *leader* used during the assignments, to use the centroid of the cluster or to use the medoid of the cluster.

At search-time, a query point q is first compared (distances are calculated) to the set of *representatives* that make up the index, to find the most similar cluster. Then, that single cluster is retrieved and q is compared to all of the clusters vectors to find its k nearest neighbors. Thus, the total number of distance calculations is minimized to $2 * \sqrt{N}$.

It is possible to store the clusters on disk if they do not fit in the main-memory. This adds an I/O cost at search-time, typically a single disk I/O to retrieve the correct cluster from disk.

Index hierarchy For large collections $\sqrt{N}$ can become quite a large number, so large that the search becomes too cumbersome and unresponsive. To address such cases, Chierichetti et al. proposed to speed up the search by building, at clustering time, a multi-level hierarchical index that could be subsequently used at search time to reduce even further the number of distance calculations necessary during search.

The hierarchical index proposed is constructed bottom-up by first randomly selecting $l = (\sqrt[L+1]{N})^L$ *leaders*, where L is the desired index depth (the base case described before was $L = 1$). Then all the assignments of the N vectors are done, just like before, creating a partitioning where each cluster has $\sqrt[L+1]{N}$ vectors on average. The second step is to use the chosen cluster *representatives* as the high-dimensional dataset to cluster at the next level of the index hierarchy, randomly choosing a new $(\sqrt[L+1]{N})^{L-1}$ set of *leaders* and doing assignments etc. This process is repeated for each level of the index, using the *leaders* of the level below as the input data for the clustering process on the level above. It should be clear that the hierarchical index is built *after* the completion of all assignments to clusters.

The generalized formula for the number of distance calculations necessary during search is:

$$(L + 1) * \sqrt[L+1]{N}$$

where L is the desired index depth and N the size of the dataset.

While the number of distance calculations necessary during search are reduced exponentially with index depth L, the number of distance calculations necessary for the initial partitioning step grows exponentially. The initial assignment process requires $N * (\sqrt[L+1]{N})^L$ calculations. There are a few reasons why Chierichetti et al. may have found this arrangement acceptable. First was that they wanted to be able to change the cluster representatives, to centroids or medoids, and that can not be done until after the clusters have been created. Another factor may have been that they did not care too much about how long the off-line indexing process takes. At very large scale this is however not acceptable. The number of calculations necessary to index a large dataset would be extremely high; A similar observation was made in the context of BoF and motivated the need for the hierarchy of the Vocabulary tree [NS06].

Search extensions Sometimes, reading a single cluster may not yield results of satisfactory quality. In such cases, it is possible to read b clusters to answer each query; the basic algorithm corresponds to $b = 1$. The cost of retrieval then consists of accessing the disk b times and doing $(L + 1 + b) * \sqrt[L+1]{N}$ distance calculations. Note, that the value of b can be set at search-time.

Alternatively, it is possible to increase the quality of the results by assigning each vector to its $a > 1$ most similar clusters, and reading only one cluster ($b = 1$) at query time. Each cluster will then contain on average $a * \sqrt[L+1]{N}$ vectors. While it will take just as many Euclidean distance calculations as using b, a only requires accessing the disk once. The downside of using a is that it requires a times more disk space if clusters are stored on disk and it can only be set at indexing construction time.

These search extensions were discussed earlier, in our discussion about the BoF algorithm in Section 1.4.1, where they where termed "soft-assignment" (a) and "query expansion" (b). A detailed discussion and evaluation of the two approaches, applied to image retrieval at large scale, can be found in [PCI+08].

The main advantage of CP is its simplistic and yet flexible partitioning and search method. As the vectors are kept, it is possible to tune the aggressiveness and quality with the a and/or b parameters. To compensate for the cost of keeping the vectors, the number of distance calculations is kept to the absolute minimum by splitting the work according to the $\sqrt[L+1]{N}$ formula.

There are however several disadvantages. By keeping all the vectors, the storage requirement is very high. Also, unlike the Vocabulary tree, CP's hierarchical index structure only benefits the search. Because of how the index hierarchy is built bottom-up, increasing the depth L of the index will actually cause an exponential growth of the assignment costs. Bare in mind that there are $N^{\frac{L}{L+1}}$ clusters created at the bottom level. Hence, the cost of indexing very large collections is

enormous.

Assigning the dataset to randomly chosen *leaders* should in theory create evenly sized partitions. In practice however, there is always an imbalance in the distribution. If some clusters are abnormally large, regardless of where they reside in the index hierarchy, accessing and scanning them will always require more distance calculations than expected. On top of that, as we have to assume that the queries follow the same data distribution, the problem is compounded by the fact that the costly large clusters are also more frequently accessed. While the imbalance in the cluster size distribution may not be as severe in CP as has been observed in k-means, see [SJHA05], it is still a subject worthy of our consideration and evaluation.

CP was designed to be in-memory, but the authors proposed that the bottom level clusters could be stored on disk if the database was very large. However, the cost of accessing the disk(s) was not taken into consideration. Accessing disks translates into physical operations, where one access can generate multiple I/O operations when the clusters are large, or in the case of small clusters, each I/O operations will actually read a lot more data than is actually needed. As we shall see later, the only way to make the algorithm truly scalable is to take both CPU and I/O costs into account right from the start.

1.4.2 CBIR: data-on-disk design

The BoF family of algorithms and CP rely heavily on RAM to keep the algorithms responsive, and extending them to secondary storage was done more as an afterthought. The algorithms we will look at in this section use secondary storage.

First we briefly look at ImageTerrier, a disk-based implementation of the BoF algorithm. Then we discuss the NV-tree, that assumes that secondary storage is necessary and takes that fact into account right from the start.

ImageTerrier

ImageTerrier [HSDL12] is a scalable open-source search engine platform proposed by Jonathon Hare et. al in 2012. It was evaluated by indexing up to 10 million images (10 billion SIFT descriptors) using a single-pass BoF indexing algorithm. What is different about ImageTerrier from most other BoF systems is that the inverted index is disk-based instead of keeping all the data in RAM. This makes it possible to index such a large dataset on a single machine.

In the search process, ImageTerrier's average response time per image was less than 1 second for datasets under 1 million images in size. This was because most, if not all, of the inverted index lists where being cached by the OS in RAM. When they indexed the full scale set of 10 million images, there was not enough

RAM to cache all the data anymore and the disk had to be frequently accessed. As a result, the response time per image jumped to 28 seconds.

ImageTerrier is not truly optimizing its disk access as the I/O assumption is not fully integrated into the design of the CBIR algorithm that is at the hart of the system.

NV-tree

The NV-tree [LAJA06], published by Lejsek et al. in 2006, is an indexing algorithm especially designed to deal with very large amounts of data, so large in fact that the authors assume it must be stored on secondary storage. They thus incorporate the assumptions of using disks as a key component of their design.

Dealing with I/O costs severely complicates things. Not only are there new costs to take into account, but they are more complicated to estimate as the performance of disks will depend on several factors. For example, the specifications of the underlying hardware can vary, access patterns are very important and the size distribution of the data matters as well.

The basic idea behind the NV-Tree is to use a scalar quantizer that iteratively projects high-dimensional vectors, like SIFTs, onto a random line and segments the projected data into evenly sized partitions. In each iteration of projecting and segmenting, smaller and smaller segments of similar vectors are created.

Unlike the previously discussed algorithms, this segmentation process is repeated until the segments reach a predefined I/O friendly size. At that time, the segment is written to disk. As in BoF, whole descriptor vectors are not stored in the segments. What is actually written is a list of all the data vectors assigned to the segment, detailing only the imageID each descriptor comes from and the distance value of the last projection.

At search-time, a query vector is quantized in the same way to discover the most similar segment of data. That segment is then retrieved from disk in a single I/O. Unlike BoF, the k-nearest neighbors can be found by using the similarity of the last projection value as a filter. By keeping only the last projection values, a lot of disk space is saved and the costly similarity comparison of full vectors is replaced by a much cheaper comparison of just a single value.

The NV-tree has therefore many desirable features. Similar to BoF, the cost of scanning the partitions for a query-vector's nearest neighbors is minimized. But unlike BoF, there is some discriminative power preserved in the form of the distance value of the last line projection. Consequently, a k-nn search is still possible. The NV-tree is also more of a database system then many other CBIR systems. For example, it supports ACID (Atomicity, Consistency, Isolation and Durability): It is possible to add and/or remove a reasonable amount of images without having to rebuild the whole partitioning and index. And, as its database and a copy of the current index is always stored on permanent secondary storage,

restarting the NV-tree can be done quickly and safely without loss of information.

On the downside, random line projections are often only rough estimates of the true similarities. As we have discussed before, vectorial quantizers tend to give better quality than scalar ones. For example, when filtering the k-nearest neighbors based only on the last projection value, the inaccuracy of the similarity can result in a need for a large neighborhood k or else the true nearest neighbor may be lost.

Also, as each stored projection value is a relative distance estimate that is only applicable to the objects residing in the local segment, classical search extension to more than one segment is not possible. As a result, if another segment is the one most similar to the query vector's segment, there is nothing that can be done.

1.4.3 Summary

The aim of every CBIR system is to efficiently perform high quality search over a given image collection. Even if the image collection is vast, containing millions of images and billions of descriptors, the search must be responsive and/or have high throughput. The challenge to overcome is the vast amount of distance calculations necessary for good quality results.

The traditional CBIR systems tried to use mathematically sound methods to minimize the amount of work necessary, but in high-dimensional space, this proved all but impossible.

A new breed of CBIR algorithms emerged that use approximate methods, giving up guarantees of finding "the nearest neighbors" for drastically improved performance. Some rely on reducing the storage-footprint and keeping all of its data in-memory (BoF), while others are designed for secondary storage right from the start (NV-tree).

A first approximation is to prune not only the "safe" branches of the hierarchy but also the "unlikely" branches. In fact, most modern algorithms narrow the focus down to only a single most likely partition, pruning off all others.

A second approximation has to do with minimizing the cost of scanning that single most similar partition. In the NV-tree, the similarity metric is reduced to a comparison of a single value, the last random line projection. BoF does one better and gets rid of the scan entirely, as it bases the similarity between images only on the *visual word* frequency they have in common. With the need for scanning the partition removed, there is also no longer any need to keep the bulky descriptor vectors around. In the NV-tree only the imageID and a single distance value are stored for each descriptor, while in BoF only the imageID is kept.

A third approximation is to use a multi-level hierarchical index structure to quickly identify a vector's most similar partition(s), both at search time and

during index construction. For a large scale CBIR system, tasked wit indexing millions or even billions of images, a deep hierarchical index is essential to its viability.

However, all of the approximations described above come at a cost. The vastly improved performance is typically payed for with loss in search quality. A balance must be struck between the two, maximizing performance with acceptable quality loss.

Out of the algorithms discussed, BoF is the most commonly used and it has spawned a whole family of algorithms based on its core ideas. The NV-tree is the only one, and one of very few today, that assumes that secondary storage is necessary and takes I/O costs into account in its design. ClusterPruning is based on a simple but brute idea and is perhaps the most adaptable algorithm. But, its lack of using an index hierarchy during assignments is a major drawback and it also keeps all the descriptor vectors, making its storage requirement much larger than the others.

The typical CBIR search is designed for active users and thus geared toward answering each image query in a serial manner, minimizing the response time for each query. This kind of design may however not maximize throughput, as that may require the sacrifice of individual query response times for the benefit of the many. There is not always an inpatient user on the other end and thus we will also focus on high-throughput search. Scenarios that would benefit from high-throughput are copyright detection, content filtering, automatic classification/tagging or any kind of data mining, just to name a few.

As our goal, down the road, is to harness the power of parallelism and distributed computing, we also need to take into consideration the complexity of the algorithms and how difficult it will be to develop. We also assume secondary storage is a must, but only the NV-tree was really designed with that in mind.

1.4.4 Choosing algorithm for web scale indexing and search

We know that the dimensionality of the SIFT features as well as the scale of the image collections that we intend to work with are such that the traditional algorithms, like the K-D tree [Ben75], the M-Tree [CPZ97] etc., will not work. The strongest candidates are the approximate algorithms that we have discussed: Bag of Features, CP and the NV-tree.

NV-tree's main advantage is that it is designed for using secondary storage. Parallelizing and distributing the algorithm will however be difficult. The indexing has data dependencies in each round of projections and segmentation. A lot of the hard work will therefore not distribute well. Distributing the search is possible, but it is likely to require a lot of synchronizing and that can become a bottleneck. In addition, it is a patented solution that is not freely available for

academic work. We therefore exclude it as a viable option.

As for Bag-of-Features, the key concept is to get rid of the cost of scanning a partition. This is done by defining the similarity of images as the frequency of clusters that they have in common. The downside is that for there to be good discriminative power, the clusters need to be small, as they contain no information to do filtering at search time. To keep the clusters small, the number of clusters is set very high, but that causes other problems like increased indexing cost and potential loss in quality due to similar descriptors falling across cluster boundaries.

With regard to accessing data, as long as BoF is running in-memory, it will not mind having many small partitions. But remember that a key assumption that we make is that the size of the datasets we intend to work with is such that they simply cannot be kept in memory. Storing the BoF's *visual words* (clusters) on disks is possible. However, those small partitions are far from being I/O friendly and so it is likely that the cost of doing the I/O, to retrieve the *visual word's* inverted index list from disk, will become the bottleneck in a disk-based BoF system.

If the algorithm is I/O bound anyway, why not have the available CPU power do some similarity evaluations while it waits? We saw a good example of this in the disk-oriented BoF implementation in ImageTerrier [HSDL12].

Working with small images, as we intend to do, is very challenging. For such small images, the worst case is that we only get a handful of SIFT descriptors from each image. And approaches, like BoF, that base their similarity solely on cluster overlap, are faced with a daunting task when the query image only has a handful of descriptors. There simply is not enough discriminative power to derive a short ranked list of candidates.

This leaves us with ClusterPruning. The simplicity of the CP is quite attractive. It is relatively easy to parallelize and distribute, as the randomly selected *leaders* are known a priori, and, unlike k-means, there is only a single iteration for actually clustering the data.[3] CP can use a hierarchical index to keep the search cost down and still provides the means to control the aggressiveness of the approximation with the a and b parameters. However, unlike both BoF and the NV-tree, CP keeps all the descriptor vectors. This is in fact an interesting feature as we want to investigate what is at stake when managing very large collections of descriptors that are stored on disk(s). Keeping the IDs only would facilitate the job, would be more efficient, but we would miss the opportunity of investigating those specific problems.

The authors of CP proposed that the lowest level of the index could be stored on disk, but payed no attention to I/O friendly clusters size or really considered to

3. There are iterations in building the index hierarchy however.

include the I/O cost in the cost minimization formulas used. Both the clustering quality and the balance of the cluster size distribution are questionable when there is only a single step of assignments. In addition, to our knowledge CP has never been applied to images.

However, we think there is room for improvements. We therefore extend CP and create a new indexing and searching algorithm called extended ClusterPruning or eCP. eCP is designed to meet the following objectives:

- **Objective I** We want to take I/O costs into account during database construction as we assume the clustered data will reside on disk.
- **Objective II** We also want to minimize costs for search but also during database construction.
- **Objective III** We intend to evaluate how extensive the imbalance of clusters size is and find means to make the size distribution more even.

In the next chapter we will describe our implementation of the eCP algorithm and do an extensive evaluation of that implementation.

Chapter 2

Extended ClusterPruning, eCP

This chapter presents eCP. eCp extends CP in order to deal with large collections of images that are stored on disks. Therefore, the extensions presented in this chapter include mechanisms to best make profit of I/Os, to avoid the imbalance of clusters that is detrimental to performance as well as mechanisms dramatically accelerating the assignment of points to clusters.

The chapter is split into 6 sections. We start with a quick review of the most common storage systems in a background section numbered 2.1. We then move on to Section 2.2 where we describe the complexity of CP, pointing out the reasons that make the CP scheme intrinsically inefficient at large scale. Building on these issues, Section 2.3 is devoted to our proposed extensions and to giving a general overview of our eCP algorithm. This is followed by Section 2.4 that contains more specific implementation details of eCP. In Section 2.5 we validate eCP with a range of proof of concept experiments. The goals of our experiments are to validate our ideas and to establish guidelines for how to set the various parameters of eCP. Finally, we end this chapter with a conclusion section where, in addition to summing up the chapter, we discuss why we consider eCP to be a prototypical CBIR algorithm and why we think it is a good basis for researching tomorrow's large scale CBIR systems.

2.1 Background: Storage systems

The limitations of secondary storage technology plays a crucial part in the design decisions for CBIR systems that rely on them. Because of the limitations, we start with a brief review of the state-of-the-art in storage technology, focusing on the aspects that are most relevant for our work.

2.1.1 Magnetic Disks

Magnetic disks are the standard for cheap secondary storage. During the last decade, only the capacity of the disks has dramatically increased. Their read/write performance has improved little as their mechanical parts are inherently slow. There are two moving parts in the device, a rotating disk(s) that contains the data and an arm that holds the heads that do the actual reading and writing. The two mechanical costs are thus the *seek time*, the time it takes to move the arm such that the heads are correctly placed, and the *rotational delay*, the time it takes the disk(s) to be spun so that the correct area passes under the heads.

Small and random operations are greatly impacted by these two costs. In random operations, the arm has to be frequently moved and each time the disk has to be spun into place, making random operations significantly slower than large sequential operations, where the movement of the arm is minimal. Accessing data that is contiguous on disk is therefore a key to good disk performance.

In modern disks, there is sophisticated software embedded in the device controller that tries to minimize the cost of reading and writing. By using buffers (typically 16-32MB or more), the disk controller can reorder individual I/O requests, enforce contiguity and allow asynchronous writing, all to minimize the need for arm movement and preventing the costly mechanical delays. As the size of the buffers has increased, it has become common to use the buffers to prefetch more data than is requested, making consecutive requests for that data very fast. This is especially beneficial for large sequential reads.

How exactly a disk's embedded software is programmed is vendor specific and typically a trade secret. Programmers of applications have therefore little or no control over these low-level decisions being made on the disk. The general rule is to make all disk access as sequential as possible.

2.1.2 Network Attached Storage (NAS)

NAS is typically a quite large secondary storage solution made available over a network. It usually contains an array of disks, operating in parallel thanks to an advanced RAID controller, and is made available through network connected file-server(s) or dedicated hardware directly connected to the network. The performance of the NAS can be very hard to evaluate as there are many layers of hardware, caching and communication, each with its own bottlenecks. Often the network links between the server and the clients limit the throughput as the links are typically shared by many clients.

2.1.3 Solid State Disks (SSD)

SSD technology is based on flash memory chips. There are two types of memory cells in use: single-level chips (SLC), which are fast and durable, and multi-level chips (MLC) that take more space and are not as durable, but are cheaper. Recently Intel, with its 710 Series SSD, introduced new MLC technology that is nearly on par with SLC in endurance. This new SSD is targeting the needs of the enterprise with both endurance and capacity.

The Flash memory modules are typically arranged in 128KB blocks. Since there are no slow mechanical parts, reading from an SSD is extremely fast and sequential or random reads are equally fast. In contrast, writing is more costly as it sometimes requires a special erase operation to be done. Each erase operation is done at the block level. Thus, to write even a single byte may require an erase of an entire 128KB block. The cost of writing is therefore not uniform and write performance is typically unpredictable from a programmer's point of view.

In addition, to minimize write costs, internal controlling algorithms do wear-leveling to extend the life span of the chips. SSD performance has been extensively studied (e.g., see [BrJB09]).

With the release of the SATAIII standard, the potential transfer rate doubled, from 300MB/sec to 600MB/sec. In turn, the SSD vendors released 500+MB/sec. capable disks for the public market. The enterprise market, however, has shown more restraint in this area and has focused on durability and capacity instead. For example, the Intel 710 Series disks have only 270MB/s read capability, and 170MB/s write capability, far below the capacity of SATAIII.

2.1.4 Interaction with the OS

In all operating systems there exist many sophisticated routines that try to reduce the costs of accessing secondary storage. As on the disk level, prefetching is a common technique used by the OS. The idea is to use otherwise unused RAM to read more data than is asked for, in the hope that it will be requested later and thus subsequent disk read operations can be avoided from even being issued. Reads are blocking operations and the requesting process can only be resumed once the data is in memory, but writes might be handled asynchronously as there is effectively no need to wait for the data to reach the disk. Overall, the operating system uses free RAM as buffers for I/Os and fills or flushes them when it so desires, trying to overlap the I/O and CPU load as much as possible. If reads or writes are issued too rapidly, there is little overlapping and performance degrades.

2.2 The complexity of ClusterPruning

Section 1.4.1 in Chapter 1 already outlined the CP algorithm. In this section we will give a much more detailed analysis of the cost involved, both during the indexing and the search.

We will first look at the cost of doing the indexing with CP. We then turn to CP at search-time. And finally, we will apply CP to a real example of a large scale collection of 30.2 billion descriptors extracted from 100 million images. With this example, we demonstrate the complexity of CP in practice and highlight the issues that we hope to address with eCP.

2.2.1 Estimating the number of distance calculations at indexing time

To fully grasp the complexity of this process, we must recall the following important fact. The multilevel index hierarchy, eventually created by CP, is not used during the cluster assignment process. The index hierarchy is in fact created bottom-up once all the data points have already been assigned. Therefore, it is impossible for CP to use the index hierarchy during indexing as that structure does not exist yet. It is essentially created as the last step of that indexing process and it's purpose is only to speed up subsequent uses, at search-time.

CP only determines the number of nodes it needs in the final index tree and computes the distance between the points in the database and all those nodes. Therefore, we also know that the bottom-up construction results in an extremely expensive indexing in the terms of the number of distance calculations that are necessary. All N vectors have to be scanned against the initial random selection of $l = (\sqrt[L+1]{N})^L$ *leaders*. The number of distance calculations necessary for that partitioning is captured in the following formula:

$$C_{Assignment} = N * (\sqrt[L+1]{N})^L \tag{2.1}$$

where L is the desired index depth.

Another way to write Equation 2.1 is:

$$C_{Assignment} = N^{\frac{L+1}{L+1}} * N^{\frac{L}{L+1}} = N^{\frac{2L+1}{L+1}}$$

and in this format we can see clearly the exponential growth of this initial step of partitioning.

The cost (i.e. # of distance calculations) of building the index hierarchy bottom-up is captured in the following formula:

$$C_{Index} = \sum_{i=1}^{L-1} (\sqrt[L+1]{N})^{L+1-i} * (\sqrt[L+1]{N})^{L-i} \tag{2.2}$$

where the first part of the equation, $(\sqrt[L+1]{N})^{L+1-i}$, is the dataset that is being indexed (starting with the $l = (\sqrt[L+1]{N})^{L}$ initial *leaders* randomly picked and used to partition the data) and the second part, $(\sqrt[L+1]{N})^{L-i}$, is the size of the current index-level (the number of new *leaders* randomly selected for that index level) at each time step i.

The full cost of building an indexed database with CP is thus:

$$C_{DB\ construction} = C_{Assignment} + C_{Index} \tag{2.3}$$

where the dominating factor is the cost of the doing all the assignments to the $l = (\sqrt[L+1]{N})^{L}$ *leaders*.

2.2.2 Estimating the number of distance calculations at search time

As discussed before, the index hierarchy is available at search-time and we know that it can be used to greatly reduce the number of distance calculations necessary to discover the most proximate cluster(s) to a query. Note that in both formulas presented, we are assuming no search expansions are used (i.e. a and b both equal 1). Following is the generalized formula for any index depth L:

$$C_{Search} = (L+1) * \sqrt[L+1]{N} \tag{2.4}$$

where L is the desired index depth and $\sqrt[L+1]{N}$ is the average size of each cluster (I/O cost is not taken into consideration here). And the formula for just traversing the index is very similar, as we simply skip one level, replacing the $(L+1)$ with just L as follows:

$$C_{Traversing\ index} = L * \sqrt[L+1]{N}$$

2.2.3 CP applied to a real example

To get an even better understanding of the issues that arise when applying CP at large scale, we give an example using a collection of 30.2B descriptors extracted from 100M images. The numbers are presented in Table 2.1, where each column represents a different index depth setting, ranging from L=0, where there is no index and the algorithm has to resort to a sequential scan, to a six level deep index structure (L=6).

We calculate the number of clusters created on each level of the index, shown in rows 1-6. Row 6 is always the number of *cluster representatives*, one per cluster created on disk (i.e. the initial set of *leaders* used to assign all the 30.2B descriptors to clusters).

In row 7 we calculate the average number of points per cluster and in row 8 we estimate the size those points will require on disk. In these rows we can see that as the index grows deeper, the average size of clusters quickly becomes very small. And, as there is no way we can store all 3.6TB of data in memory, those clusters will have to be stored on disk. The "problem" is that for any given storage device, there is a fixed minimum cost (e.g. the *seek time* and *rotational delay* for magnetic disks) regardless of the I/O size. Thus, as the clusters become ever smaller, we will be paying the same fixed price for ever diminishing return in information. A balance must therefore be struck between the device performance and the minimum size of clusters (i.e. to minimize time while maximizing quality).

The three rows 9-11 show us the estimated number of distance calculations necessary when a=b=1: for traversing the index (row 9), to do the database construction of Equation 2.3 (row 10) and for searching a single query descriptor as captured by Equation 2.4 (row 11). Rows 12-14 are the same calculations as rows 9-11, except this time a search expansion is used, a (or b) is increased from 1 to 3.

We should note that the estimates of index traversal and the cost of searching (rows 9 and 11) are based on the theoretical assumption that the distribution of points during assignments (both of the data as well as in the index hierarchy) are balanced, creating even sized clusters every time. In practice this never happens and the size distribution is typically quite skewed. The index traversing and the scanning of clusters from skewed size distributions will result in more overhead than is estimated. However, they are fairly good estimates on average and we use them as such, knowing their limitations.

In row 9 we can observe how the index depth is good at minimizing the cost of traversing the index and therefore good for the search process (row 11). At the same time, as the index structure is not used for the assignments, we see in row 11 the drastic increase in the cost of doing the database construction. We should also note that the use of a or b makes no difference for CP's database construction (bottom-up without index), and thus row 13 is the same as row 10. The b is a search-time setting and as for a, it is sufficient to keep track of the a most similar clusters and assign the point to each.

Overall, the flexibility of CP is just limited. The hard line of minimizing only the number of distance calculations, regardless of other potential costs, is simply not very applicable to a solution that is to be based on storing its data on disk at very large scale.

Row		L=0	L=1	L=2	L=3	L=4	L=5	L=6
1	*leaders* on level 6							32
2	*leaders* on level 5						56	988
3	*leaders* on level 4					125	3.1K	31.0K
4	*leaders* on level 3				417	15.6K	174.0K	975.2K
5	*leaders* on level 2			3.1K	174.0K	1.9M	9.7M	30.6M
6	*cluster representatives*	1	174.0K	9.7M	72.6M	242.5M	543.1M	963.1M
7	Points per Cluster	30.3B	173,968	3,116	417	125	56	32
8	Clusters size on disk	3.6TB	22.56MB	413.90KB	55.40KB	16.58KB	7.41KB	2.71KB

Using no search expansion (a=1, b=1)

Row	Cost	L=0	L=1	L=2	L=3	L=4	L=5	L=6
9	Traversing index		174.0K	6,233	1,252	500	280	192
10	Indexing		$5.27\ 10^{15}$	$2.94\ 10^{17}$	$2.20\ 10^{18}$	$7.34\ 10^{18}$	$1.64\ 10^{19}$	$2.92\ 10^{19}$
11	Searching	30.2B	347,937	9,350	1,669	624	336	224

Using search expansion a=3, b=1 (or a=1, b=3)

Row	Cost	L=0	L=1	L=2	L=3	L=4	L=5	L=6
12	Traversing index		174.0K	12,466	2,920	1,248	728	512
13	Indexing	No change (index hierarchy is not used for assignments in CP)						
14	Searching	30.2B	347,937	15,582	3,337	1,373	784	543

Table 2.1: Examples of calculations cost of using CP on a collection of 30.2B descriptors that represent 100M images. The only variable is L, the depth of the index hierarchy, that ranges from L=0 (in this case there is no index and CP will do a sequential scan) to L=6.

2.3 Extended Cluster Pruning: general overview

In this section we will provide a general overview of our extended Cluster-Pruning algorithm (eCP). We will focus especially on a high-level discussion on the objectives that we set, the design choices that we make and how the eCP algorithm differs from the CP algorithm that we build upon. A more detailed low-level discussion will be provided in the Section 2.4.

We start with setting the objectives for our algorithm, including a quick reiteration of the three main objectives described at the end of the last chapter, in Section 1.4.3. We then look at the design choices we made and briefly discuss their impact on eCP. After that we highlights the contrast between the behavior of eCp and CP by applying eCP to the same large scale real example that we applied CP to in Section 2.2.3. We end this section with a thorough discussion on several topics related to the eCP algorithm both during indexing and search. Topics included are top-down vs. bottom-up index construction, the different ways a search can fail and the importance of disk access policy that is used.

2.3.1 Objectives

We would just like to reiterate our objectives for eCP that we defined at the end of Chapter 1 in Section 1.4.3. Our objectives are to address three main issues in order to create a more efficient and scalable CBIR algorithm that will be a good platform for further development. Following are the objectives we set:

- **Objective I** We want to take I/O costs into account during database construction as we assume the clustered data will reside on disk.
- **Objective II** We also want to minimize costs for search but also during database construction.
- **Objective III** We intend to evaluate how extensive the imbalance of clusters size is and find means to make the size distribution more even.

In addition to the three main objectives, we also describe a few key settings, features and optimizations that need to be figured out and evaluated in our experiments.

The size of clusters One of the main tasks is to establish a cluster *target-size*, or *ts*, that strikes a three-way balance between the CPU cost of scanning the cluster, the I/O cost of retrieving it and the search quality it gives.

We mention this task first as it greatly influences the number of clusters in the index and therefore influences the traversal cost and thus also the cost of indexing and search. Relatively high database construction cost may be acceptable. The main goal is to have fast and good quality search results. But there is no reason

to spend more time on database construction than is necessary and the time it takes has to be within some reasonable limit.

We should also note that any balance of ts arrived at will be upset if the underlying hardware is changed significantly.[1]

Index depth and search expansions We also need to evaluate the index depth L and the proposed search expansions, a and b, and come up with guidelines for setting these parameters. The guideline should cover both the case when data is on disk and in-memory, as is the case when search expansions are used in the index structure.

Size of the neighborhood If the k-nn used during the scanning of clusters is too large, we get a high "noise-level". And if k-nn is too small, we may miss the correct match. Experimentation is needed to establish guidelines for the size of the neighborhood used in the k-nn search.

Optimize the image-level search: The image-level search involves doing several k-nn searches, one for each query descriptor. The response time to minimize is that of the whole image, and not individual query descriptor. As we know that multiple k-nn queries will be issued for each image and that the search is I/O bound, there are ways to optimize the image-level response time by minimizing the I/O cost. For example by merging redundant cluster requests for the same cluster and by retrieving the clusters requested in order, such that disk access is made as contiguous as possible.

2.3.2 Design choices for eCP

We will now develop our eCP algorithm and describe four design choices that we propose, which significantly enhance performance.

1) Top-down index Unlike Chierichetti et al., we propose to always use the randomly chosen *leaders* as cluster representatives. That way, we can select and create all the levels of the index right from the start, before the first descriptor is even assigned. We also propose to build this index, top-down, such that both the index construction process and the assignment of descriptors to clusters can benefit from using the index hierarchy, dramatically reducing the database construction time.

1. CPU cost is of course determined by the processor used and the I/O cost by the secondary storage device. Changing either one may also warrant a change in ts to find a better balance.

2) I/O friendly clusters The results reported in [SJHA05] indicated that cluster size is a key factor in the performance of systems where the clusters reside on secondary storage. Therefore, the size of the clusters should be heavily influenced by the performance characteristics of the underlying devices that the data resides on. However, when we applied CP to a real large-scale example, see rows 7 and 8 of Table 2.1 in Section 2.2.3, the granularity of the clusters becomes very small as we increase the depth L of the index. While CP's behavior minimizes the number of distance calculations (and thus CPU cost) it allows little or no control of the I/O size created.

What we propose is to force the cluster size to be more I/O friendly by making it a key factor that we control. Instead of randomly selecting $l = (\sqrt[L+1]{N})^L$ *leaders* to represent clusters on disk, we propose to set the desired average cluster size and then determine the number of *leaders* as follows:

$$l = \frac{N}{(desired\ cluster\ size/descriptor\ size)} \tag{2.5}$$

We will refer to the "*desired cluster size/descriptor size*" as the cluster *target-size* or ts and thus the formula will more commonly be written as $l = \frac{N}{ts}$.

Using this new number of cluster *leaders*, the size of the index hierarchy is calculated in a similar way as before. The top level of the index will have $\sqrt[L]{l}$ representatives and each intermediate-level representative still represents $\sqrt[L]{l}$ points at the level below etc. Thus, if we would set ts to the same setting as proposed by CP, there would be no change in the derived index structures (except of course we propose to build it top-down).

While CPU cost is sacrificed on scanning the larger clusters (if the ts is forced higher then CP would have set it), the I/O cost is now under our control as we can set ts to any value we like. By decoupling the size of the clusters from the choice of L, we gain the following benefits:

1. We can tune ts to find the optimal setting for the underlying hardware.
2. Time is not wasted on unproductively small I/Os.
3. Larger cluster size means fewer clusters, and thats leads to a narrower (smaller) index that requires less memory.

3) Balancing the cluster size distribution As was pointed out in [SJHA05], while many natural clusters in a descriptor collection are very small, the largest might be as large as 5-20% of the collection. Large clusters result in both a more expensive I/O operation and additional CPU cost, while small clusters still require an I/O operation for a small information contribution. Thus, numerous imbalanced clusters can reduce performance at search-time. Furthermore, large clusters tend to get selected for scanning more often, impacting performance even further.

An algorithm like k-means discovers the natural clustering of the data. CP's simplified version of clustering does however not go that far since it never iterates beyond the first step. In theory, the random leader selection process should generate equally sized clusters; in practice, however, this never happens. It is therefore important to evaluate how imbalanced the eCP clustering is and address the potential problem if necessary and possible.

Proposed and published solutions, such as [TJA11], are effective but tend to be complicated and costly. We propose a simple, yet surprisingly effective method to balance the size distribution. We first determine the number of clusters to create according to Equation 2.5 ($l = \frac{N}{ts}$) and then we intentionally choose X% additional *leaders* to create extra clusters. At the end of the assignment process, we then eliminate the corresponding number of the smallest cluster by reclustering their descriptors into the l remaining clusters. In addition to the obvious advantage of eliminating the smallest clusters, the use of more randomly selected *leaders* gives a better sampling of the descriptor distribution and therefore also reduces the size of the largest clusters created.

4) Search expansion, in-memory and on disk CP proposed two options for search expansions. A soft-assignment policy called a, where each descriptor is assigned to the a most similar clusters during database construction. The other, b, is to expand the search to the b most similar clusters at search-time. Both policies are well described and discussed in [PCI+08]. In CP, the policy used (a or b) would then apply to all levels of the index hierarchy and clustered data.

In general, we consider a a better option as it is more focused. It expands the region of the cluster in all directions, creating a small overlap with many neighboring clusters. However, as we are expecting to work with a lot of data, we cannot realistically consider $a > 1$ as an option for data on disk. It would simply require way too much storage space to a-fold the dataset by assigning each descriptor to its a most similar clusters. For data on disk, we therefore favor the use of b. Also, as b is set at search-time, it is much more flexible, allowing failed searches to be repeated with a larger and more costly scope.

For the in-memory index structure we favor a over b as there we can solve the soft-assignments with pointers instead of actually a folding the data. However, as we shall see in the next chapter, the use of pointers can cause poor memory alignment and degrade performance when the index structure is very large.

What we propose is to allow for a separate policy being used for data on disk and data in-memory. We will keep using the a and b parameters to refer to the policy for data residing on disk. For the in-memory index structure, we define a new set of variables, *treeA* and *treeB*, that refer to our in-memory policy. Typically, our default values are a=1 and *treeA*=3.

2.3.3 eCP applied to a real example

In Table 2.2 we revisit our example of the 30.2B SIFT descriptors, this time using eCP as the basis for indexing and cost calculations. As the number of clusters created on disk is now controlled by our I/O friendly *ts* (set to 665KB here) we see that the values of row 7 and 8 do not change. With the current setting, 6,052,988 clusters are created, where the average size of each cluster is 664.06 KB.

The most striking difference between eCP and CP (see Table 2.1) is however found in row 10, the indexing cost. At all levels, except L=1, the indexing cost of eCP is only a fraction of CP's. The reason that eCP's indexing cost is much higher at L=1 is that there is no index hierarchy yet. The index is only a flat list and with a *ts* of 665KB, there are 35 times more clusters created on disk by eCP (6.1M clusters) than by CP (174K clusters). For all other levels, instead of the cost growing with the index depth L, we have an exponential reduction in the cost. This is an absolute key factor in making eCP a viable option at large scale. For example, even when we use our default setting of *treeA*=3, see row 13, the cost of indexing the dataset using eCP at L=2 is 5.67% of the cost indexing the set with CP at its cheapest setting of L=1. If both use L=2 eCP's cost ratio is down to 1.01%.

The cost of searching is shown in rows 11 and 14. We can see how the need for a deep index, that reduces the cost of indexing, makes the cost of traversing the index at search-time negligible in comparison with the cost of scanning the cluster, almost regardless of how small we make *ts*. This added CPU cost of scanning the clusters is the price we pay for controlling the size of the I/Os.

2.3.4 Discussion

In this section we will focus on a high-level discussion of the eCP algorithm. We look at the effects, consequences and limitations of our proposed design choices and try to anticipate how the algorithm will behave.

The first topic is our proposed top-down index hierarchy that is used both for search as well as database construction.

Top-down vs. bottom-up indexing What should be clear from the real example, where we apply both CP and eCP on a collection of 30.2B descriptors, is that using the hierarchical index during the assignment process is a key factor for making the indexing of large scale collections viable.

The way we build the index top-down is that we start by pre-calculating how many *leaders* should be created for all levels of the index. We then randomly pick the leaders for each level, but start by creating the top-level. As we create the

Row		L=0	L=1	L=2	L=3	L=4	L=5	L=6
1	*leaders* on level 6							13
2	*leaders* on level 5						22	182
3	*leaders* on level 4					49	516	2,460
4	*leaders* on level 3				182	2,460	11.7K	33.2K
5	*leaders* on level 2			2,460	33.2K	122,0K	266.4K	448.4K
6	*cluster representatives*	1	6.1M	6.1M	6.1M	6.1M	6.1M	6.1M
7	Points per Cluster	30.2B	5,000	5,000	5,000	5,000	5,000	5,000
8	Size on disk	3.6TB	665KB	665KB	665KB	665KB	665KB	665KB

Using no search expansions (*treeA*=1, *b*=1)

Row	Cost	L=0	L=1	L=2	L=3	L=4	L=5	L=6
9	Traversing index	n/a	6.1M	4,920	547	196	110	78
10	Indexing	n/a	$1.83\ 10^{17}$	$1.49\ 10^{14}$	$1.65\ 10^{13}$	$5.93\ 10^{12}$	$3.33\ 10^{12}$	$2.36\ 10^{12}$
11	Searching	30.2B	6.1M	9,920	5,547	5,196	5,110	5,078

Using search expansion *treeA*=3

Row	Cost	L=0	L=1	L=2	L=3	L=4	L=5	L=6
12	Traversing index	n/a	6.1M	9,840	1,276	490	286	208
13	Indexing	n/a	$1.83\ 10^{17}$	$2.98\ 10^{14}$	$3.86\ 10^{13}$	$1.48\ 10^{13}$	$8.66\ 10^{12}$	$6.30\ 10^{12}$
14	Searching	30.2B	6.1M	14,840	6,276	5,490	5,286	5,208

Table 2.2: Examples of calculations cost of using eCP on a collection of 30.2B descriptors that represent 100M images. The only variable is L, the depth of the index hierarchy, that ranges from L=0 (in this case there is no index and eCP will do a sequential scan) to L=6. The cluster *target-size* is set to 5,000 descriptors or approximately 665KB.

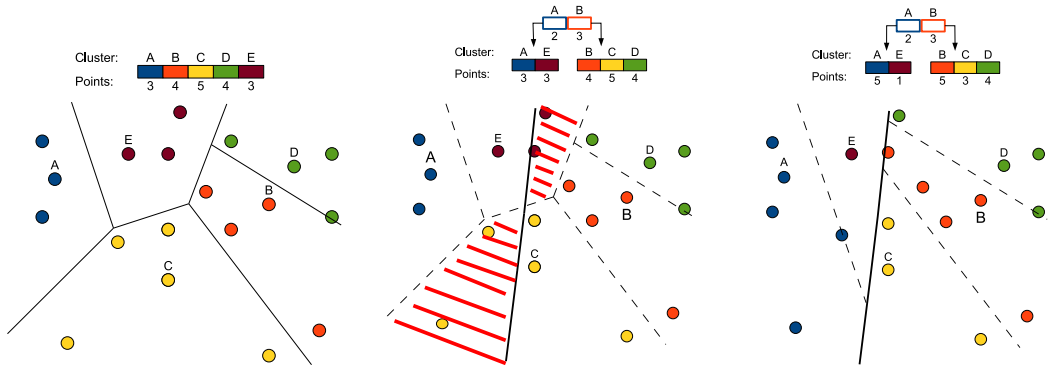

Figure 2.1: The difference between top-down and bottom-up clustering.
Left: Level 1 clustering with a flat index. No difference yet.
Center: Level 2 clustering built bottom-up using clusters A and B on the upper level. Bottom level clusters are split by upper level Voronoi-cell boundaries. Queries that fall into the red areas will follow the "wrong" top-level branch of the index.
Right: Level 2 clustering built top-down using clusters A and B on the upper level. As the index is used during the assignments, the clustering created respects the upper level Voronoi-cell boundaries all the way down. Queries may still fail, but the index structure used is the same at search-time as was used during the assignments.

second-level, we start by scanning the completed top-level, assigning each second level *leader* to the *treeA* most similar top-level *leaders*. Then, as we create the *leaders* of the third-level, we search the already completed index structure and assign each to the *treeA* most similar leaders on the level above (i.e. the second-level). This process continues for each level of the index, each time searching the already completed index and assigning each new *leader* to the *treeA* most similar *leaders* on the level above, until the full index structure has been completed. Once completed, the index structure is a static read-only structure that can be used to assign all the descriptors of the input collection.

We should note however that the clustering created this way, top-down, is not the same as the clustering that would have been built bottom-up. In Figure 2.1 we can see three figures. On the far left we see a L=1 indexing using a flat index. Points A, B, C, D and E are selected as *leaders* and the remaining points are assigned to the most similar *leader*. In this case, as no hierarchy is being used yet, the clustering created will be the same for bottom-up and top-down.

In the center image we see how CP would work, doing a bottom-up L=2 clustering of that same data. We can still see the original Voronoi-cells of the initial clustering as dashed lines. At the top level of the index we have points A and B as *leaders* and thus only two Voronoi-cells are created. The data they split between them are the *leaders* of the level below (i.e. points A, B, C, D and E) and a single Voronoi-cell boundary is created (the solid line). We can see how clusters

C and E are split by this upper-level boundary. The areas marked in red are those parts of lower-level clusters that are more similar to another upper-level *leader* (A or B) than the cluster representative of the cluster they belong to. Queries for points residing in the red areas will traverse the "incorrect" upper-level branch at search-time and thus not be scanned against the cluster where the most similar data resides. [2]

In the image to the far right we see how the top-down L=2 clustering of eCP would work for the same data. Again points A and B are used as the *leaders* on the top level of the index and points A, B, C, D and E are the *leaders* for the bottom-level (i.e. clusters on disk). However, as we can see, the clusters created are very different from what was created at L=1. To minimize the number of distance calculations, the descriptors traverse the index and can therefore only be assigned to those clusters they are scanned against, namely the bottom-level *leaders* that share the same path through the index. What is created is a clustering that respects the Voronoi-cell boundaries created at all levels of the index. As we can see, in the top-down indexing, descriptors are **not** always assigned to **the most similar** cluster representative (bottom-level *leader*), but only the one most similar that resides in that path through the index. It is therefore important to use the same index depth L both at clustering and search-time. Reducing the index depth, or even scanning all the cluster representatives, is most likely to give worse results even if a more similar cluster representative is found. In a sense, eCP's top-down clustering is getting the assignments wrong, it just does so consistently.

This brings us to our next topic, that is on the ways in which the search can go wrong.

The three ways eCP's k-nn search can fail The search process is essentially three steps: index traversal (using *treeA*) to discover the b most similar clusters, the scanning of those b clusters and a vote aggregation (see Section 1.1.3 in Chapter 1). The scanning of clusters is straight forward, but traversing the index is not.

We will define the traversing of the index, from the top-level to the *cluster representatives* on the bottom-level, as the *path* through the index. However, the last step of the *path* is different from the others as it involves selecting from a sub-branch of *cluster representatives*, i.e. selecting what clusters the query will request for scanning. The difference is that here is where the search expansion b is applied, while in all other steps of the *path*, the search expansion was controlled by the *treeA*.

2. We are assuming no search expansions are being used (i.e. a=1 and b=1)

We can therefore identify two different ways the search can "go wrong" in traversing the index. Either the "mistake" happens before the last step, and the search is lost in the wrong subsection of the hierarchy, or the mistake is done in the selection of *cluster representatives*, i.e. the "wrong" cluster was selected for scanning at the bottom-level. If the former is the case, there is not much we can do at search-time as *treeA* is set during indexing. In the latter case however, re-searching with a greater search expansion b will eventually result in the scanning of the cluster where "correct-match" was assigned.

It is therefore interesting to know how frequent the two cases are as that can tell us what is more important, using large *treeA* or searching with large b.

Let us get a good understanding of the three ways the search can fail by taking an example. Assuming that we have a multi-level index structure, a query descriptor q and for that q there is in the known ground truth "correct match" descriptor m that is in the database. The first two ways the search can fail, as we already discussed, are in the index traversal. First is that the index traversal gets lost above the last step of the *path* (hopelessly lost). Second is that the index traversal gets lost in the last step of the *path* (increasing b will work).

The third way is that the correct cluster is scanned but m's distance rank is lower than the size of the k-nn and thus falls outside the neighborhood and is not returned. We should note that this also means that even though m is the "correct match", there are descriptors in the database that are more similar to q than m. In content based image retrieval, where the images have often been heavily distorted in various ways, this is quite common. This third way to fail can be addressed at search-time by re-searching with a larger k, but that comes at a prices as a large k also increases the "noise-level" in the vote aggregation (as was discussed in Section 1.1.3 in Chapter 1).

As the scale of the datasets grows larger, eCP's answer is to create more and more clusters that widen the index. To keep the cost of traversing the wider index down, eCP increases the index dept. The result is increased approximation, with a longer *path* and relatively fewer *cluster representatives* scanned, making it more and more likely that m's and q's *paths* are different, and that we need larger *treeA* and b to compensate. We should therefore expect that as the number of clusters grows, it becomes harder for eCP to cope with a large distance between q and m and the deeper the index structures is, we should expect more *path* failures before the last step, that are so hard to do anything about.

Intuitively, we are inclined to use a vast number of clusters and a deep index for a large dataset. But where we draw the line between performance and approximation has no easy nor straightforward answer.

Setting the depth of the index hierarchy While making the index deep reduces the cost of traversing it, such a setting also makes the *path* longer and, as we just discussed, that degrades the search quality. From the example of applying eCP to real large scale problem, see Table 2.2, we can see in row 9 the cost of traversing the index. At first the number of calculations are reduced quickly as the index depth L becomes deeper, but then the gain diminishes at each added depth. For example, the level 4 deep index requires 196 distance calculations while the level 3 deep index requires 547 distance calculations. The ratio between them is therefore 196/547=35.83%. The 5 level deep index requires only 110 distance calculations to traverse and the ratio between it and level 4 is 110/196=56.12%. This clearly shows the diminishing return in reducing the traversing cost with a deeper index.

Even with 6.1 million clusters created on disk, there are very few top-level *leaders* created at L=5 and L=6. The top level tells us how many clusters there will be on average in every sub-branch (assuming *treeA*=1) and if this value is very small, the granularity of clusters in the index is small and the search is more likely to make mistakes.

The risk is that the search expansion (*treeA*) will need to be increased to compensate for the quality loss from using a "too" deep index. As we can see from row 12 of the same Table 2.2, the increased overhead of *treeA*=3 is considerable, increasing it further will quickly become very costly. Any additional gain obtained from using a deeper index would quickly be undermined if it comes at the cost of a greater search expansion.

We will now turn to the issue of imbalance size distribution of clusters and why this is a concern for eCP.

The importance of addressing the cluster size distribution We should start by pointing out that the imbalanced clustering is an issue both for the clusters created on disk, as well as for the clustering done on the lower-levels of the index (i.e. all levels below the top). Therefore, it affects not only the search during the scanning of clusters but also the traversal of the index and thus also the assignment process in the database construction.

Not only are the large clusters more expensive to scan, but the problem is compounded by the fact that they are also more frequently requested (assuming query data follows the same distribution as the indexed data).

Undersized clusters are also a problem for the search as they still require I/Os to be issued, but scanning them will provide little added value. It would be possible to compensate for small clusters by having more clusters, but that will strain the RAM footprint of the index structure. In either case, the reality is that eCP's formulas are optimized for an estimated *target-size* and large deviations

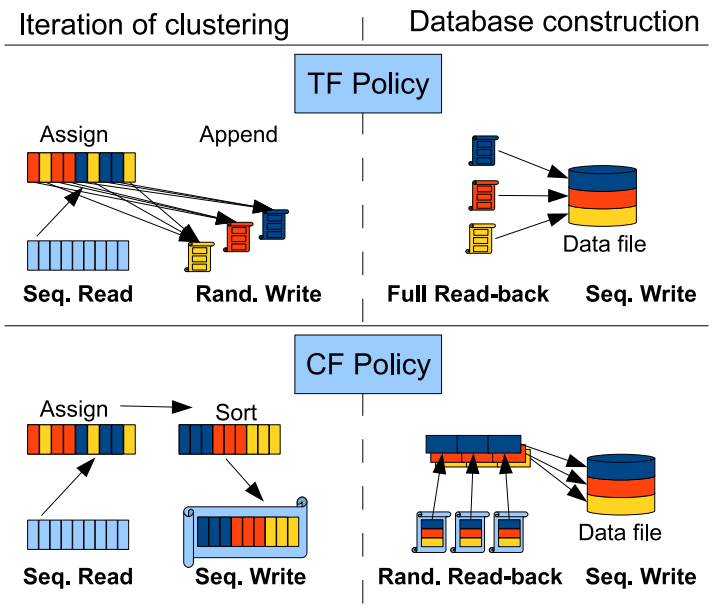

Figure 2.2: *TF* and *CF* policies in graphical representation.

from that estimate will result in more costly indexing and search. What we do not know, is the extent of the problem and our proposed solution, of picking X% extra clusters, only addresses the problem on the disk level.

Iterative indexing and disk access policy During the indexing, the amount of data to process is typically much larger than the available memory. Therefore, the assignment process will have to be done in rounds, creating intermediate file(s) at the end of each round. From the intermediate the final database file is then created. How the intermediate files are handled during the indexing will have a great impact on the performance of eCP.

The indexed database consists of three files: a data file where the descriptors are grouped by clusters; an index file containing the index structure and offset information to access the datafile efficiently; and a configuration file with general information about the database.

As stated above, the assignment process proceeds in rounds. In each round a chunk (typically 128MB or more) of the raw data is read into RAM. We call this the *in-buff*. Then, eCP's index structure is used to assign each descriptor in *in-buff* to its a most similar clusters (although we typically do not use $a > 1$). Before the next round can start, *in-buff* has to be freed up by writing the assigned data to intermediate files.

One simple and intuitive policy would be to create a file for each cluster that

would contain all the data assigned to that cluster so far. We will call this policy TF, short for *temporary file per cluster*. At the end of each round, all the clusters are looped through and each file opened, appended to and closed, as they are too numerous to remain open. Then, once all the data has been assigned, the final database file is created by simply concatenating all the temporary cluster files. In terms of access patterns, TF performs, at cluster assignment time, large sequential reads to fill the *in-buff* with new descriptors as well as many small random writes, one per cluster, every time all the data in *in-buff* has been processed. In the concatenation of the intermediate files into the final database file, TF performs cluster-sized sequential read of each intermediate file and large sequential writes to the final concatenated database file. However, as the ts of clusters is typically very small, the actual read-back of the intermediate files is a random-read pattern.

The second policy, and the one we typically use, is called *Chunk-Files* or CF. This policy is based on a sort-merge principle. At the end of each round of assignments, the descriptors in the *in-buff* are sorted based on the assigned cluster and then written into a single intermediate file before being filled again with new data. Then, once all the data has been assigned, CF merges all the sorted *chunk-files* in a merge-sort like manner. In terms of access patterns, CF performs, at cluster assignment time, large sequential reads to fill *in-buff* and large sequential writes when creating each *chunk-file* (each *chunk-file* is the same size as *in-buff*). When creating the final database file, it performs many small random reads, to gather all the cluster data from all the intermediate *chunk-files* and a large sequential write to the final database file.

The advantage of the CF policy is that there are much fewer small random I/Os used. Even the small random reads that are still present, i.e. the reading of the cluster data back from the sorted intermediate files, will benefit heavily from prefetching as, eventually, all the data from each file will be read, in order.

A graphical representation of the two polices can be found in Figure 2.2

Optimizing image level search One of our objectives is to try to take advantage of the fact that we know the image search is actually a set of k-nn searches, one for each query descriptor extracted from the query image. We have come up with two optimizations.

The first optimization is to minimize the image-level I/O cost by eliminating redundant cluster requests and increasing the contiguity of the disk access. This is done by first traversing the index structure for all the query descriptors, discovering what clusters they will want to access, before we start reading any data. Knowing this information allows us to group and re-order the cluster requests, eliminating redundant requests and maximizing contiguity.

The second optimization has to do with terminating the image search early,

before all query descriptors have been scanned and thus before all the clusters have been read from disk. This process is called *early-halting* and is based on the idea to vote aggregate at some interval those k-nns that have already been completed and check if a ranked result list can already be determined with a high probability. If so, then the I/O requests and cluster scans necessary for the remaining k-nns can be skipped. The earlier the termination can be done, the bigger the benefit of halting early.

In the copyright detection scenario (that we frequently use), *early-halting* is often very effective. This is because in this scenario, only the top-ranked image in the result-list is really important and the logic of *early-halting* is therefore greatly simplified.

In Section 2.3.1 we defined three main objectives. Number one was to take I/O into account, second was to minimize the overhead of indexing and third was to evaluate and address the imbalance of cluster size distribution.

With the extensions we have proposed for the eCP algorithm, we feel that we have met all three main objectives. Objective one was meet by defining the I/O friendly *target-size* for clusters and objective two has accomplished by building the index hierarchy top-down and used it during the assignment process. The third objective we address with the idea of of indexing with X% extra clusters and then removing the same number of the smallest clusters. Because of the better sampling, this will reduce the size of the largest clusters created as well. However, we still need an evaluation of how imbalanced eCPs size distribution is, how much it effects the search performance and whether indexing with $X\%$ extra clusters is worth the added effort.

2.4 Implementation details

Essentially, the indexing process is a very different algorithm than the search. The only thing they have in common really is that they both use the index hierarchy that is constructed at the beginning of the indexing process. We will start by describing how the indexing is implemented, then we describe the search process and finally we will describe the implementation of the *early-halting* policy we experimented with.

Indexing The eCP database construction can be split into three phases:

Phase #1: Index creation. During this phase, cluster representatives are picked from the collection and organized in a in-memory L deep hierarchy.

Phase #2: Assignments. Descriptors are assigned to clusters in rounds of executions, each round fills the *in-buff* and writes to the intermediate files

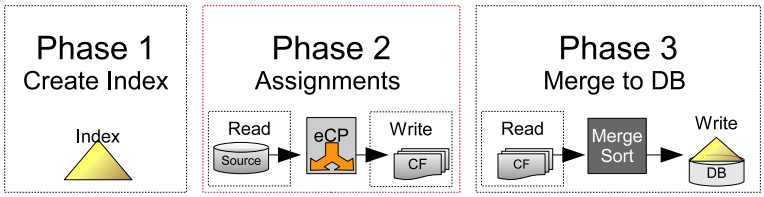

Figure 2.3: Here we see the 3 phases of eCP: #1 Index creation, #2 Assignments and #3 Merging. In this description the disk access policy is CF (our default policy), but the essential steps are the same when using TF.

according to the disk access policy used (CF of TF).

Phase #3: Merging. This phase creates the three database files: the datafile, the index file and the configuration file. How the datafile is created depends on disk access policy, but ultimately, it should contain all the assigned descriptors, grouped and ordered by clusters. The offset information of each cluster in the datafile is included in the index files for fast and efficient access at search-time.

A graphical representation of the three phase process can be seen in Figure 2.3.

The process of building the index hierarchy, i.e. phase #1, has already been described, but essentially the index is built top-down, using the already constructed hierarchy to quickly build the lower levels. This is a CPU intensive process but it takes relatively little time in comparison with the other phases.

The second phase takes by far the most time and is both very CPU intensive, traversing the index and doing the assignments, and requires a lot of I/O operations at the start and end of each round, filling and flushing the *in-buff*.

The third phase requires very little CPU, regardless of disk access policy, and is thus mostly an I/O intensive process. This process also requires relatively much less time than the CPU intensive second phase. The amount of time still depends on the disk access policy used and the performance of the underlying hardware.

During the indexing process, RAM is a scarce resource. The index structure requires its share of RAM, so does the *in-buff* and whatever RAM is left will be put to good use by the OS as buffers, to do prefetching and other I/O cost minimizing methods. The only variable we can directly control, without affecting database constructed, is the *in-buff*. Intuitively we expect that a large *in-buff* will give better response time as the larger the *in-buff* is the fewer the rounds of assignments and also the fewer times data has to be written to disk. This should be especially beneficial to the TF policy and the costly random-writes of appending data to each cluster file.

We should also note that during the indexing we need quite a lot of capacity on the secondary storage. If we delete the input collection before the merging of

the intermediate files we get the best case of needing twice the storage capacity of the input collection. More typically (as we do not want to delete the source file), we need 3 times the storage capacity, once for the raw source file, once for all the intermediate files and once for the final database data file.

Searching As has been mentioned, the search process of a single query descriptor can essentially be split into two parts. The traversing of the index to discover the b most similar clusters and the scanning of those b clusters for the k-nearest neighbors.

At the image-level however, tens, hundreds or even thousands of query descriptors have to be searched for each image. An added process is also required, the vote aggregation, that converts the descriptor-level ranked k-nn results into a single ranked list of the most similar database images.

The process of searching a query image proceeds as follows:

1. The query image is read from disk and the query descriptors are extracted. [3]
2. All the clusters to retrieve and scan are discovered by traversing the eCP index structure for each query descriptor.
3. A lookup-table is created such that for each cluster, the query descriptors that want to scan that cluster can quickly be found.
4. The clusters are retrieved in-order, to maximize contiguity of the I/Os, and scanned against the requesting descriptors, to fill their k-nns.
5. An image-level ranked result is created by vote aggregating the k-nn's with a one-vote-per-neighbor policy.
6. The image-level result is written as text to a result file on disk.

As was described in Section 2.3.1, one of the objectives is to optimize image-level queries. The first optimization described there is why we do the discovery of all the clusters first (steps 2 and 3), before we start reading data in step 4. This gives us the ability to both remove redundant requests as well as to get more contiguous access of the clusters. However, when the database is large and the amount of clusters far outnumbers the query points extracted from a single image, the contiguity becomes less important. This is because typically the access between the accessed clusters is too high for prefetching to give any advantage. Any redundant requests are still removed though and for some images, that can make a difference. Also, as we shall see in the next chapter, there is another good reason for doing things this way.

3. In our case this has already been done and we read the extracted descriptors directly from a file on disk.

Early-Halting In that same image-level optimizations objective, in Section 2.3.1, a second optimization was described. This optimization is aimed at terminating the search early if it is highly likely that the scanning of the remaining query points will not change the rank of the most similar database images already established. The earlier the search can be stopped, the more clusters can be skipped and the better the response time will be. This is called *early-halting*.

We will now describe the policy that we experimented with as a proof of concept for this strategy of early termination. To terminate the search earlier, we must do a vote aggregation and check the probability that a likely result has already been established. Typically, easy queries will quickly rise above the "noise-level".

In our case of copyright detection, the probability calculations become even easier as we are only interested in whether the top ranking images will change or not. For "easy" queries, the "correct" database image gets by far the most votes and will quickly rise above the other candidates. By aggregating a completed subset of the query descriptor k-nns, this high vote ratio can be detected and a winner declared with very high probability. Also, if a fair amount of the k-nns have been completed and yet no strong candidates discovered, it is possible to declare, with high probability, that there will be a tie. For the copyright detection, a multi-way tie indicates that the query image is not in violation. Unfortunately, a lot more k-nn's are needed for this case as the "noise-level" is typically very low and it will only take a few votes for a winner to emerge.

The down side of *early-halting* is that a vote aggregation has to be done each time we try to halt early and doing the vote aggregation takes both CPU power and causes the attention to be shifted away from the task of scanning clusters. While this may sound trivial, constant switching of tasks will cause degraded response time. We therefore limit our strategy to only 5 vote aggregations, at fixed intervals.

We should also stress again that the strategy that we come up with is only a proof of concept, not an optimal termination policy. Following are the four simple rules that we used for determining early termination:

1. If the top voted candidate image has twice as many votes as the second highest candidate, a winner is declared.
2. A fixed minimum number of k-nns must have been completed before any vote aggregation.
3. Only four early aggregations will be done, at 10%, 20%, 40% and 80% of completed k-nns, given that rule 2 is satisfied as well.
4. If $b>1$ is used, only k-nns that have been scanned against their most similar cluster request, count toward the thresholds.

The second rule is necessary as many query images have only a handful of

Dataset	Dimentionality	#Descriptors	#desc./img.	Size on disk
20M	72	20,000,000	667	1.5GB
110M	128	110,000,000	1,100	13.6GB
ANN_SIFT	128	1,000,000,000	n/a	126.7GB

Table 2.3: The datasets used in this chapter to evaluate the eCP algorithm.

query descriptors extracted from them. With the simple metric used in the first rule, a vote aggregation without a minimum threshold would not be viable. The last rule is to prevent vote aggregations with poor quality results. If care is not taken, it is very possible to do the first vote aggregation without having scanned a single "most-similar-cluster" for any of the query points.

As can be seen by the four simple rules, we only halt early if we can declare a winner. We do not try to declare ties. The probability calculations are more complicated and the low "noise-level" makes declaring a tie harder.

Replacing our simple policy with a more sophisticated one should be a straightforward task.

2.5 Proof of concept experiments

The datasets used in our experiments are described in Table 2.3. The two image datasets, 20M and 110M, are not very large. The 20M set was used in our earliest evaluations and the 110M set was limited in size by the capacity of the secondary storage devices we used, especially the low capacity of the Solid State Disks. [4]

The experiments and results presented in this section are mostly based on two of our published works. The first paper [GJA10] was published at ACM MM 2010 VLS-MCMR workshop in 2010 and the other [GAJ12b] was published in 2012 at the FLASHDB workshop, held in association with the DASFAA 2012 conference. Over a year passed between the two sets of experiments and thus they were done with slightly different versions of eCP, with different datasets and different hardware. The most recent experiments are the point-level evaluation of the eCP index structure using the ANN_SIFT collection. This set of experiments has not been published elsewhere.

4. Note that during the indexing, the need for storage capacity will be at least 3 times that of input file. The 13.6GB 110M collection will thus require 40.8GB of storage space in the course of a single experiment.

Disk	Type	Spec. Ave. Seek Time	Spec. Ave. Rot. Latency	Spec. Cache Size	Measured Seq. R/W Thr.put.
Seagate	Magnetic	11.0 ms	4.16 ms	8 MB	46/40 MB/s
Fujitsu	Magnetic	11.5 ms	4.17 ms	16 MB	68/53 MB/s
STalent	SSD	<1 ms	-	Unknown	124/34 MB/s
Intel	SSD	<1 ms	-	16 MB	220/66 MB/s
PV MD1200	RAID5	-	-	256MB*	207/80MB/s
PV MD1200	NAS	-	-	>128MB	63/42MB/s

Table 2.4: Key storage device performance indicators for experiments with the 110M dataset. Cache size marked with * indicates the RAID controller memory.

2.5.1 Hardware specifications in experiments

Experiments with the 20M set were run on a Dell PowerEdge 1850 machine, equipped with: two 3.2GHz Intel Pentium 4 processors with 1MB cache, 2GB of DDR2-memory, and two 140GB 10Krpm SCSI disks. The machines OS is CentOS 5.0 Linux (2.6.18 kernel) and disks are formatted with the ext3 file system.

Most experiments with the 110M set, all but the ones that use the RAID5 or the NAS, are run on a Dell Precision T3400, equipped with: a single 3.0GHz Intel E6400 dual core CPU with 6MB L2 cache, 4GB of DDR3-memory and we use various secondary storage devices (see Table 2.4. The machines OS is Debian and all local disks are formatted with the ext3 file system.

The experiments with the 110M set that use the RAID5 are run on a Dell R710 equipped with: one Intel E5620 4-cores at 2.4GHz, 8GB RAM and the disk volume is a DAS connected Dell PowerVault MD1200 with 15 7200rpm SATA disks in a RAID5 configuration. The RAID controller is a Perc 6/i with 256MB of RAM. The size of the volume is 15TB of usable disk space. This machine is a file server and hosts this volume as a NAS. The machine that does the work in the NAS experiments is a Dell PowerEdge 2950, equipped with: two 3.0GHz Intel Xeon 5160 dual core CPU with 4MB L2 cache, 16GB of DDR3-memory. The NAS is a NetApp. 3070 that accessed via a 1Gbps Ethernet network.

The descriptor level experiments using the ANN_SIFT set were run on a Dell R710 equipped with: two 2.67GHz Intel X5650 6 core CPU with 12MB L3 and 144GB of RAM. The secondary storage volume is a DAS connected Dell PowerVault MD1200 with 12x 15k-rpm, 600GB SAS disks organized in a single RAID6 configuration by a Dell PERC H800 controller with 512MB of RAM. The size of the volume is 5.7TB of usable disk space. This disk volume is not in the Table 2.4 as it was not used in the image-level experiments with the 110M dataset.

Please note however that ALL experiments in this chapter use only a single core for both indexing and searching.

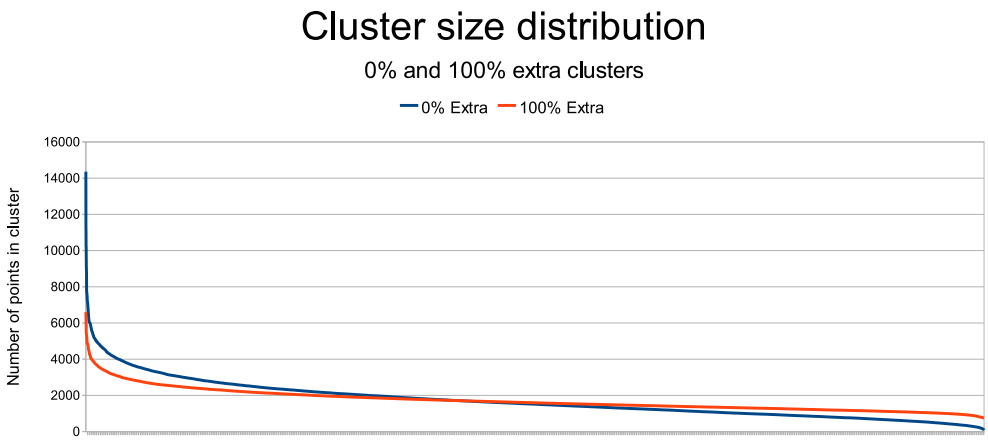

Figure 2.4: Size distribution of eCP clustering. The graph shows the size distribution for all clusters with no extra clusters (blue) and 100% extra clusters (orange).

2.5.2 Analyzing and evening out the size distribution

We are especially interested in the size distribution of the clusters created by eCP's simple top-down assignment approach. We are hoping that our size distribution will be far from the natural distribution that has been reported as high s 5-20% of the collection [SJHA05].

For this first experiment we use the small 20M descriptor dataset (see Table 2.3) and to avoid repetitive hardware descriptions, all the hardware descriptions are given in Section 2.5.1, organized by dataset. The cluster ts used is 1,724 descriptors or 128KBs cluster size on disk and that gives us $leaders = \frac{N}{ts} = 11,860$ clusters on disk. We chose to set the index depth to L=2, i.e. two in-memory levels, and in the index we use our default value of $treeA$= 3, and no duplication of data on disk, i.e. a=1.

We then build the database and the results can be seen in Figure 2.4. We also test our idea [5] to even out the size distribution. In the graph we have cluster on the x-axis and the size on y. The clusters have been sorted left-to-right, with the largest clusters to the left. The blue line depicts the default clustering, with just 11,860 random descriptors as *leaders*.

The largest cluster contains 14,353 descriptors, while the smallest has only 76. Although this is not the even size distribution we wanted, it is still nowhere near the natural cluster size of 5-20%. The largest cluster is 8.3 times larger than our desired target size but still only about 0.07% of the size of the dataset. The orange line in that same graph depicts the size distribution when we pick 100%

5. To intentionally pick extra clusters and remove them by reclustering the smallest ones.

extra clusters, i.e. there are initially 23,720 randomly selected *leaders* and 11,860 of the smallest clusters are removed and reclustered. We will come back to this graph again later.

In the two graphs of Figure 2.5, we have a more detailed size distribution analysis using various number of extra *leaders*. To simplify the graphical representations we have aggregated the size distributions into five categories: 1) clusters with $< 1K$ descriptors; 2) $1 - 2K$; 3) $2 - 3K$; 4) $3 - 4K$ and 5) those with $> 4K$ descriptors.

We then plot the cluster and data distributions in two area-graphs. On the x-axis we have five different clusterings, using from 0-100% extra clusters. On the y-axis we have the percentage of clusters (left graph) / descriptors (right graph) that fall into each of the five categories. In graph on the left, we have the cluster size distribution and on the right we have data distribution.

With this representation of the data we get a better overall picture and better understanding on what actually happens when we cluster. For example, focusing on the blue $< 1K$ category, in the left graph, we can easily read that in the original clustering (column 1 labeled 0%) just over a quarter of the clusters are smaller than 1,000 points. Reading the graph on the right hand side, we see that those same clusters only represent about 10% of the data.

Knowing the data distribution is in a way more important as it tells us how likely those clusters are to be accessed. Again, we can see that the largest category $> 4K$ (maroon) is only about 4% of the clusters (that is not so bad, right?), but we also see that they contain about 11% of the data. I.e. every 10^{th} search will scan one of those clusters.

Accumulating categories is also easy, $> 3K$ (maroon + green) has 12% of the clusters but about 25% of the data (every 4^{th} scan).

The region we would like to be dominant is the orange category of 1-2K descriptors, as that is where our cluster *targets size* of 1,724 descriptors is.

So far we have only been reading a from a single column of the graphs, column 1 where no extra clusters are used (0%). As we move along the x-axis, from 0% to 10%, 20%, 50% and 100% extra clusters, we can see that our simple idea, picking X% extra random *leaders* and removing the X% smallest clusters by reclustering them, is quite effective. It both eliminates small and large clusters, evening out the size distribution.

At 50% extra clusters, the desired orange region has grown from 36% of the data (42% of clusters) to 55% of the data (66% of the clusters). By the time we reach 100% extra clusters, we have just over 60% of the data (70% of the clusters) in our target region.

Coming back to the graph in Figure 2.4, the second result, depicted by the orange line, shows the full size distribution for the 100% extra clustering (i.e. the full distribution for the last column of the graphs in Figure 2.5). As expected,

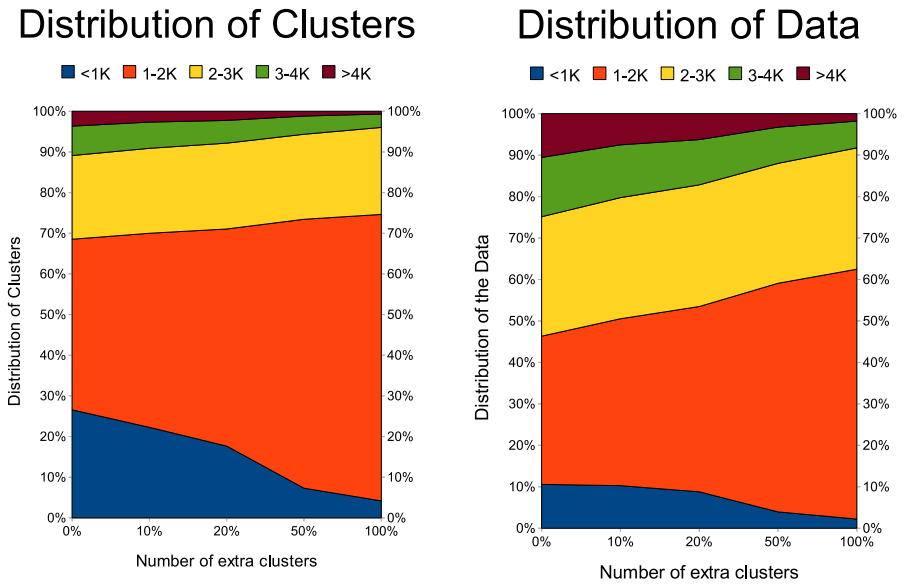

Figure 2.5: Size distribution of eCP clustering grouped into 5 size categories. Here we see stacked area graphs that show the distribution after being grouped. On the left we have the distribution of clusters into the five categories and on the right we have the distribution of the descriptors.

we can clearly see the elimination of the small clusters (the smallest has 756 descriptors instead of 76), but the distribution curve has also leveled out and there are far fewer large clusters. Those few large clusters that remain are much smaller, with the largest containing 6,599 descriptors, down from 14,354, and overall only about 8% of the data remains in clusters in the combined category of $> 3K$ (maroon + green).

Successful as this may be at evening the size distribution, there is a price to pay. Reclustering the smallest clusters requires added overhead, but that cost is not our main concern. The majority of added overhead comes from doing the initial clustering of the whole dataset with the X% extra *leaders*. Clustering with 10% extra clusters took 1.29 times longer, at 20% it was 1.56, 50% took 2.43 and at 100% extra clusters the assignments took 5.07 times longer. The cost does not grow linearly and there are a few reasons why that is. One of the reasons for this is that we are also having imbalance issues in the index itself. Thus, the cost of traversing the index is higher than otherwise estimated. In addition, we are using the $treeA = 3$ in the index and that tends make the uneven index-size-distribution problem even worse. It is possible to apply the same X% extra cluster policy on every level of the index, but that will severly complicate the index construction process. Also, it is possible use a deeper index hierarchy to compensate for the

added cost of indexing (i.e. increase L by one).

However, the imbalance problem is nowhere near as bad as we had feared. Therefore we are aware that our idea for balancing the cluster distribution works, but **we will not** be applying it in future experiments.

2.5.3 Experiments to determine appropriate eCP settings

We need to determine guidelines for setting several settings. Among them are the cluster *target-size*, the index depth L and the appropriate setting for the *in-buff*.

We will start our work by examining the effects of the size of clusters, i.e. how to set the ts.

Cluster size For this first initial experiment we use the small 20M descriptor dataset (see Table 2.3) and to avoid repetitive hardware descriptions, all the hardware descriptions are given in Section 2.5.1, organized by dataset.

The setting of ts is a three way balancing act between the time it takes to do the indexing, the response time of the search and the search quality. For the purpose of evaluating the search, we created a set of altered images to use as queries. We start by first randomly choosing 120 images from the collection and then from those 120 images, we use 26 different StirMark variants (resizing, cropping, compression, rotation, etc.), to create 3,120 altered (or you could call them attacked) images that we will use as our queries.

We index and search the 20M set using a L=2 deep index and five different ts settings. The ts settings evaluated range from 32KB to 512KB on disk.

> **Please note** that it may not be possible to get a 100% correct identification between all variants created and the original image. We base our ground truth on knowing what original image each variant comes from. That does however not tell use if it is actually possible to discover that fact. Only a full sequential scan can actually determine if a variant of the original is truly discoverable or not.

In Table 2.5 we can see the results. In the first two columns we have the five ts settings. We see the number of points per cluster and the size on disk (2^{nd} column), as well as the number of clusters created on disk (1^{st} column). The relative indexing times are in the column 3, highlighted in blue, and the search results are in columns 4 and 5, highlighted in red.

In the table we choose to use the ts=128KB, 3^{rd} row highlighted in green, as a reference point for both indexing and searching. Each of the 3,120 query images is searched in sequence, one after the other, and we report two values: the

Clusters	Cluster *ts*		**Indexing**	**Search *b*=1**		**Search *b*=3**	
on disk	KB	#desc.	rel. time	time	P@1	time	P@1
2,964	512	6,898	x0.35	x3.01	73.41%	x7.32	75.56%
5,928	256	3,449	x0.58	x1.63	73.73%	x3.65	74.86%
11,859	128	1,724	**1.00**	**1.00**	71.91%	x2.05	74.66%
23,719	64	862	x1.48	x1.00	71.23%	x1.40	74.47%
47,438	32	431	x2.21	x0.96	68.57%	x1.31	73.51%
11,859	128	1,724	1.00	x3.05	71.91%	x8.45	74.66%

Table 2.5: Balancing indexing cost, search cost and search quality using the 20M dataset. The settings are L=2, k=20 and the *in-buff*=128MB. The only variable setting is the size of clusters ts. We use the 128KB ts setting as a baseline (highlighted in green). All rows have hot cache, but in the last row the RAM had been reduced from 2GB to 750MB.

total wall clock time relative to the baseline setting of ts=128K and b=1; and the proportion of the queries where the correct image was the top voted candidate or precision@1 (P@1).

Note that in this set of experiments, the dataset (1.5GB) is smaller than the RAM (2GB). Thus, after the first few images, the search is running on hot file-cache. This is unfortunate, but it is impossible to both flush the cache between each query and, at the same time, make accurately running time measurements. We therefore also report the running times for the reference setting after reducing the RAM to 750MB, see the bottom row that is highlighted in gray. While the cache is still hot, the whole file cannot be cached anymore and now the average processing time is 4sec per image.

Not surprisingly, the indexing cost is strongly influenced by the number of clusters and thus ts. The growth of indexing time is however sub-linear, this is due to the benefit of using a two level deep (L=2) top-down index hierarchy. we see also how index depth can be used to mitigate the cost of having a small ts.

Turning our focus on the search, we see that scanning 3 small clusters gives better precision than a single 4x bigger one. For example, searching three (b=3) 128KB clusters is both faster and has a 1.25% advantage searching a single (b=1) 512KB cluster. The speed-up is understandable, fewer points are scanned (3*128KB=384KB) and as we are running on hot cache, retrieving 3 clusters instead of just 1 has very little added overhead. The increased quality is however more interesting and may be an artifact of eCP's simple one step clustering. If the query and its correct match fall close to a Voronoi cell boundary, it can happen that they fall on opposite sides. This will happen regardless of the cluster size and probably more frequently in eCP then in a clustering that minimized the internal variance of clusters (like k-means).

We also observe that the largest setting of ts always has a significantly higher overhead (more descriptors to scan in each cluster) but more interestingly there is only a marginal improvement in the quality. The lowest setting, of ts=32KB, on the other hand has a significantly higher indexing cost as there are so many clusters created on disk and thus the index structure is much wider and more costly to traverse. We also see that the quality of the search suffers but yet it takes almost the same amount of time as the 64KB setting.

The 32KB and 512KB settings are not viable options, but the other three are. We choose the middle ground in the 128KB setting, knowing that any significant change in hardware requires us to revise this decision.

Search expansions The idea of using the soft-assignment for the data, or a, has been abandoned. We simply do not think we will have the storage capacity for a-folding of the data on disk.

In the previous experiment, see Table 2.5 column 5, we see results of using query expansion, or b=3. In our experiments we evaluated every setting of b between 1 and 5. The results varied a bit between the ts size settings, and were in general more effective when ts was small and many clusters created. The difference between b=1 and b=3, as can be seen in the table, varies between 2.15 and 4.94% and between b=1 and b=5 the variance ranges from 2.31 to 5.81%. In general the first added cluster was most valuable, for the 128KB setting for example going from 1 to 2 gave an additional 2.02% while going from 2 to 3 only added another 0.74% and from 3 to 4 a meager 0.35% is gained.

Setting the neighborhood size In our search experiments we tried various settings for k, and the default setting we came to use was k=20. Setting k too high would increase the "noise"-level and result in worse search quality. For some a setting of k=20 may sound very low, but we are getting good search quality with it. Part of the reason we do not need to set k higher is because we are not using estimates of the distance between descriptors but doing the full Euclidean distance calculation.

Evaluating the impact of index depth The last major remaining factor is to evaluate the impact of index depth L. For this experiment we will use the larger 110M set. Again we refer to Section 2.5.1 for information on the hardware used. The secondary storage used is the Seagate magnetic disk described in the first row of Table 2.4.

For the indexing we use a ts of 992 descriptors. That results in 111,424 clusters on disk, each 128KB in size on average. The *in-buff* is set to 128MB so it will take 109 rounds of assignments and thus 109 intermediate files will be created

using the CF disk access policy. We then index using both a L=2 and a L=3 deep index hierarchy.

At L=2 the process takes 33,508.34 seconds or 9.31 hours. When we use the L=3 index the same process takes 11,578.35 seconds or 3.22 hours. The L=3 indexing thus takes 34.55% of the time it takes the L=2 or we could also say that L=3 is 2.89 times less work. Considering how long the L=2 indexing took we did not do the flat index of L=1.

With respect to search times and quality we could not notice a difference between the two. The set of queries for this dataset is however very small, only 533 queries, and the search quality was very high, above 98% for both settings of L. More details on this search scenario, using the L=3 index, can be found in Section 2.5.5 on the *early-halting* optimization.

The effect of *in-buff* and the number of rounds of assignment In this set of experiments we also tested various settings of the *in-buff*, ranging it from 32 to 1024MB resulting in 433 to just 14 rounds of assignments.

As expected, the TF policy did much better with a large setting, basically the larger the better. What was however surprising was that the CF policy actually did better when the *in-buff* was small. The fastest setting was also the lowest that we tried, only a 32MB *in-buff*. Increasing the size to 256MB added 8.34% to the wall clock running time of the task when running on the Seagate magnetic disk. This was because the I/O time was increased (the amount of time the CPU is idle), from 15.38% at 32MB to 17.2% with the 256MB *in-buff*.

The fact that CF does better with a smaller *in-buff* is very good news as that leaves more RAM for both the index structure and for the OS to do file buffering. There is however another consideration that must be taken into account. Depending on the size of the dataset, we may be inclined to set the *in-buff* such that we do not create too many intermediate files. If they are too many, we will have to open and close each file between reads. That would limit the OS's ability to cache for use and thus severely degrade the performance of the merge phase. The setting of *in-buff* should therefore try to compliment the size of the dataset, such that the number of intermediate files does not become too large.

2.5.4 Indexing on a wide range of storage devices

The primary goal of the following experiments is to evaluate the two disk access polices TF and CF using a wide range of secondary storage devices. For this task we use again the 110M dataset. The storage device we use are two magnetic disks, 3.5" Seagate Barracuda 7200.10 and 2.5" Fujitsu MHZ2160BJ. Both are 7200rpm disks with similar *seek time* and *rotational delay*. We also used two types of SSDs: a SuperTalent FTM28GL25H and we have two Intel X-

25M, type SSDSA2MH080G1GC. In addition to that we have a RAID5 volume connected both by DAS and NAS. A summary of all the devices can be found in Table 2.4 of Section 2.5.1. In that same section is also a description of the hardware used to do the experiments.

We are very interested in the performance of these new SSD devices as they do not carry the same penalty for random access as magnetic disks do. The solid state technology has been called the biggest breakthrough in secondary storage in decades. The magnetic disks, with their mechanical moving parts, have stagnated in performance and yet remained unchallenged as the standard in permanent storage for a very long time.

Measuring I/O time: Accurately measuring the I/O cost is quite hard and the cost will depend on the underlying hardware. In the indexing process of eCP, the main reason for an idle CPU is that it is waiting for I/O operations. We therefore use the loose definition of I/O time: as **the difference between the wall clock time and the time spent using the CPU**, on a machine that is used in isolation.
As we are only using a single core, we can use the *time* command in Linux, to derive our I/O time by:

$$I/O\ time = real\ time - user\ time$$

We must however keep in mind the limitation of our definition.

Unless otherwise specified, all experiments are conducted with the default setting of L=3, *treeA*=3, ts=992 and a 128MB *in-buff* size resulting in 109 rounds of assignments. The number of clusters created on disk is 111,424 and each cluster is on average 128KB in size. We measure the running time of our experiments using the *time* command in Linux, as it will tell us the wall clock time, called "*real*", as well as the amount of time spent using the CPU, called "*user*".

Access policies, CF vs. TF We start by analyzing the difference between the two policies. For this purpose we index the dataset on the Seagate magnetic disks using both L=2 and L=3 deep index.

Using the L=3 index, the number of distance calculations is drastically reduced. Despite this, the TF policy cannot take full advantage of the deep index using the magnetic disks. The reported *user* (CPU) time during the L=3 indexing is only 28% of the reported *real* (wall clock) time. Thus, the CPU was idle for 72% of the running time, waiting for I/O operations.

With the CF policy, the reported *user* time is 84% of the wall clock time using the same magnetic disk. The CF policy is therefore not I/O but CPU bound.

On the magnetic disks, CF can take full advantage of the deeper index structure. The running time of CF is shorter by a factor of 2.5 compared to TF and the CPU is only idle 16% of the wall clock times instead of TF's 72%.

Scaling the index-up is very effective. By incrementing the index depth L by only one, from 2 to 3, the amount of time the CPU is busy (the amount of work needed) is decreased by a factor of 2.86. I.e. the same database is constructed with a 3 level index in only 35% of the wall clock time it took using a 2 level index.

On the Fujistu magnetic disk, the 109 rounds of the 128MB *in-buff* took 60% longer than the 14 rounds of 1024MB *in-buff*. Like we said, this is no surprise. The magnetic disk does not handle random access very well and with 109 rounds and 111,424 clusters, we have to append 12.1M times to those intermediate cluster files, while with 14 round we only need to do 1.6M appends.

Changing the size of the *in-buff* did however not make a significant difference for the TF policy when we used the Intel SSD device. This is because the SSD disk has no added overhead in doing random writes. Interestingly, using an SSD can therefore reduce the algorithm's need for RAM and hence, there is no longer a need use RAM to buffer and delay random write operations with a large *in-buff*.

We also observe that the reported user times for CF are always a little lower than using TF, indicating that the CF policy is more efficient regardless of the I/O costs. In other experiments we stick to the relatively small 128MB *in-buff* because we want to simulate how TF would do at larger scale.

Using different storage devices In this set of experiments we want to evaluate the role of the secondary storage device using our default setup and our two disk access policies CF and TF.

In Figure 2.6 we have the results from running the same indexing process, using L=3 deep index and all the same setting as before but using a different storage device each time. Each bar is the full wall clock running time it took to complete the task. The white section of the bar is the reported *user* time where the CPU is busy. Next, in orange, we have the I/O time that was measured during assignment phase, that is reading the source file *in-buff* by *in-buff* as well as the cost of writing the intermediate files. The last section, in yellow, represents the I/O time measured during the merge phase that includes the reading back the intermediate files and creating the three database files.

Please note that the RAID5 and NAS experiments had to be run on different machines and thus the CPU cost is a little bit higher. One of the machines has direct access over DAS and the other uses NAS over a 1Gbps network.

The first and most obvious observation is that the CF policy performs better on all devices and is clearly a superior policy. The second observation is that the TF policy is I/O bound, or close to it, on all devices (the white bar is equal or

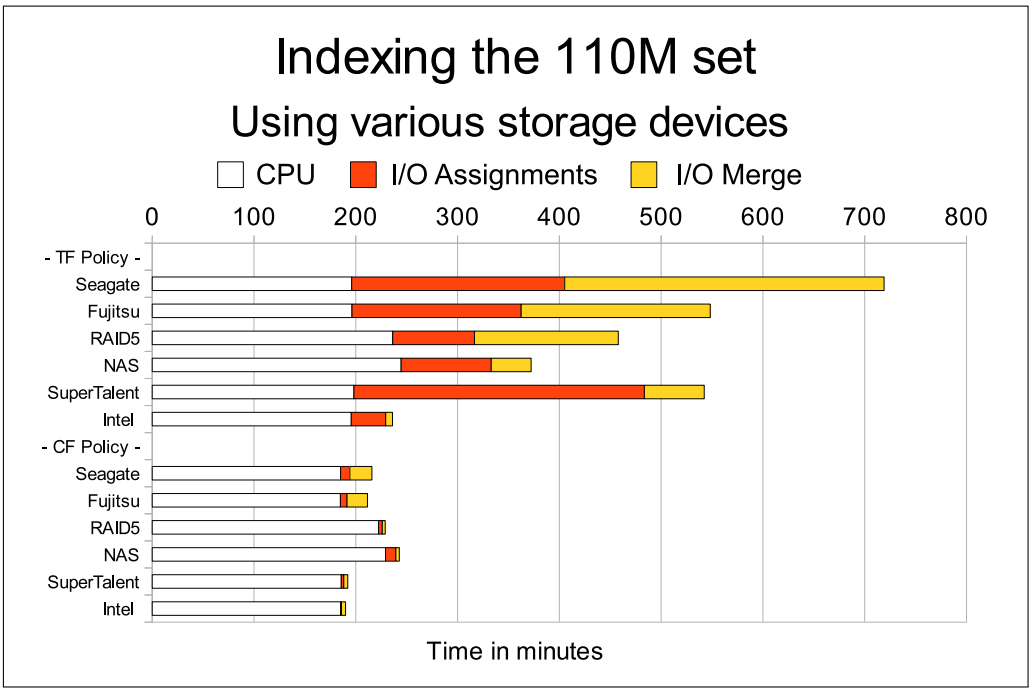

Figure 2.6: Performance of eCP index creation policies, single drive setups.

smaller than the two colored bars). We can also see that the I/O cost for TF is actually higher when running directly on our NAS server, than when another machine runs the task using the NAS over the network. Why this happens is not exactly clear. We think it may have something to do with the NAS configurations, as it is set up for maximum caching on both client and server side. We do not allow eCP to use more than 128MB of RAM for the *chunk* of data being processed, but the machine has 8GB of RAM available.

We also see that the SSD disks, especially the Intel device, are powerful and fast devices that dominate the magnetic disks. However, it is surprising how poorly the Super Talent handles the random appends in the TF policy. It actually has the worst random-write performance of all the devices and is only better than the Seagate overall, thanks to its fast read-back of those intermediate files. The lesson is that not all SSDs are guaranteed to deliver the full potential of the technology. For that, one should make sure to get one of the "good" models that typically also cost a little bit more. This technology is new and as it matures, we expect that sub-par performance of this magnitude will no longer happen.

In light of the excellent performance of the Intel SSD in the TF policy, it was the only device that was not I/O bound, we therefore also ran experiments using two devices. We put the source data and the final database on the Fujitsu

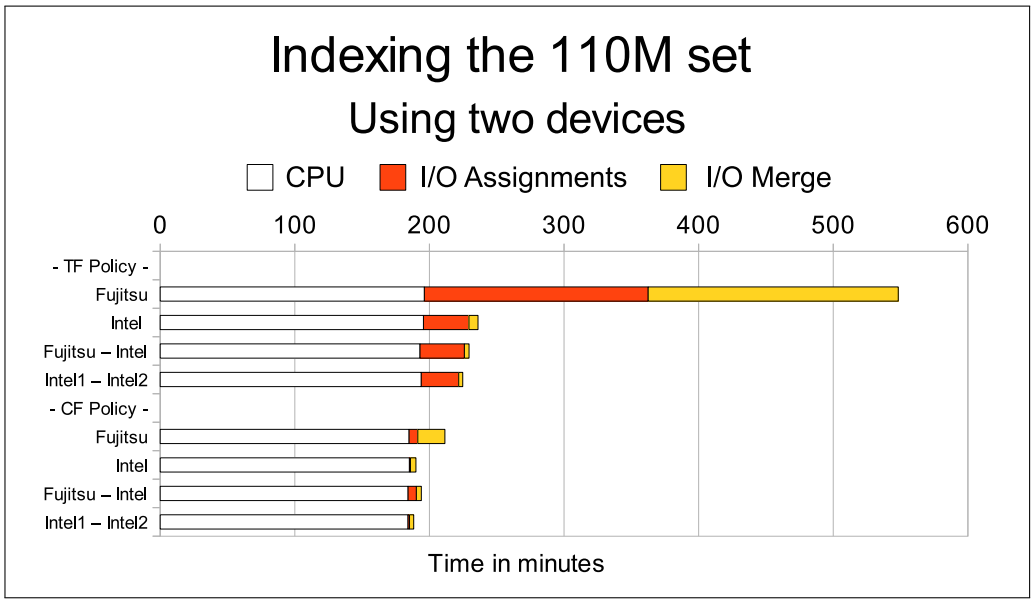

Figure 2.7: Performance of eCP index creation policies, two drive setups.

magnetic disk and used an Intel SSD for the random access pattern of hosting the intermediate files. We also performed an experiment where we use two separate Intel SSD devices.

The results of the two disk experiments can be found in Figure 2.7 in the rows labeled "Fujitsu-Intel" and "Intel1-Intel2". For comparison we include as well the experiments that used only one disk, both for the Fujitsu and the Intel SSD.

As we can see, it is not necessary to put the high-performance and pricey SSDs everywhere, using them where the random I/O occurs is enough.

2.5.5 Early termination of image level search

In the experiments with the 110M set, the main focus was on the index aspect of eCP. For this reason we only used a small set of queries to validate the indexing, 533 queries that where created with 49 StirMark modifications of 11 randomly picked images. The evaluation is the same as before, a copyright detection scenario where we include the 11 original in the dataset and use the 533 modified images as queries.

As the performance characteristics of the SSD devices is quite different than the magnetic disk we used before, we evaluated also a cluster *target-size* of 256KB and 512KB in addition to our baseline setting of 128KB *ts*. For other hardware specifications we refer to Section 2.5.1.

ts	**Indexing 110M set**		**Full search 533 queries**			**Early halting 533 queries**		
KB	mag.	SSD	mag.	SSD	P@1/R@1	mag.	SSD	P@1/R@1
512	7,099s	5,488s	4,003s	1,539s	98.31%	215s	151s	98.31%
256	9,955s	8,240s	2,847s	984s	98.12%	207s	133s	98.12%
128	13,158s	11,196s	3,164s	1,055s	98.50%	375s	249s	98.50%

Table 2.6: Balancing indexing cost, search time and search quality. The reported indexing times are for the 110M set is using the CF policy, the Fujitsu magnetic disk and the Intel SSD device. The index used is L=3 deep and we try 3 sizes of ts: 128KB, 256KB and 512KB. The search is done using no search expansion, b=1, on both disks with and without *early-halting*.

In our search experiments we only use the best performing disks of each category, the Fujitsu magnetic disk and the Intel SSD device (see table 2.4 for specifications).

The results of our experiments are summarized in Table 2.6. The database construction is reported in the 2^{nd} column, highlighted in blue. The results of the search are in columns 3 and 4: First is the full scan, highlighted in red; and second is the early halting policy as described before, highlighted in yellow.

As expected, the database construction of the small and numerous clusters takes longer, but thanks to the deep index the difference is sub-linear.

The searching has a much higher accuracy then before, up to 98.5%. The main reason for this is that the 26 StirMark variants used before are quite severe attacks and a perfect recognition is impossible, even with a full sequential scan. The 49 StirMark variants used here are not as severe and thus the task at hand is simply not as hard.

For the speed of searching, the best ts setting is is 256KB. The full scan on the magnetic disk takes on average 5.34sec. per image with that setting while with a 128KB ts it takes on average 5.94sec. and 7.51sec. with a 512KB ts. The pattern of the performance is the same when we use the SSD device, except the running times only takes about 35% of the magnetic disk times. With the optimal setting of 256KB ts, the average time per image is down to only 1.84sec.

In the last column we have the results for the *early-halting* search. As we can see, the quality does not change. This may in part be because of how high it was to begin with (high quality indicated easy task and thus many correct matches). This is the optimal scenario for our positive only halting policy.

The best performing setting is still the 256KB *target-size*. What is different are the running times, they are just a fraction of how long it took the full search for both storage types.

On margin, the magnetic disks gain the most by terminating the search early.

This is simply because they are much more I/O bound and have therefore more to gain. Using the magnetic disk and the optimal *ts* setting of 256KB, the average time per images of the full search is 5.34 seconds, but when we use *early-halting* the average time falls to only 388ms per image. On the Intel SSD, the average time per image of the full search is 1.84 seconds and with *early-halting* this time is down to only 250ms. Thus, early halting is 13.75x faster on the magnetic disk and 7.4x faster using the SSD, with no loss in recognition quality.

The performance we observe is very good and the response time is very low. There are however two factors that may be biasing our results and we should be aware of.

First is that our 533 queries may be fairly "easy" as we get a >98% correct matching rate. I.e. all of our queries do have a corresponding database image and almost all of them are correctly found during a full scan search. This is the optimal case for our simple *early halting* policy that only halts when a winner can be declared. If the queries were harder or a few of the queries did not have a matching database image, many more searches could not be terminated early and that would of course quickly raise the average processing time per image.

Second is that we are probably benefiting a lot from the file caching being done by the OS. When a query can terminate early, it will only have read a few clusters from the beginning of the datafile. When we can terminate early again and again, multiple queries in a row, the OS will have cached the beginning of the datafile for us and many of the early terminating searches read only data from the OS buffers (i.e. RAM). This explains the very good performance of the magnetic disks and why they are performing very close to the SSD. If there were queries mixed in that did not terminate early and therefore would force the search to read from the full size of the datafile, the OS's file cache would be much more stressed and the performance would most certainly degrade. For this reason, it might be a good policy to have *early-halting* eCP search do its own caching, where clusters could be pinned more to the RAM (we know that the next search will always read from the beginning of the file again, but the OS does not).

One of the main limitations of SSD devices today is their limited capacity. Therefore, when we index and search large collections, using SSDs only may not be a viable option. However, the capacity of an SSD is typically larger than the available RAM and it is much cheaper than adding more RAM. For the eCP search, especially when we do *early-halting*, it is therefore still possible to make good use a of them. The idea is to add the SSD as another layer of cache, between the RAM and the magnetic disk(s). This could be especially beneficial to an early terminating search as hopefully, it can terminate before having to access the costly magnetic disk(s).

2.5.6 Single point experiments and index quality

Another way to evaluate the indexing is to do evaluation on a per-query-descriptor basis. For that, you need a set of query descriptors with the ground truth pre-calculated. The only way to do those pre-calculations is to do a full sequential scan of the entire collection for each query descriptor in our evaluation set. For large collections this can be quite a lot of calculations.

The largest publicly available single descriptor dataset, that we know of, is the ANN_SIFT1B dataset [AJ10]. The set is comprised of 1B SIFT descriptors and 10,000 randomly selected query descriptors that have the 1,000 nearest neighbors pre-calculated and provided as ground truth.

To create an indexed database, we first pick the appropriate *ts* for the clusters. The RAID6 volume we will be using has very high-performance magnetic disks and we could probably us a fairly large *ts* (256-1024KB). But we choose to use the 128KB *ts* as it has been our default setting in most experiments and our focus is on the quality aspects, not performance.

We then calculate the number of *leaders*, i.e. clusters on disk, as follows:

$$ts = (128 * 1024bytes)/(4byteID + 128bytevector) = \ 992pts.$$

$$leaders = \frac{N}{ts} = \frac{1,000,000,000}{992} = 1,008,065$$

That is, we split the 1B SIFTs into approximately 1M clusters, each hopefully containing close to 992 descriptors. Then, at search-time, we will pick only the b most likely clusters for retrieval and scanning.

We also need to pick an index depth L and we need to choose a replication factor *treeA* to use in the index soft-assignment search expansion. We choose to test setting L to 4 and 5 and *treeA* to 1 and 3, for a total of indexing four different databases that we will designate L4A1, L4A3, L5A1 and L5A3.

As for the search, we can set b at search-time and we are therefore free to range it to any value we want. In fact we will report searches with up to b=100. To keep the quality reporting simple, we only check if the single most similar point (i.e. the nearest neighbor) is in the k-nn. As the k-nn is a ranked list, this translates to doing precision at 1 (P@1). Thus, if the nearest neighbor is in the returned k-nn we define that as a "correct match", and otherwise the search has failed (regardless of how many of the near neighbors we did return).

Our reasoning for doing this is that in the copyright detection scenario that we almost always use, there is only a single correct database descriptor that can match each query descriptor (there should be a one-to-one correspondence). And, in most cases, that single correct point is the nearest neighbor.

The initial analysis showed that we only get between 12-14% correct matches, when searching with b=1 on all four databases. When we increased clusters

searched to b=10, our correct match rate increased, but only to between 21-31% correct matches. This does not exactly look promising.

The problem may in part be due to how the query points in this dataset were selected, namely randomly. What that means is that there is no natural bound on how far away the nearest neighbor is, i.e. it may be quite far away.

What we did was to analyze the queries and the ground truth, looking at how far away the nearest neighbor is by looking up the distance distribution to the first nearest neighbor of each query using the provided ground truth.

Please note that the distances are squared (n^2) but not rooted ($\sqrt{n}$), to minimize the cost of doing the calculations.

In the pie chart of of Figure 2.8 we can see that it turns out that for 76.19% of the queries, the nearest neighbor is at a distance greater than 20,000 square distance. We also see that there are very few query points, only 3.28%, that have very near neighbors, i.e. less than 5,000.

This poses a problem for us, as this is a poor representation of the CBIR problem that we typically face. It is not surprising that we get such poor results when our deep index has to, again and again, find the correct Voronoi-cell of such dissimilar SIFT descriptors. In this case we may be haunted by the curse-of-dimensionality, like trying to use apples to find oranges.

The number of query descriptors extracted from each image, both original and attacked query images, varies greatly. In our system however, we typically only need a handful of descriptors to match the original for a correct image-level identification. This is because the "noise-level" is typically very low, even if the database is huge.

What we did was to look at a few of our images and do a full scan comparison between the original image and the 49 StirMark variants that we use as queries. The distance to the nearest match depends a lot on the attack used, but typically at least 10 descriptors have a nearest match below 20,000 distance for all 49 variants. One of the images had only 41 descriptors extracted from the original image, and even then 7 of them had a match in all 49 variants below 20,000 in distance. Many original descriptors will always be lost as some of the attacks are very severe (like cropping 75% of the image). For many of the easier variants we get a lot of query descriptors that match with a very low distance (some even exact match of distance 0).

What we choose to do with the data of the ANN_SIFT1B set, is to split the queries into distance categories of <5K, <10K, <15K, <20K and >20K. The results are in the bar chart in Figure 2.8.

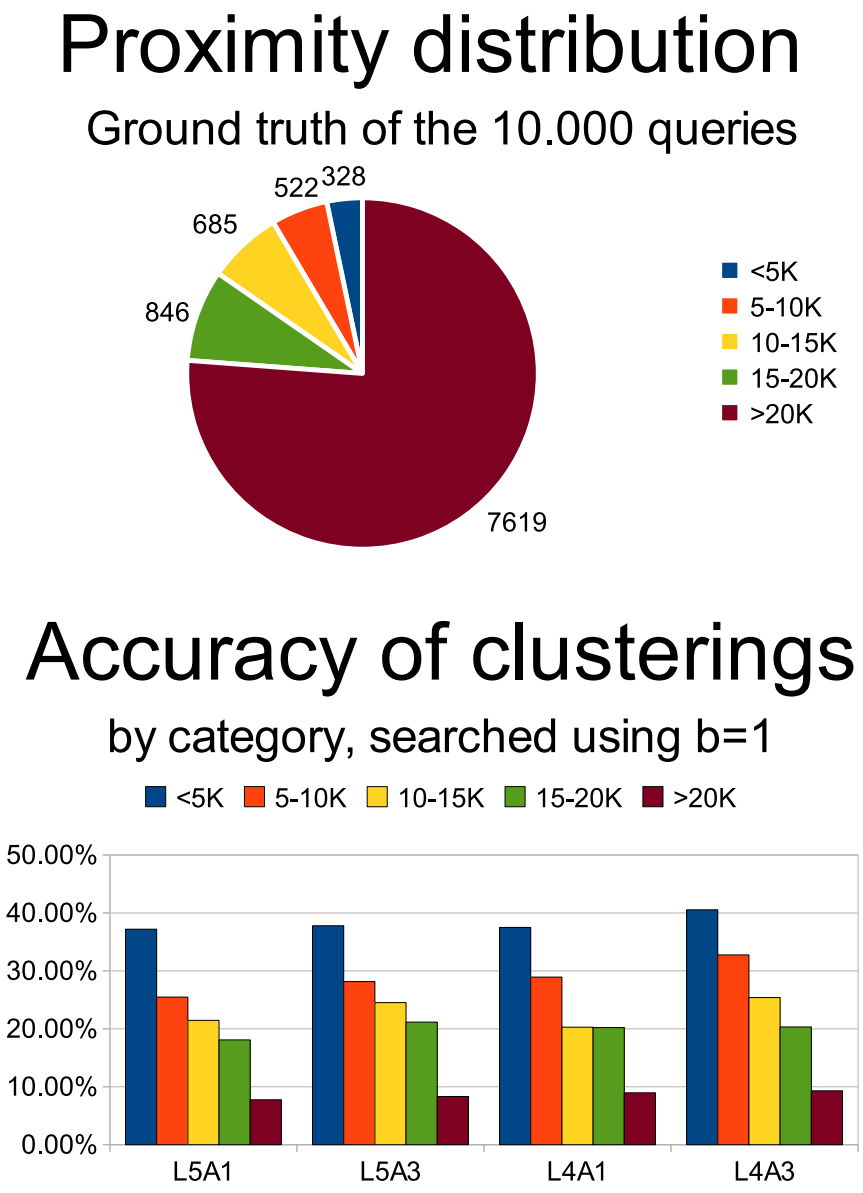

Figure 2.8: Ground truth distance analysis on the left and search quality by category on the right, where labels are the different database settings and A is short for the *treeA* value.

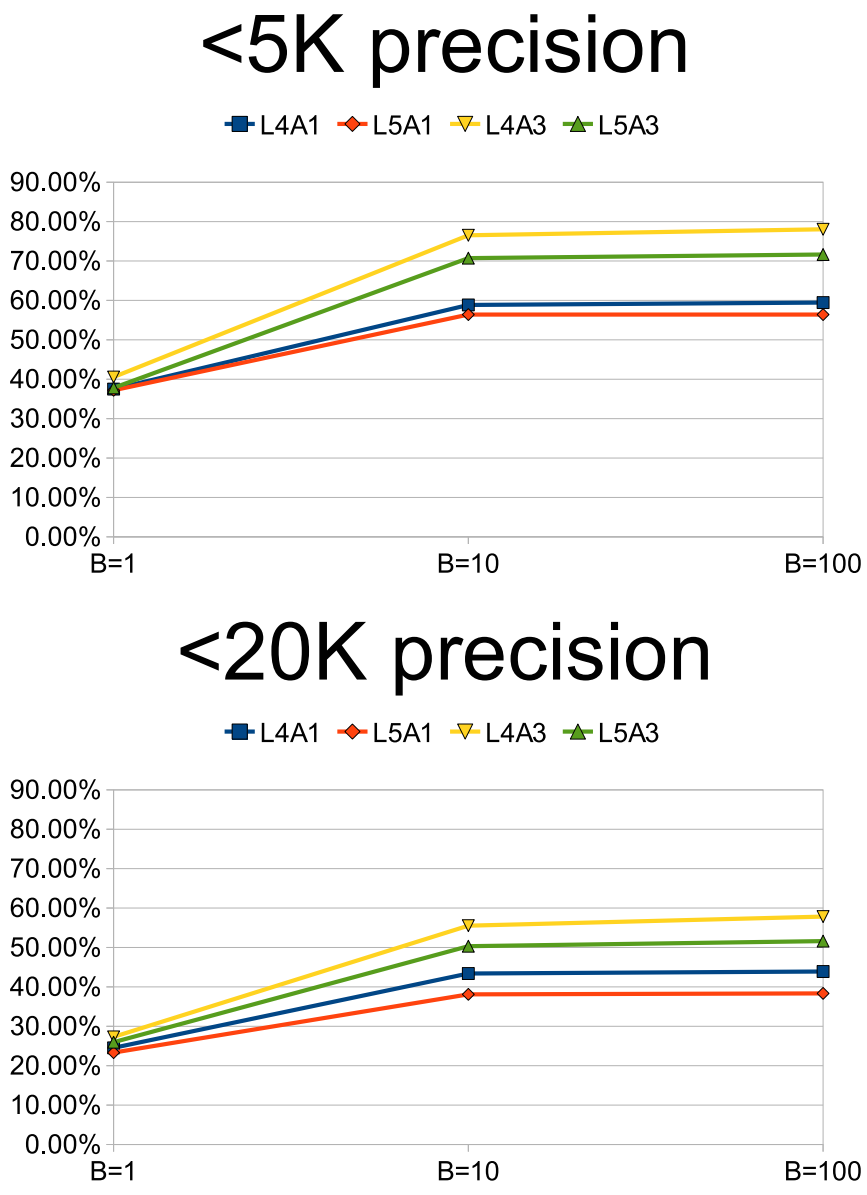

Figure 2.9: Search quality for the <5K category(left) and <20K category(right). The labels are the different database settings where A is short for the *treeA* value.

We can clearly see how the distance to the nearest neighbor matters. The by far best category is <5K with a 38-40% correct matches using b=1, regardless of *treeA*.

We create a new grouping of <20K that represents 23.81% of the queries, leaving out only the largest >20K from before. We focus on this new category, as well as the <5K, in the charts of Figure 2.9 where we show the results when searching all four databases using various values of b.

Our new category, <20K, has a precision of 24-27% at b=1 and it jumps up to 38-56% at b=10. For the <5K the jump is even higher, from around 40% up to 55-78%. The largest gain of b was in the first few clusters and very few matches are added after the 10th.

It is also notable that the setting of *treeA* matters more than the index depth L. For example, the L5A3 index (green line) does better than the shallower and non-expanded L4A1 index (blue line).

We are now in a position to discuss the topic of how to set the neighborhood size k for the search. If we retrieve and scan the "correct" cluster (i.e. the one containing the database nearest neighbor descriptor), the full scan will find the nearest neighbor, even if k=1. If on the other hand we fail to scan that "correct" cluster, k will not matter as what ever cluster we scan will not contain the true nearest neighbor. The "fault" has already happened in the index traversal (see Section 2.3.4 for description of the three ways the search can fail).

Now for a CBIR system, and especially in the copyright detection scenario, the most similar descriptor is most commonly also the "correct match".[6] We are only inclined to increase k for the heavily distorted queries, but those are also the ones that are most likely to be scanning the "incorrect" cluster. For such cases, increasing b might be a better (but costly) option. In fact, we can observe this in Figure 2.9 where increasing b to more than 1 gives better results, even for very low distances (the upper graph is for <5,000).

In the course of our image-level search experiments, we did test various values of k. For the easy StirMark variants, the value can be set very low, 1-5 was giving good results. But for the harder variants, where distortions are more severe, we need to allow a little bit larger k. Our conclusion was that for our vote aggregation policy of one-vote-per-neighbor in k, the size should be set between 10 and 30 and our default setting is k=20.

We have another reason for wanting to keep k low. In our distance calculations we use the current farthest distance to early terminate each distance evaluation during the scan of a cluster. If k is large, that means true distance to the k^{th} most similar descriptor is high and that reduces the pruning power of this optimization.

6. For example, the weighted voting policy relies heavily on that low proximity is strongly correlated with a correct match.

Finally, as we shall see when we use the set of small images (only 150px on wider edge), the main reason for keeping k low is to keep down the "noise-level" created by false-positive votes. When only a handful of query descriptors are extracted from the query image, the correct database image is often found with even fewer correct matches. There is therefore no room for a high "noise-level", as a handful of correct matches can easily be drowned by the random "noise".

2.6 Discussion

By now we have established that our extended Cluster Pruning, or eCP, is actually two algorithms: indexing and searching. We should also know that the two have very different operational characteristics.

Summary The database construction is a long laborious process of assigning the entire collection of descriptors, each to it's most similar cluster. Using the top-down index and the CF policy the CPU bound process can benefit greatly from a deep index hierarchy L.

For quality, the use of search expansions, b and $treeA$, play a more significant role. Looking again at Figure 2.9; We see that while index dept L has some influence, the search expansions $treeA$ and b matter more.

The performance benefit of a deep index L for search is minimal, but so far the increased depth has not caused a significant loss in quality either. Doing all the index traversals are typically only a fraction of the total CPU overhead, ts tends dominate the cost.

However, the true performance limitation of the search is that it is I/O bound on retrieving all the clusters in random order. We can see this in Table 2.6, column 3. The average time per image using the optimal 256KB ts setting drops from 5.3 seconds per image on magnetic disks to 1.8 seconds using the Intel SSD. If the search was CPU bound, the better performance of the underlying hardware should not matter at all. We showed that using both better hardware, like the SSDs, and using an *early-halting* policy (see column 4 of the same table), the task can still be solved within a reasonable response time.

The eCP algorithm is simple and yet both flexible and extendable. We see this both in the extension we proposed, described and evaluated and we will shortly give further evidence for this by showing its application to other domains and how it can be easily altered to mimic the behavior of other algorithms, like BoF or the NV-Tree.

It should also be clear that keeping the all the vector information in the clusters and doing Euclidean distances calculations in the scanning process is a choice, not a necessity. Using Euclidean distance makes eCP easier to understand

and to monitor what is going on. For example, tracking the effects of a particular attack on an image can be done easily. We can track exactly the distance of each query descriptors distortion and analyze the effects of that distortion on our algorithms behavior. On the index level we can trace the *paths* of descriptors through the index and even quantify how likely one descriptor is to stray from the path.[7] And on the cluster level, we have perfectly ranked k-nns based on the actual distance between descriptors, regardless how many clusters are scanned.

eCP configuration In setting eCPs parameters, the trickiest is the *target-size* (*ts*) of clusters on disk. It represents a complicated balancing act between indexing time, search time and search quality that also hinges on the performance of the underlying hardware.

In our experiments so far, we find that a *ts* of 128KB-256KB has the best trade-off.

The aggressiveness of index depth L we can use is only limited by the need to leave enough clusters on the top-level of the index. Otherwise we risk poor initial discrimination and an early branching mistake is a lot of wasted effort.

As for search expansions, we find that $treeA$=3 is a good default setting that is worth the added overhead. The flexibility of being able to set b at search-time is also a great advantage.

Usefulness in other domains eCP is clearly not domain specific, we are ourselves adapting it from the CP algorithm that was designed for text corpora. It will actually work on any high-dimensional vectorial data.

We have already show how it can do single point experiments and we have also converted it to index and search 36-dimensional float-based audio features. A conversation that took less then one day.

In addition, like the M-tree family of algortihms [CPZ97, TTSF00, SPS04], eCP can be used to index and search data in Metric space because we abandoned the use of calculated *leaders* like centroid or mediod.

eCP, prototypical CBIR algorithm As we work with eCP more and more we have come to appreciate how prototypical this simple algorithm is. With fairly minor changes we can imitate the behavior of other algorithms.

eCP could for example be used as the quantizer for the Bag-of-Words algorithm. Or we even keep the eCP system and just skip the full scanning of clusters, returning just all the imageID's assigned to it instead.

7. We do this by comparing the distance between the "correct" branch and the most similar "wrong" path.

eCP can also be made to imitate the behavior of the NV-tree. The simplest way would be to store in the cluster a tuple of the imageID and the distance to the cluster *leader* instead of the full descriptor. By retaining only the imageIDs and the distance we have almost the same data as the NV-Tree stores in its segments. At search time the queries distance-to-*leader* is used to find the top k ImageIDs with most similar distance-to-*leader*, almost exactly like the NV-Tree does with its distance to last projection.

The authors of NV-tree realized that this is a very rough distance estimation and thus k had to be large. They propose to improve quality by using 3 indexes and OMedRank [FKS03] the results, i.e. two of the three indexes have to agree before adding the imageID to k.

Instead of having three eCP indices, we can simply use both a=3 for data on disk, and expand the search to b=3 clusters at search-time. This will cause data to overlap where the queries are most similar and that is exactly what we want. We can then use OMedRank, just like the NV-Tree, to filter the most similar imageIDs into the top k results.

Some experimental implementations have been done that keep distance values instead of the full descriptor vector (in the spirit of what we described above) but we never took it far enough to have a full discussion or experimental results on this topic. It remains on our list of unexplored avenues of research.

The bottom line is that eCP is simple and yet flexible. It is by choice that we keep all the data and do all the hard work, solving what could be called a "worst case scenario".

Scaling-Up will require Scaling-Out: Our long term goal is to do CBIR at a very large scale, using datasets with billions of SIFT descriptors, representing tens or hundreds of millions of images.

So far, the deepest index has been L=5, used on the largest dataset of 1 billion descriptors (the ANN_SIFT1B set). Even with a dataset of that size, L=5 only left 16 *leaders* on the top-level. There is thus a huge potential in applying eCP to massive datasets, even without the need to increase the index depth much above what we have already tested.

Even with the deep index hierarchy, our goal of indexing 100M images (or more) is a formidable task. The storage capacity for terabytes of data and computational power necessary to index such large collections is a job for more than a single CPU. Parallel and even distributed computing are necessary tools to incorporate in a modern day CBIR system.

Chapter 3

Scalable CBIR, Parallelism

In this chapter we will develop the eCP algorithm to harness the power of large powerful multi-core servers. We will be doing this both for the CPU intensive indexing as well as for the search.

We will start the chapter with a background section where we discuss relevant hardware topics, like multi-core CPUs, SMT and the memory hierarchy.

Multi-threading the indexing process makes a lot of sense due to the fact that it is a CPU bound process. In the second section we will develop and discuss a parallelized indexing algorithm, where we focus on multi-threading the CPU intensive assignment phase and evaluate our algorithm on a powerful multi-core server on 1TB of data.

The third section is devoted to the eCP search. To spend effort on parallelizing an I/O bound search process may sound a bit strange at first, but multi-threading the search is not the only change to the algorithm that we make. We will develop and discuss in detail a bulk loading eCP search algorithm that will search up to 100,000 images in a single batch. Large batches will shift the search from being I/O bound to CPU bound and that is where the multi-threaded search becomes valuable.

In the fourth and last section we summarize the chapter and draw conclusions.

3.1 Background

In this section we will look at some of the key evolutions in hardware that influence the performance of the parallelized eCP algorithm that we will be developing in this chapter.

3.1.1 Multi-core CPUs

For a long time, the mainstream CPU development was focused on making CPUs faster and faster. Around the arrival of the new millennium, CPU vendors could not get over the 4GHz barrier without overheating problems. They instead reached more computing power by multiplying the number of processing units. Multi-core CPUs were not new as they have been around for a long time in specialized systems. But around 2005, multi-core architectures hit the mass market and by now, every new laptop, desktop and even some mobile phones come equipped with many cores by default.

Today, a standard desktop has 2–4 cores, and a single server can have 1–4 CPUs, where each CPU has 4–24 cores.

3.1.2 Simultaneous multi-threading, SMT

The basic idea behind SMT, also well known under Intel's brand name of Hyper-Threading Technology HT or HTT, is to get better utilization of each core by scheduling multiple tasks at the same time at the level of the core (i.e. this is done regardless of what the OS is doing with regards to multi-tasking). If the core task currently running is blocked, another task can be run, utilizing the otherwise wasted cycles. For each real physical processing core in the CPU, the operating system will see and address two (or more) logical cores. The OS then schedules work on both cores and the CPU will try its best to share the physical processor between the two workloads.

The increased productivity will depend a lot on the algorithms running and their access pattern to data. CPU intensive tasks that require little data and have predictable access patterns will not leave any slack for other tasks to pick up. Data intensive tasks that frequently block during long periods of time, waiting for data to be brought into the CPU cache from RAM or disk, will create a lot of slack and the other tasks will get a lot of work done.

SMT has been around for several years but since 2002, when Intel introduced Hyper-Threading in their CPUs, SMT moved from the realm of specialized systems into the mainstream. In the early days, HT was considered not very useful and frequently disabled on most systems. [1] This was because the cost of doing the context switching was too high. In cases where switching was frequent, little gain was to be had and even, in the worst case, overall performance could degrade.

In latest processors, the switching is done at a very low level, inside the CPU cores, greatly reducing the effort involved and making SMT much more useful. In today's implementation of Intel's Hyper-Threading, the CPU keeps track of two

1. SMT is a functionality of the CPU and can typically be enabled or disabled in the computer's BIOS settings.

contexts (tasks) and the OS therefore sees two logical cores for each real physical core of the CPU (e.g. a 6 core machine will make the OS report 12 cores).

3.1.3 The memory hierarchy and caching

As the CPUs got faster and faster, the memory modules of the RAM could not keep up and a gap was created between the clock speed of the memory and the CPU. To bridge the gap, the CPU tries to predict what data will be needed and use faster but smaller memory modules to cache the needed data.

Various types of memory modules are in use today and typically there is a memory hierarchy bridging the performance gap between CPU and RAM, where very fast but small modules are at the top and as you go down the levels of caching, the modules get ever larger but slower. The smallest and fastest cache is level one (L1), typically 64KB, and is embedded with the core. Next is level 2 (L2), which can either be exclusive or partially shared between cores that are on the same die. The size per core can be 256–1024KB but, when shared, the dedicated part is small while the total size is 2–6MB. The most recent level is level three (L3) cache. It's large, 1-20MB and is typically shared between all cores of the CPU.

The search for data starts at the top (L1) and if the data is not found there, lower levels are probed at ever greater cost. When data on one level is not found, it is called a "miss". It is therefore common to see reported the % of cache misses per instruction or per 100 instructions executed.

It is possible to give some perspective on the costs of accessing the various levels of the memory hierarchy. As the cache modules are often inside the CPU chips, or very close to the cores, the access time is counted in the number of cycles that pass (i.e. number of times the CPU could have done computations). Accessing the L1 cache takes 4 cycles of the CPU. Accessing the L2 cache takes 3 times that (12 cycles) and for L3 cache it takes 9 to 11 times longer (36 to 46 cycles). The access to RAM is measured in nanoseconds and, depending on the memory-bus, it typically takes between 65-200ns. to access it. The faster the CPU clocking speed, the more cycles will have elapsed during the wait for the data. If we assume 2.66GHz CPU than the 65-200ns. amount to between 172 and 532 cycles, or 43 to 133 time more than accessing the L1 cache. Also, when the memory controller makes the request to load the data from RAM, it will already have paid the 46 cycle penalty for checking the cache levels.

Since we are on the topic, accessing the data on disk is measured in milliseconds, where magnetic disks take between 6ms. and 20ms. (*seek time* + *rotational-delay*) to read a disk page. That would amount to between 15,960 and 53,200 idle CPU cycles.

3.1.4 High performance computing and access patterns

The performance of an application will depend on the nature of its task, how it is implemented, the hardware used and the access patterns to data. High-performance computing (HPC) is typically made of complicated iterative/recursive computations where a very large number of computations are needed to solve the task at hand, but relatively small amount of data. Once the computation starts, the calculations are typically very predictable. This kind of applications and access pattern are the best case scenario for the memory hierarchy described above. Most of the data will be accessed from high levels of the cache (L1 or L2) and therefore rarely block for long periods of time. Thus, the CPU(s) are kept constantly busy, even without SMT technology, as the cache misses should be rare.

For the eCP algorithm this is not at all the case, neither for the indexing nor the search. The access pattern for the search is to read a lot of data from disk for a very small amount of query descriptor. This results in an I/O bound algorithm that spends most of the time waiting for the data to arrive from disk. The indexing is CPU intensive and with the CF policy it is CPU bound. However, it is also very data intensive, as the amount of computations for each descriptor is relatively light (depending on how deep we set the index depth L). Thus, the nature of eCP's indexing and search algorithms and their access patterns do not fit a HPC application profile. What we will be looking for is wasted CPU cycles, high frequency of cache misses and what can be done about it.

3.2 Parallelized eCP indexing algorithm

In this section we will parallelize the eCP indexing algorithm. We start with a brief overview of how this is accomplished before we go into details on the implementations in the following subsection. Then we will show and discuss our experimental results and we end this section with a short summary of the topic.

The searching of the indexed databases will be discussed in Section 3.3.

3.2.1 Parallel indexing: Overview

We should start by noting that we only consider using the CF disk access policy for parallelized indexing. As we saw in the last chapter, the eCP database construction can be split into three phases:

Phase #1: Index creation. During this phase, cluster representatives are picked from the collection and organized in an in-memory L deep hierarchy.

Phase #2: Assignments. Descriptors are assigned to clusters in rounds of executions, each round creating a sorted *chunk*-sized temporary file.

Phase #3: Merging. This phase creates the database, where the main task is to merge all the sorted temporary files into an indexed data file.

Of the three, phase #2 is a particularly good candidate for parallelization as that is where most of the CPU work is done. In each round of assignments, a

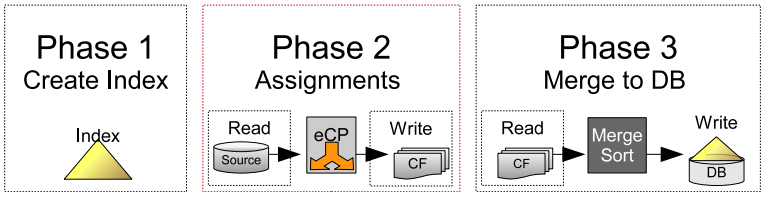

Figure 3.1: The 3 phases of eCP: #1 Index creation, #2 Assignments and #3 Merge

large chunk of data is read from disk, assigned to the most similar clusters using the L deep index (CPU intensive) and written back to disk as an intermediate temporary file.

In contrast to phase #2, phase #3 is almost solely I/O intensive as merging requires few and cheap calculations. The I/O cost could be substantial but using buffered reading will minimize that cost as it greatly reduces the number of issued random I/Os. For I/O intensive tasks, an I/O parallelization (using multiple threads to read/write to different storage devices in parallel) is possible, but this requires complicated I/O management and will have very limited scalability due to restrictions on the number of devices possible per machine. A much more common option (and the one we will use) is to arrange the disks available in a disk volume using RAID, controlled with a dedicated RAID controller.

Parallelizing phase #1 is not feasible for three reasons. The first is that this phase is only a fraction of the overall cost. Secondly, the main bottleneck is not CPU but I/O as the picked *leaders* have to be read from disk. Thirdly reason is the order dependence of the top-down construction as it makes parallelization complicated. Picking the same *leaders* in a different order would result in a very different hierarchy and we desire the index construction to be reproducible.

The key to fast parallel descriptor assignments in phase #2 is to make the index hierarchy a read only structure. This reduces the memory requirements as all the threads can share the same index structure and a read only structure does not need the costly synchronization mechanisms like semaphores and blocking code segments. The bookkeeping necessary to create the indexed database, which was previously done during the assignments, is postponed until phase #3, when we read all the data back for the final merge.

3.2.2 Parallel indexing: Implementation details

We will start by discussing the structure of the index hierarchy, how we build it and how the design can create unnecessary pressure on the cache memory and therefore degrade performance. We then turn to how we make use of multiple threads in the assignment process.

Index structure implementation When indexing large a collection, the eCP index created will be very large. The by far largest factor is the number of *cluster representatives* created, or the *leaders* created on the bottom level of the index. In theory, every level of the eCP index hierarchy could be sampled independently (randomly picked descriptors). To keep the memory requirements down however, we use only descriptors from the bottom level as then the upper levels can share the memory for the descriptor by way of pointers. I.e. we initially pick all the bottom-level *leaders* that will become *cluster representatives* and store them in an array. We then pick subsets from the array for the upper levels of the index and then we start building the hierarchy, top-down, using pointers to the array instead of allocating more memory for each descriptor.

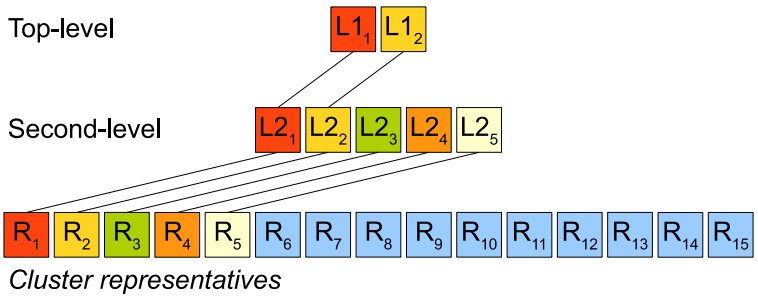

Figure 3.2: Here we see the start of the index hierarchy construction. The bottom level of *cluster representatives* is first selected, labeled R. Then the upper layers are created, labeled L2 and L1. The links between layers show assignments, and here only the automatic "self assignments" have been done. The colors indicate that the same memory segment is used, i.e. $L1_1$ and $L2_1$ are both pointers to R_1. Note that the blue R_6-R_{15} each have their own memory segment as well.

In Figure 3.2 we see the beginning of building an $L = 3$ deep index. First, labeled R, the bottom level *cluster representatives* are randomly picked and arranged in an array. Then, the upper levels of the index are created. Edges indicate assignments and the next step is to finish building the index structure by doing them (see Section 2.3.4 in Chapter 2 for full details on eCP's top-down index

construction).[2] We do however see some edges already, those are the automatic "self assignments" we do when we pick the *leaders* for each level, assigning the leader below to "it self" on the level above.

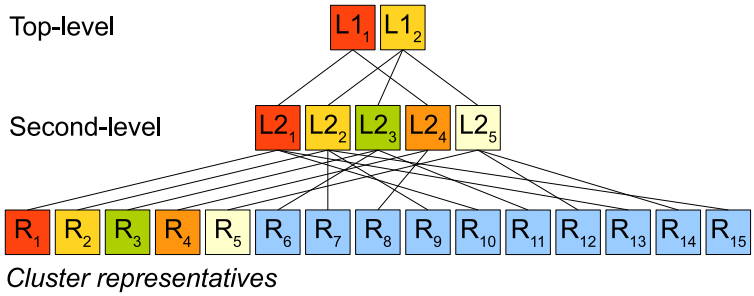

Figure 3.3: Here we see the completed index hierarchy construction that was started in Figure 3.2.

In Figure 3.3 we can see the completed index structure with all the assignment edges added. To this index, the traversal start by scanning the top layer, $L1_1$ and $L1_2$ that happen to be next to each other in the array. If the branch of $L1_1$ more proximate we next scan $L2_1$ and $L2_5$, jumping over 3 descriptors. We branch once more, following the $L2_1$ branch we now scan R_1, R_{10} and R_{13} and essentially we are jumping all over the array.

When the index structure is small, such that it can fit in the processors L2 og L3 cache, this jumping about the array does not matter. But if the jumping around is causing cache misses the performance will degrade.

In Figure 3.4 we have the same index structure but this time each sub-branch is allocated its own memory block and the descriptors are full copies (not pointers). Each color represents a block of memory. When we traverse this structure, there is no big array to jump around in and there should therefore be less pressure on the CPUs cache. The down side is that assigning each branch its own memory block requires more memory. Also, using $treeA > 1$ means a copy must be created for each assignment at each level.

Also, there may still be significant pressure on the CPU cache as each assignment is fairly quick, and each descriptor takes a unique path through the index and thus there will be a lot of jumping between sub-branch memory blocks anyway. Indexing those descriptors that have similar paths would decrease the cache pressure, but we have no way of knowing the likely path for each descriptor

2. We complete first the second level by assigning $L2_3$,$L2_4$ and $L2_5$ to the most proximate *leader* on the top-level, and then we complete the bottom level by assigning the R_6-R_{15}, searching the already constructed upper levels .

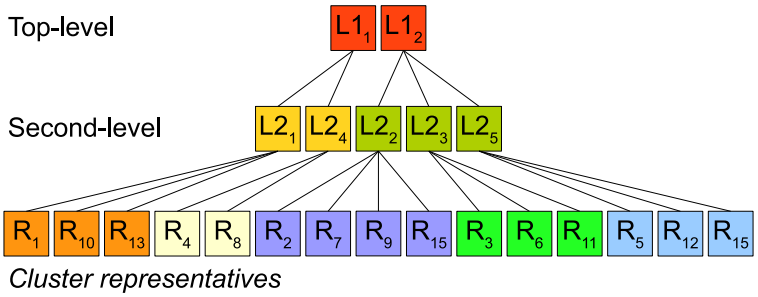

Figure 3.4: Here we have a completed index hierarchy that allocates memory for each sub-branch of the hierarchy. Each colored area indicates an aligned block of memory. *Leaders* on each level, under each sub-branch, are grouped together (re-ordered) using multiple copies instead of pointers. This way, random access is minimized during index traversal and less pressure is created on the CPU cache.

a priori. To know that would require grouping them by similarity, but that is precisely the task of our indexing.

We made implementations of both versions, but our main body of experiments is based on the version that uses a single array and pointers.

Multi-threading Once the index structure is made read-only, the threading of the assignments is fairly straight forward. What we did was to allow the number of threads used for assignments to be set as a runtime parameter. We create a thread-pool for the assignment threads, and then, in each round, we simply divide the number of descriptors in the *in-buff* evenly between all the threads and run the assignments. Once every thread is done, a single thread handles the flushing and filling of *in-buff* between rounds.

This works fine while the threads do not exceed the number of real cores, but when they do, we get a problem.

The threads running on the second logical core (i.e. a HT core) take longer to finish their equal size task. If we run 7 threads on a 6 real core machine, one core will have two logical cores active. By the time the thread on the high-priority logical core is done, the second (the 7^{th} thread) still has work to do and is bumped to first priority. But, by the time this happens, the other 5 real cores have finished their tasks and are just idle.

Thus, if the work is divided evenly between threads, and one (or a few) of those threads run on the Hyper-Threading second logical core, the task may actually take longer than running it with fewer threads on real cores only. The problem is not the Hyper-Threading, but our assumption of dividing the work evenly. What is actually recommended is to create many more small tasks and allow some cores

	Images	Image size		Dataset size	
Set	in set	wide edge	descriptors	descriptors	on disk
110M	100K	512px.	1100	110M	13.6GB
8.1B	25M	150px	300	8.1B	1TB

Table 3.1: The two SIFT-based data sets used in this round of experiments.

(real) to complete more task while others (HT) complete fewer.

We simply changed the implementation such that a thread is assigned a small batch of descriptors and then it runs. If there is more work to be done when the thread completes its batch, it is assigned another, and so on, until all the work is done.

3.2.3 Parallel indexing: Experiments and results

The experiments are run on a Dell r710 rack server with two Intel X5650 2.67GHz CPUs. Each CPU has 6 real cores and each core has 256KB of L2 cache and the 6 cores share a 12MB L3 cache. With Hyper-Threading enabled, the number of cores that the OS "sees" is doubled, to 24 (12 real and 12 HT). The RAM consists of 18x8GB 800Mhz RDIMM chips for a total of 144GB of RAM. This RAM is very large, but relatively slow (200ns access time) compared to modern standards. For secondary storage we have a Dell PowerVault MD 1200 with 12x 15k-rpm, 600GB SAS disks organized in a single RAID6 configuration by a Dell PERC H800 controller for a total of 5.7TB usable disk space.

Note that most, but not all, experiments are run in isolation, i.e. with no other user on the machine. We will specify clearly when experiments are not run in isolation.

In our experiments we use two image collections, the 110M set that we have seen before as well as a collection that has 8.1B descriptors (approximately 25M images) and requires about 1TB of disk space. This is a subset of a larger 30.2B collection (100 million images) that is the ultimate goal to index and search. For now, we limit ourselves to a 1TB subset, as the storage capacity and scalability of our hardware is limited. See Table 3.1 for more details.

Part of the evaluation of the database construction is to search the database. We will however postpone that part of our discussion until Section 3.3 where we focus on multi-threaded batch search.

Evaluating scalability The goal of the first experiment is to determine the scalability when using an increasing number of cores. For this experiment we index the 110 million SIFT collection, always using the same configuration of

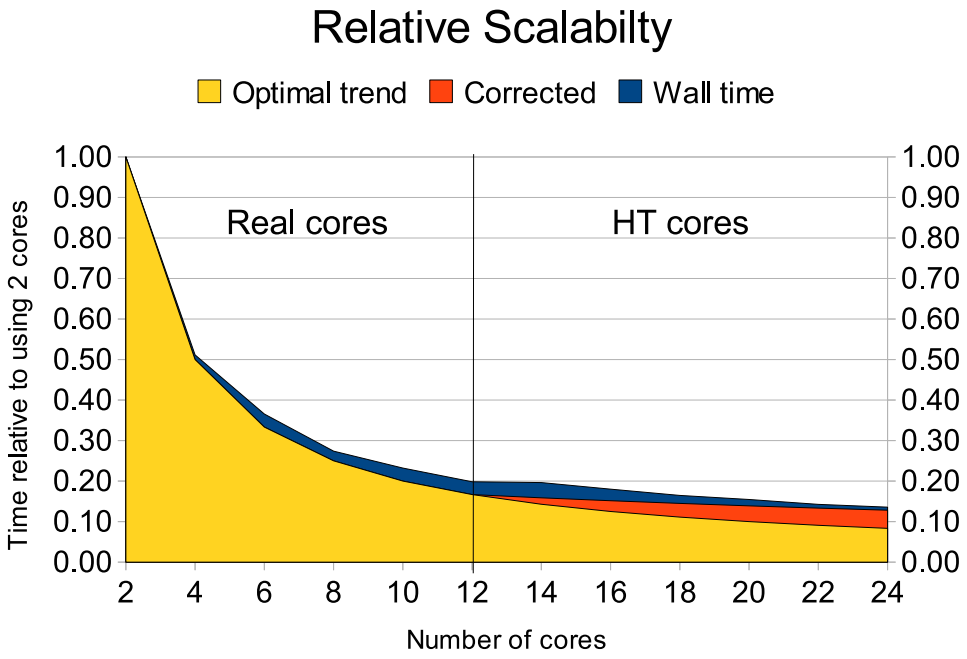

Figure 3.5: Relative response time of parallelized eCP indexing using 2 to 24 virtual cores.

$L = 3$, *treeA*= 3 and creating 111,424 clusters on disk. The index created is only 14.5MB and almost all of it can fit in the 12MB L3 cache. Since our machine has 24 logical cores (12 real and 12 HT), we split the experiment into two parts. In the first part, we disable the Hyper-Threading and can thus only use the 12 real cores. We enable HT in the second part and use all 24 virtual cores.

The results for both parts can be seen in Figures 3.5 and 3.6. In both the graphs, drawn in blue, we have the wall clock running time but in Figure 3.5 it is relative, using 2-core setting as the reference point. In Figure 3.5 we have also, in yellow, the optimal scale-up trend line (i.e. all cores do the same amount of work). As we start using the Hyper-Threading (i.e. more than 12 threads with HT enabled), we know that such threads cannot realistically be expected to do the same amount of work. Intel's material about HT says that a maximum of 30% additional work should be expected of the HT-cores if the application is data intensive. Therefore, to give a reference to how well our parallel eCP indexing scales, we show two trend lines. In yellow we have an optimal trend line that assumes all cores are equally powerful and in orange we have a "corrected" trend line that only expects the HT-cores to accomplish 30% of the work of a real core.

In addition to the wall clock time in Figure 3.6, we have also the reported user time (the cumulative amount of time the CPUs are reported as busy) shown

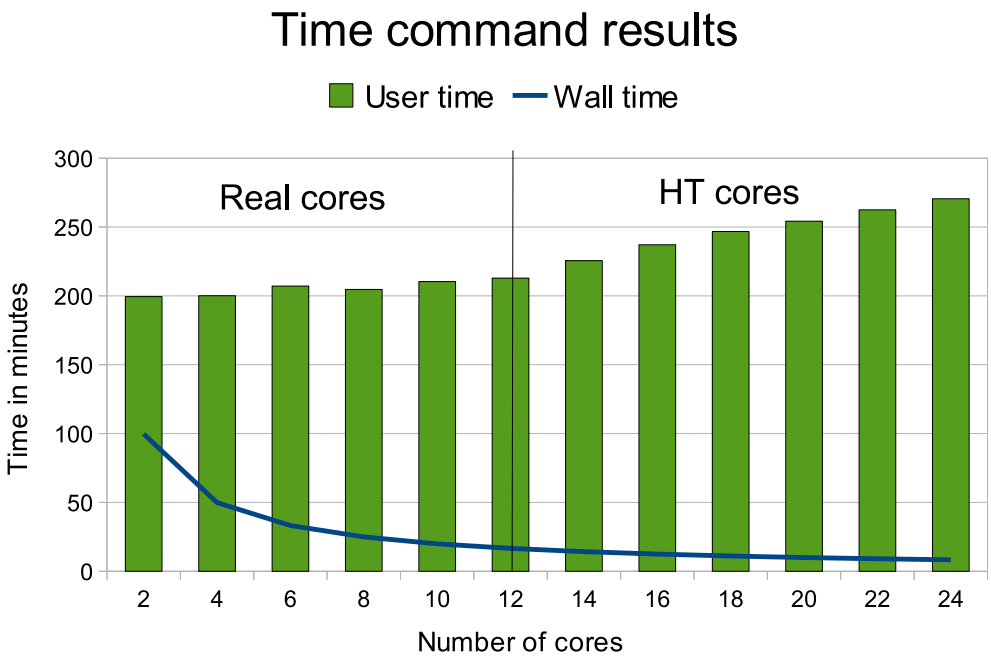

Figure 3.6: In this graph we see the wall clock (Real) time in blue and the CPU (User) time in green, as reported by the Linux *time* command. The blue line (wall clock time) in this graph corresponds to the blue line (relative wall clock time) in the graph in Figure 3.5.

in green.

Our first observation from Figure 3.5 is that overall our algorithm follows the trend quite closely, indicating that the parallelized eCP indexing scales quite well. We see this also in Figure 3.6, as the reported user time for the first 12 cores is relatively stable around 200 minutes of CPU time. As we move to the right on the x-axis, toward 24 cores, we see that the amount of reported work rises sharply when we use the HT cores. This is because the OS does not distinguish between virtual cores, whether they are real or HT. Therefore, while the HT thread is busy, doing only ≈30% of the work, it is reported as 100% busy in units of time that the *time* command is accumulating as the user time. This also happens in other popular commands, for example, if we use *top* on a HT enabled machine. The reported CPU load % is based on that all cores are equal. Thus, when *top* reports 50% load it actually is much closer to 70-80%.

The second observation is that as we move right on the x-axis of Figure 3.5, toward using all 24 cores, we can see that the gap between the blue and orange lines is getting very narrow. This would indicate that the HT cores are actually doing more work than the expected (orange line) 30%. That also means that our

code must be blocking, such that the second thread has a chance to get its work done. This is exactly what the HT is designed for, to use otherwise wasted CPU cycles.

The implementation shown, in both the graphs, is the one where we divided the work evenly between all threads. Therefore, it actually takes longer to index with 14 threads, using 12 real and 2 HT cores, than it took using only 12 threads, all real cores. We see this in Figure 3.5 as a small "bump" in the relative running time around the 14 core setting on the x-axis. As we already discussed in the previous section on implementation details, this is not a problem of HT, but rather our division of labor between the threads. When we divide the work between threads into small loads, this bump goes away and follows the trend line quite closely again.

Large scale indexing and monitoring cache pressure The goal of this experiment is to index the 8.1B collection as well as to monitor the performance of the indexing when the size of the index is much larger than the CPU cache. We are particularly interested in finding and assessing the potential bottleneck(s) in the pointer based index structure.

To do this we need to monitor low-level information during the execution of the eCP indexing. For this purpose, we use the tool *TipTop* [Roh11], developed by Erven Rohou. *TipTop* was developed to extend standard monitoring tools (like *top*) especially to facilitate the monitoring of parallelized applications and it enables us to monitor several things like IPC (instructions per cycle), cache misses for both level 2 and level 3 cache, etc.

We focus on evaluating our default index structure when using a single array and pointers, described in Section 3.2.2. The issue we would like to monitor is how much pressure is created on the cache in traversing the index, when the sub-branches are pointing to descriptors all over the bottom-level array of *cluster representatives* (see Figure 3.3 in Section 3.2.2).

We build 4 index structures for the collection. Two of them are realistic settings for later search, while the other two are used more as a reference for the first two. The 162K and the 16K settings have a huge ts, 50K and 500K descriptors respectively, that results in a narrow index but clusters that are too large to make the search viable. We use the small index size as a reference for evaluating cache pressure, while the large ts can be used as a reference for search quality.

The first viable index is the 8.1M, that has 8.1 million clusters on disk, a ts of 1,000 descriptors and a L=5 deep index structure. The other viable index is the 1.6M, that has 1.6 million clusters on disk, a ts of 5,000 descriptors and a L=4 deep index structure. See columns 1-4 in Table 3.2 for more details on all 4 index structures.

ts	Clusters on disk	*L*	Index size	Traversal #Dist. calc.	Indexing time(h)	IPC	Cache miss% L2	Cache miss% L3
500K	16K	2	2MB	-	-	1.25	0.9	0.0
50K	162K	3	22MB	382	33.05	0.88	0.8	0.2
5K	1.6M	4	212MB	357	40.06	0.73	0.9	0.4
1K	8.1M	5	1.1GB	313	42.71	0.68	0.8	0.4

Table 3.2: Columns 1-4 are information on the index structure. Column 5 shows the estimated indexing time using 12 cores, in isolation. Columns 6-8 are the results measured with *TipTop*: Instructions per cycle and the level 2 and level 3 cache misses per cycle.

Clearly, the pressure on randomly accessing the descriptor array of the 8.1M index is going to be substantial. It is by far the largest structure and it requires over 1GB of RAM. For the 1.6M index of 212MB, only 5.6% of the index will fit in the 12MB L3 cache. The two smallest index structures, 162K and 16K, require 22MB and only 2MB of RAM respectively. About half of the 162K index structure fits in the L3 cache and the 16K index is 6 times smaller than the L3 cache and should therefore never get a L3 cache miss.

The results are presented in columns 5-8 in Table 3.2. In column 5 we estimate the indexing time to build the full databases in isolation. We have to estimate this as we could not have the machine to ourselves long enough to finish all the experiments. The estimate is based on indexing a 4.28% subset (347M descriptors or 43GB of data) on 12 cores in isolation. We validated our estimate by indexing both subsets and full set of smaller collections, and the estimates are very accurate. The full construction of the databases was run in a shared environment, using only 4 cores, and it took several days.

During the indexing of the 4.28% subset, we used *TipTop* to monitor the threaded eCP code. In columns 6-8 we report the measured IPC and the L2 and L3 cache load misses per instruction, obtained with *TipTop*.

As was expected, there is a significant loss in the IPC as the size of the index grows. We can also see that the bigger the index is, the more pressure there is on the level 3 cache. This causes the CPU to be starved, as more cycles are spent idle, waiting for data to arrive from RAM, and thus the IPC drops.

We also made a preliminary evaluation of memory block per sub-branch version of the index structure using the large 8.1M clusters on disk configuration. This structure required much more RAM, but there was less pressure on the L3 cache and therefore the IPC was also higher. However, a better implementation and more experiments are needed. We grouped descriptors in aligned memory as much as the current design allowed, but with a redesign an even better descriptor

grouping and memory alignment is possible. If we can spare the RAM during the indexing, and the time to re-design and re-implement eCP again, we can make an improvement over the results reported here.

3.2.4 Parallel indexing: Summary

Overall, we showed the scalability of eCP by indexing a much larger dataset (8.1B descriptors), using a deeper index ($L = 5$) and using up to 24 (12 real + 12 HT) logical cores.

We evaluated the index structure and, as we expected, there is significant pressure on the cache both because of the throughput of data and the way we implement the index structure.

Indexing the full 8.1B (1TB) descriptor set using a viable index setting (like the 8.1M or the 1.6M) takes over 40 hours.

As we see in the last two columns of Table 3.2, even when using the deep L=5 and L=4 settings for the index, that should give us less calculation per index traversal and thus a lower indexing cost, the reality is that it takes more time to process the wider index. This is caused by the imbalanced cluster size distribution in the upper levels of the index hierarchy.

The increased cache pressure of using a wider index results in a loss in response time, as cache misses become more frequent. The randomness of the memory accesses that cause the cache pressure is due to how we use pointers in the index hierarchy to keep down the amount of RAM it requires. It is possible to reduce the pressure on the cache by allocating a block of memory each sub-branch of the index, and copy the *leaders* into that memory block instead of using arrays of pointers into one very large memory block with all the *cluster representatives*. However, that will require eCP to use a lot more memory for the index structure, especially if *treeA*>1, as each branch of the index will have a copy of its *leaders* in its memory block, effectively *treeA* folding the memory requirement.

3.3 Parallel search and batching

The goal of this section is to develop a multi-threaded batch eCP search. We start with the motivation for doing batch search, describing how we can trade response time for throughput. Then we describe some implementation details, followed by a presentation of our experiments and results. Finally we end this section with a short summary of it.

3.3.1 Parallelized eCP batch search: Motivation

As we saw in Chapter 2, the eCP search process is heavily I/O bound when only a single image is searched. This is because only a small subset of all the clusters are needed and retrieved from disk, making the access to disk a random read pattern. We also saw how better hardware (SSD) and the *early-halting* search policy can reduce the response time without sacrificing quality. Nevertheless, searching suffers from the randomness of the I/Os.

Another way to address search response time would be to do disk management with parallel access to multiple disks. However, when there are several disks in a single machine it is most common to use specialized hardware, like a RAID controller, to arrange the disks in a RAID volume that provides varying degrees of parallel access and fault tolerance. Also, the number of storage devices in a machine is limited without the specialized controllers, and thus also the scalability doing the I/O management in the eCP algorithm. For our experiments, we do in fact have 15 high performance (15K-rpm SAS) magnetic disks and they are arranged in a single RAID6 volume controlled with a high-end RAID controller.

The one thing we can do with regards to the disk access patterns of the eCP search process is to increase the size of the query. What we mean here is that we can batch multiple image queries together and run them all at the same time as a single search. The advantages are that some query descriptors will have overlapping cluster requests and by reordering the requests for clusters, we get better contiguity of the disk access. Thus, we are trading the response time of each individual image for greater throughput.

The overlapping cluster requests and better disk access was well described in Section 2.3.4 on optimizing the image-level search, from Chapter 2. Essentially, that discussion still holds true with regard to the batching of multiple images into a single search. One of the conclusions was that the second optimization, of reading the clusters in order, would not do much good for a single query on a large database with many clusters on disk. The scarcity of the clusters requests was such that the in order reading was a random access pattern anyway, due to the large offsets between the clusters read.

However, in the batch search, we can increase the number of requests (by adding more images to the batch) until we get better contiguity of disk access and start to benefit from the prefetching of the OS.

As the batch becomes larger and more and more clusters are prefetched by the OS (because of the good contiguity) the search process shifts from being I/O bound on random I/O to being I/O bound on the cost of reading the data sequentially. At first, this shift happens only temporarily, when a few clusters are read from the same region of the data file. But at some point, the cost stabilizes at the fixed cost of sequentially reading the whole data file.

At this point, when we keep adding more images to the batch, the search process will quickly become CPU bound on doing the scanning of the clusters. This is where the multi-threading comes in as an important factor. With multiple cores and threaded scanning, the average wall clock time needed per image will keep dropping. At least until we run out of computing power, by which time the search process is truly CPU bound.

Please note that even if we are reading the whole data file, we are still only scanning the b most similar clusters for each query descriptor in the batch and thus we are nowhere near doing any kind of sequential scanning of all the data against all the query descriptors.

3.3.2 Parallelized eCP batch search: Implementation

If we recall the eCP search process, we have three steps. First is the index traversal of all the query descriptors and the creation of a lookup-table where we can know what descriptors should be scanned against each cluster. Second we have the in-order loading and scanning of those clusters that are needed, creating the k-nn results for each query descriptor. And third and last step is the vote aggregation of all the k-nn results into an image-level ranked result.

To be able to load multiple images in a batch we need to add an identifier for each query image, such that we can separate them again before we do the vote aggregation. The search process is essentially not changed from the description in Section 2.4, except we need to add the step of identifying query images (step 2) and create a ranked image-level result for each image (step 6). The search process can therefore be described in the following 7 steps:

1. All query images in the batch are read from disk and the query descriptors are extracted. [3]
2. Uniquely identify each image and its descriptors in the batch.
3. All the clusters to retrieve and scan are identified by traversing the eCP index structure for each query descriptor.
4. A lookup-table is created such that for each cluster, the query descriptors that want to scan that cluster can quickly be found.
5. The clusters to retrieve are accessed in-order, to maximize contiguity of the I/Os, and scanned against the requesting descriptors, to fill their k-nns. This is the core of the search process.
6. Do a vote aggregation for each query image in the batch, creating the image-level ranked results.

3. In our case the extraction has already been done and we read the query descriptors directly from a file on disk.

7. The image-level results are written as text to a result file on disk, one per query image in the batch.

The main implementation change to the eCP search algorithm is in multi-threading the process. We devote one thread to do all the loading of clusters and we have a pool of threads to do the scanning. The loading-thread goes through all the clusters, in order, and loads any cluster that the lookup-table shows a request for. For the scanning of clusters we create a pool of threads. Once a cluster has been loaded, one of the scanning-threads is assigned to process it and populate the k-nn of the requesting query descriptor(s). This is essentially a FIFO producer-consumer protocol where the loader-thread produces clusters in memory and the scanning-threads consume them.

3.3.3 Parallelized eCP batch search: Experiments

For our experiments we use the 8.1M and the 1.6M databases that we created in Section 3.2.3 and are described in Table 3.2.

The size of the 8.1M index was no arbitrary choice. The ts of 1,000 points corresponds roughly to an average size of clusters of 128KB. Essentially, we are applying the same settings of ts as we did before, on a more than 73 times larger collection. To cope with this vast increase in the number of *cluster representatives*, the depth of the index structure is deeper than before or L=5. At that depth, the estimated cost of traversing the index structure is only 313 distance calculations.

The other index that we will be using, the 1.6M, has a ts of 5,000 descriptors per cluster (average size of 665KB on disk) and is L=4 levels deep. The estimated number of distance calculations in traversing this index is 357.

We mention the estimated cost of traversing the index structure, even if we know it is not a very accurate estimate (this is because of the clustering size distribution imbalance). Having this estimate can still be useful. For example, when we do indexing we use it as a guideline to set the index depth L. It is however also useful in the search process. For example, it tells us approximately how many clusters on the lowest level we are scanning and thus can be used as and a rough estimate of the approximation that we do.

We will explain with an example using the 8.1M settings and *treeA*=3. If we had used L=1 for that index structure, we would have to scan all 8,122,260 *cluster representatives*, but we would be guaranteed to find the most similar one every time (indexing with this setting would have taken a very very long time). At L=2 we do 11,400 calculations in total, but only 8,550 of them are to scan the bottom-level *cluster representatives*. Using the L=5 setting that we use, of the 313 distance calculations, only 72 of them spent scanning the bottom-level

cluster representatives. I.e. we are picking the most similar cluster(s) based on checking 72 of the 8,122,260 *cluster representatives* or only ≈0.0009% of them. [4]

We also do some searching on the 162K, ts=50,000 descriptors. This is however not a viable database as the scanning of such huge clusters takes a long time. The smaller and shallower index should be less prone to errors and we can use it as a point of reference for evaluating the loss of quality due to errors in the index traversals.

To evaluate the quality of our databases we use again the copyright detection scenario. We use two evaluations sets. Both are based on randomly picked pictures altered with StirMark and from each image, we create several query images.

The CopyDays set is based on a random selection of 127 images, each altered with 9 crop variants, 9 jpeg compressions and a strong variant. Included in the strong label are several manual attacks. In total, the CopyDays query set has 3055 queries.

The second set, called 49K, contains 48,883 query images derived from 1,000 randomly picked database images. There are 49 different StirMark attacks used: affine transformation(3), conversion(2), cropping(2), jpeg compression(2), median transformation(1), adding noise(1), PSNR(1), rescaling(6), RML(4), rotation(8), rotation and cropping(8), rotation and rescaling(8) and SS(3).

As before, we only consider a "true match" the case when the correct database image is the top ranked (P@1) image-level result returned.

Evaluating search quality Previously, in Table 3.2, we described how we built three indexed databases over a collection of 8.1B descriptors. A mandatory task is to evaluate the search quality of those databases, especially the 8.1M and the 1.6M indexes. In Table 3.3 we have the search quality and average running time per image, using all three indexes using both b=1 and b=3.

> **Please note** that for this set of experiments, the running times where not obtained in isolation. I.e. other users could use the machine at the same time. We must keep this in mind as we asses the results. All experiments where repeated several times and the best runs are reported.

The first observation is that the search quality is very good. We know that a 100% correct identification is impossible. From some StirMark variants no or only a few query descriptors can be extracted; 31 of the 3055 CopyDays variants

4. This is not the only approximation, as then, assuming we use b=1, we are going to populate the k-nn of each descriptor by only fully scanning (on average) only 1,000 descriptors of the 8.1 billion in the database. That is ≈0.000012% of the full set.

DB settings				CopyDays, all 3,055 query images			
	Clusters			b=1		b=3	
ts	on disk	L	*treeA*	time/image	P@1/R@1	time/image	P@1/R@1
50K	162K	3	3	1.16s.	90.11%	-	-
5K	1.6M	4	3	0.93s.	86.68%	1.45s.	90.11%
1K	8.1M	5	3	0.64s.	84.48%	0.88s.	87.33%

DB settings				49K set, all 48,883 query images			
	Clusters			b=1		b=3	
ts	on disk	L	*treeA*	time/image	P@1/R@1	time/image	P@1/R@1
50K	162K	3	3	-	96.58%	-	-
5K	1.6M	4	3	0.18s.	95.03%	0.22s.	96.57%
1K	8.1M	5	3	0.18s.	93.58%	0.23s.	96.56%

Table 3.3: Search quality and running times of batch search using the CopyDays and 49K query sets. The three databases are built indexing the 8.1B descriptor collections using the 8.1M, the 1.6M and the 162K index structures described in Table 3.2. The search loads the full query set in a single batch and the k-nn size is set to k=20.

have less than 8 query descriptors extracted. Also, many variants are severely altered, making them very hard to match to the original image. An example of this is the manually attacked Strong category in CopyDays. If we exclude it from our results, the overall quality for all the other variants is increased 4% on average.

The second observation is that again, we see that picking many small clusters is both faster and gives the same, or even better, quality as scanning a single large cluster (see 8.1M b=1 vs. 1.6M b=3 in the result table). This indicates that the cause of quality loss lies more in the last step of traversing the index, i.e. in the choice of *cluster representative*, than in the scanning of clusters or the size of k-nns. When we run the same searches with k=100 the quality tended to drop, in some cases as much as 2%.

To dig deeper into what happens at the query point level, we pick one of the 49K query images and look at what is happening for each of the 49 variants for all three index structures. The original image consists of 339 SIFT descriptors. When all 49 variants are searched against the different databases, we get the following:

- On 162K we have 100% recall as the correct database image is the top-ranked result for all 49 variants. The average number of votes for the correct image was 80.89 with a maximum of 294 and a minimum of 12.
- On 1.6M we have 98% recall as one variant did not find the original image as its top ranked result. The average number of votes for the correct image

was 56.45 with a maximum of 274 and a minimum of 4. There was thus a reduction of 24.53 votes on average.
- On 8.1M we have again 100% recall. The average number of votes for the correct image was 47.18 with a maximum of 263 and a minimum of 6. The "lost" variant was regained, but the correct image is getting fewer votes on average, 33.80 less votes than we got using the 162K index. The k-nn is always the same size, so the loss here is only due to the search not accessing the correct cluster to scan.

Clearly we are loosing recall at the point-level but our search requires only a few points to make a positive identification, so for many of the variants we "can afford" the loss. We also need to keep in mind that the scale-up difference between the 162K and the 8.1M cluster index is a factor of 50. At least we have shown that a level 5 index, of 8,122,260 clusters on disk, is still providing quite good quality results for a collection of 8.1B descriptors, extracted from ≈25M thumbnail-sized (150px on wider edge) images that have on average only 300 SIFT descriptors per image.

Evaluating throughput in batch search Our theory is that there are two advantages we should expect from searching large batches. With more query descriptors, we can reduce the total number of cluster requests as redundant request, that overlap, can be merged. The second advantage is that as the number of clusters requested grows, we are more likely to benefit from better contiguity and the prefetching of data.

What we do not know is how large the batch must be to get better throughput. This will depend on the size of the dataset (the number of clusters on disk), how frequently cluster the requests overlap and the amount of prefetching that is being done by the OS.

We use the two "reasonable" databases, the 1.6M and the 8.1M (see Table 3.2), that we created in the multi-threaded indexing experiments described earlier.[5] Our goal is to run multiple searches, with ever larger batches, on these two databases and measure the performance of the multi-threaded eCP batch search.

We use three query sets. The first two are the CopyDays set and the 49K set that we used in the previous search experiment and we have already reported the search quality, see Table 3.3. The third set, that we call the Unique set, consists of 100K unique images downloaded from Flickr that are NOT in the database. There is therefore no need to report the search quality for this set.

5. We do not search the 162K database, that has a ts of 50,000 descriptors, as it is too large to be considered feasible. It would simply make more sense to search more clusters (b>1) using the 1.6M database as we have shown it to give better quality and be more responsive.

We should recall how the first two sets are created. The large number of query images we create using StirMark are in fact derived by altering a smaller set of randomly selected images that are in the database (the CopyDays set is created from only 127 database images and the 49K set from 1,000). Because each variant is an altered version of the original image, the query descriptors extracted from them are likely to be similar to the descriptors extracted from the original image (i.e. there should be a one-to-one correspondence). As this relationship is transitive (A≈B, B≈C therefore A≈C) all the variants are also similar to each other. When we batch query images, and many variants of the same original image are included, there will be much higher overlap between of cluster requests than could be considered a "normal payload" of equal size where all the images are unrelated or unique. This is why we include the third set of unique random images, i.e. to establish a realistic baseline for a realistic payload.

We should also be aware of a subtle difference between the batches of the 49K set and Copydays set. The order of the query images in the 49K batch is by imageID and thus the first 49 queries are all variants of the same original image. This will create the maximum overlap with even the smallest of batches. The batch of CopyDays images is sorted by variant, and thus the first 127 queries are from unique original images.

In Table 3.4 show information on the batch structure and how it relates to the database. The rows are the batches we use, range from 10 - 100,000 images (2,455 to 31,170,600 query descriptors). In the first column we have the query set size in images. Then, for each query set, we show 4 columns:

1. The number of query descriptors in the batch (Query desc.).
2. The number of cluster requests issued (Clst. req.).
3. The ratio of descriptors that do not issue a cluster request (Q-desc. overlap).

$$\frac{(\#\text{descriptors} - \#\text{cluster requests})}{\#\text{descriptors}}$$

4. The ratio of database clusters requested and accessed (Clst. % accessed).

If the size of every cluster was the cluster *target-size*, the ratio of clusters accessed (Clst. % accessed) would also tell us the ratio of the data that was relevant to some query descriptor in the batch. However, we know that the size distribution of clusters is imbalanced. In fact, we are more likely to access the larger clusters and thus, the ratio of the data that is relevant to the batch is probably higher than this ratio indicates.

Let us dig into the content of Table 3.4 by focusing on one database, the 1.6M, and the overlap of cluster requests (Q-desc. overlap).

With only 10 images in the batch for the 49K set, 32.43% of the 2,892 query descriptors have overlapping requests (i.e. 938 query descriptors want to scan

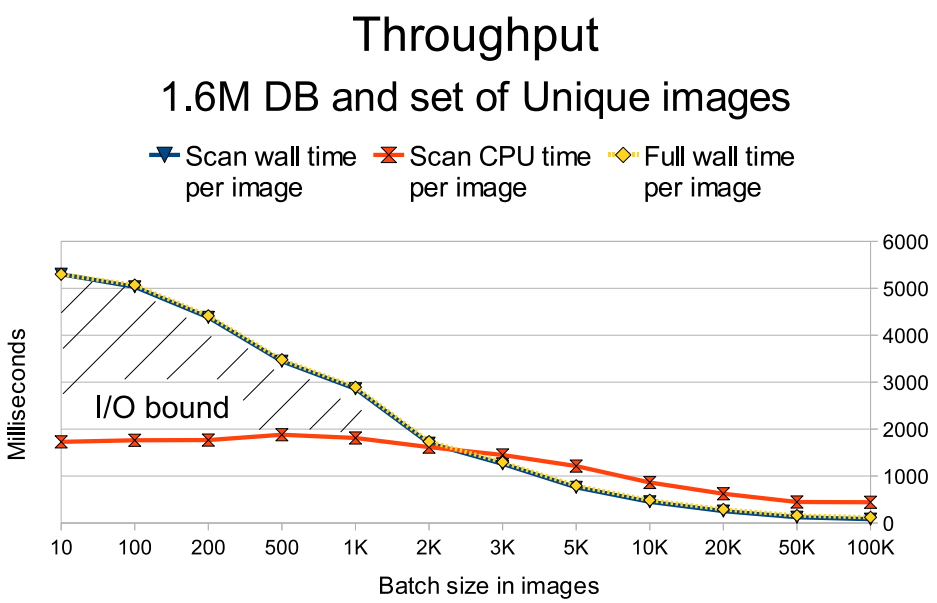

Figure 3.7: In this graph we are searching the batches for the Unique image set on the 1.6M database. In blue we have the average wall clock time per image during the cluster retrieval and scanning only. In orange we have the average CPU time per image for that same process. The yellow dashed line represents the full average wall clock time (i.e. including the cluster discovery and vote aggregation). The search is not expanded, i.e. b=1.

the same cluster as some other query descriptor). The same batch size in the CopyDays set and Unique set, have only a ≈5.0% requests overlap. Even for the small batches, the main gain for the 49K set is this reduction in the number of cluster requests it needs to do. The benefit of prefetching and contiguity of access requires higher density of requests than 1,954 out of 1.6M clusters on disk (the average jump between clusters is over 500MB).

With a batch of 3,000 images (the 3K row) the descriptors in the batch have grown to ≈950K descriptors (967K for Unique set, 955K for CopyDays and 940K for the 49K). For this batch size the Unique set has a 39.48% overlap, and is accessing 36.02% of the database, while the CopyDays has more overlap, 56,00%, it is only accessing a quarter (25.86%) of the clusters (same applies to the 49K set, 61.43% overlap and 22.33% clusters accessed). While the Unique set may not gain much from overlapping requests, it is increasing the density of cluster requests much faster and will start to gain from prefetching earlier than the batches from the other two sets.

As the number of images in the batch becomes larger, and there are more query descriptors, the overlap is increased as well. With a batch of 5,000 images (the 5K row) the Unique set has 1.6M query descriptors, or exactly the number of

8.1M Index

Images in batch	Unique set Query desc.	Unique set Clust. req.	Unique set Q-desc. overlap	Unique set Clst. % accessed	CopyDays Query desc.	CopyDays Clust. req.	CopyDays Q-desc. overlap	CopyDays Clst. % accessed	49K set Query desc.	49K set Clust. req.	49K set Q-desc. overlap	49K set Clst. % accessed
10	3,530	3,399	3.71%	0.04%	2,455	2,377	3.18%	0.03%	2,892	2,120	26.69%	0.03%
100	31K	29K	4.02%	0.36%	31K	30K	3.51%	0.37%	37K	20K	46.94%	0.24%
200	63K	60K	4.59%	0.74%	63K	59K	6.41%	0.73%	59K	31K	46.69%	0.39%
500	158K	149K	5.89%	1.84%	158K	135K	14.38%	1.66%	165K	90K	45.11%	1.11%
1K	315K	290K	8.01%	3.57%	316K	571K	19.61%	3.13%	302K	165K	45.34%	2.03%
2K	630K	557K	11.59%	6.86%	626K	430K	31.28%	5.30%	581K	312K	46.26%	3.85%
3K	967K	821K	15.05%	10.11%	955K	571K	40.20%	7.03%	940K	500K	46.83%	6.16%
5K	1.6M	1.3M	20.64%	15.43%					1.5M	788K	49.01%	9.70%
10K	3.1M	2.1M	31.49%	26.10%					3.1M	1.4M	53.85%	17.72%
20K	6.2M	3.3M	46.09%	41.22%					6.2M	2.4M	60.81%	30.12%
49K	15.6M	5.2M	66.67%	64.03%					15.0M	4.1M	72.43%	51.04%
100K	31.2M	6.4M	79.45%	78.86%								

1.6M Index

Images in batch	Unique set Query desc.	Unique set Clst. req.	Unique set Q-desc. overlap	Unique set Clust. % accessed	CopyDays Query desc.	CopyDays Clust. req.	CopyDays Q-desc. overlap	CopyDays Clust. % accessed	49K set Query desc.	49K set Clust. req.	49K set Q-desc. overlap	49K set Clust. % accessed
10	3,530	3,357	4.90%	0.21%	2,455	2,331	5.05%	0.14%	2,892	1,954	32.43%	0.12%
100	31K	28K	7.07%	1.75%	31K	29K	6.35%	1.79%	37K	17K	54.46%	1.03%
200	63K	57K	8.99%	3.53%	63K	56K	11.08%	3.45%	59K	27K	54.24%	1.65%
500	158K	137K	16.69%	8.42%	158K	121K	23.41%	7.44%	165K	75K	54.23%	4.64%
1K	315K	251K	20.40%	15.45%	316K	213K	32.63%	13.11%	302K	132K	55.53%	8.27%
2K	630K	435K	30.97%	26.77%	623K	333K	46.74%	20.52%	581K	241K	58.59%	14.81%
3K	967K	585K	39.48%	36.02%	955K	420K	56.00%	25.86%	940K	363K	61.43%	22.33%
5K	1.6M	784K	50.36%	48.24%					1.5M	527K	65.92%	32.43%
10K	3.1M	1.1M	65.57%	65.56%					3.1M	811K	74.00%	49.91%
20K	6.2M	1.3M	78.88%	80.76%					6.2M	1.1M	82.20%	68.39%
49K	15.6M	1.5M	90.32%	92.97%					15.0M	1.4M	90.71%	86.00%
100K	31.2M	1,6M	94.94%	97.06%								

Table 3.4: Statistics for various batch sizes, no search expansion is used (b=1). First is a set of Unique images, then the CopyDays set and last is the 49K set.

clusters on disk in this dataset. Here we can see that almost half (48.24%) of the descriptors are overlapping their cluster requests. One row below, with a batch of 10,000 images and 3.1M query descriptors, the query descriptors outnumber the clusters almost two-to-one and the overlap has grown to 65.56%. We must also remember to take into account that the imbalance of cluster size distribution and that the smallest clusters are least likely to be requested. Therefore 65.56% of clusters may represent a much higher proportion of the data.

In Figure 3.7 we can see the time to do the "retrieve and scan clusters" part of the search, using the Unique set of images and the 1.6M indexed database. Reported are both the wall clock time per image (blue) and the reported CPU time per image (orange). I.e. the vote aggregation and cluster discovery are not included in those times. As we can see, the yellow dashed line that shows the average "full wall time" follows the "scan wall time" exactly. This shows that the dominating cost factor is the retrieval and scan process. We can therefore talk about the average scan wall clock time as if that is the only time that matters.

The Unique set of images is the set of queries that has the least amount of cluster request overlap, and thus we are using it to evaluate how large the batch must be for it to start benefiting from the prefetching. To tell exactly when this happens is impossible, some batches may switch between being I/O bound on the random I/O to being I/O bound on the cost of sequentially reading. It will take a lot of query descriptors to keep all the 24 logical cores busy. But we can tell for sure that any batch below 2,000 images (630K query descriptors) is I/O bound. This is because after the 2K mark, the eCP search is reporting more CPU time used than the wall clock running time. However, as our search is multi-threaded we may still be waiting on I/O for the larger batches.

The average wall clock time is dropping the whole time, from around 5.3 seconds per image in the 10 image batch to only 81.46 milliseconds per image in the batch of 100,000 images. Some of this gain is due to the cluster request overlap.

As the batch grows, the search process goes through stages of bottlenecks. First it is I/O bound on doing random I/O, jumping all over the datafile to retrieve the few and scattered clusters that are relevant and need to be scanned. Then the bottleneck becomes less and less the randomness of the I/O and more on the cost of doing the sequential I/O. Ultimately, if there are enough query descriptors wanting to scan each cluster, it will become CPU bound, as all the cores are kept busy for most of the running time.

We can tell that this has not happened yet, as we have 12 real cores and 12 HT cores we can estimate that the gap between measured wall clock time and CPU time, should be 15.6 times higher (assuming the HT cores do 30% work). The reported difference here, using a batch of 100,000 images, is only 5.42 times (81.46ms. wall clock vs. 441.65ms. CPU clock). We should thus be able to run

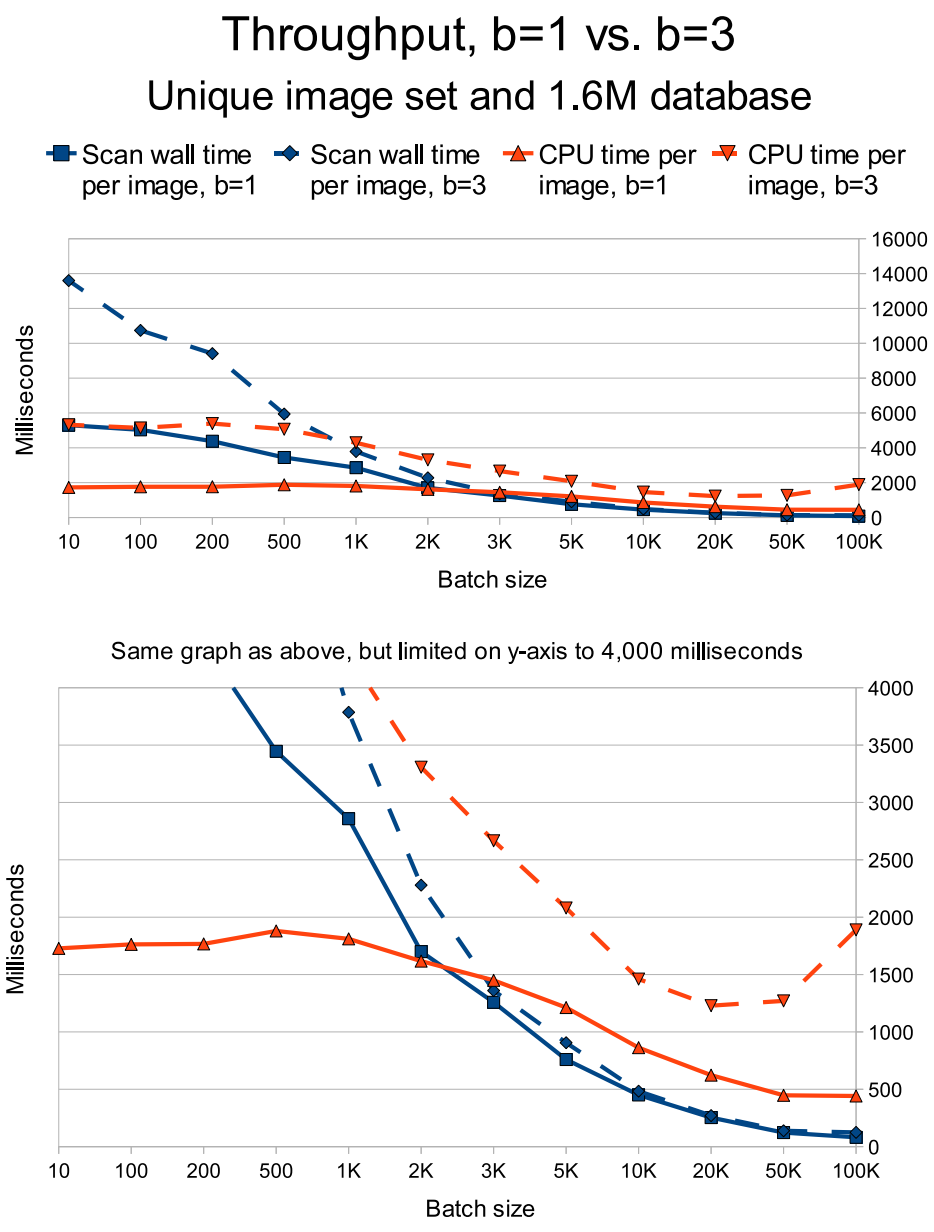

Figure 3.8: In this graph, the batches of query images come from the Unique set of images and we are searching the 1.6M database, showing both b=1 and b=3. This is in fact one graph, the second image is just a zoom in of the first graph. In solid blue we have the average "scan" wall clock time per image and in solid orange we have the average scan CPU time per image for searching with b=1 (the same values as in Figure 3.7). In addition, we have added the same values for searching with b=3, using the same colors but dashed lines.
Please note the CPU time for the largest batch (100K) using b=3. The reported time is going up, indicating that the search is, or is near to, being CPU bound. The increased CPU time is caused by the "misreporting" HT cores, and they are only used significantly when the real cores are already in full use.

at least 200,000 images in a single batch and still be gaining more throughput on this hardware.

A different approach is to divide the measured wall clock and CPU time by the number of cluster requests we issue (i.e. by the number of clusters retrieved and scanned). By dividing with the wall clock time, we get the average time it takes to do both retrieval and scanning per cluster, while dividing by the CPU time, we only get the average time it takes to do the scanning the clusters retrieved.

In this data we see a steady cost of ≈15ms. per cluster that starts dropping at 1K (11.4ms) to the lowest value at 3.9ms. at 20K batch size. As the batch grows larger, this value grows slightly again, but that is understandable. The largest batches are accessing almost all clusters (>90% for 50K and 100K) and there are simply more query points to process per cluster.

In Figure 3.8 we report the running times for the batches of the Unique image set, using both b=1 (as before) and b=3. The most interesting thing about this graph is that the 100K batch using b=3 is the first time we see an increase in the average reported CPU time per image. This is strong evidence that we are CPU bound (or very close to it) as we are starting to using the HT cores that are "misreporting the CPU times" (see discussion in Section 3.2.3). The gap between the reported wall clock time and the CPU time for this batch is 15.22 times more CPU time (164.43ms. wall clock vs. 1889.39ms. CPU clock per image), almost the 15.6 estimate that we calculated earlier as the maximum before we would be CPU bound.

We should also note that when the I/O cost stabilizes at a fixed cost of reading the whole database sequentially, the b=1 or b=3 do not matter either; We see this happen when the solid blue and the dashed blue lines meet at the 10K batch size. The setting of b does not matter because the multiple-threads are capable of doing the added CPU work, at least until the batch becomes so large that we are truly CPU bound. This is just starting to happen with the 100K batch for the b=3 setting.

In Figure 3.9 we see the average wall clock time and average CPU time per image when we scan the Unique set on the 8.1M database. Essentially, this is the same graph as we showed for the 1.6M set in Figure 3.8.

From the graph for the 8.1M database, i.e. Figure 3.9, we can see that scanning the 5 times smaller clusters (ts=1,000 instead of 5,000) requires much less CPU power and the average CPU times are much lower over the whole range of batch sizes. Unlike before, the b=3 setting for the largest batch has no increase in CPU time. Therefore, we should be able to increase the batch sizes well beyond the the size of 100,000 images and still improve on the average wall clock processing time per image.

We should also note that it takes much more images in the batch before the wall clock time line crosses the CPU time line, indicating that the search has

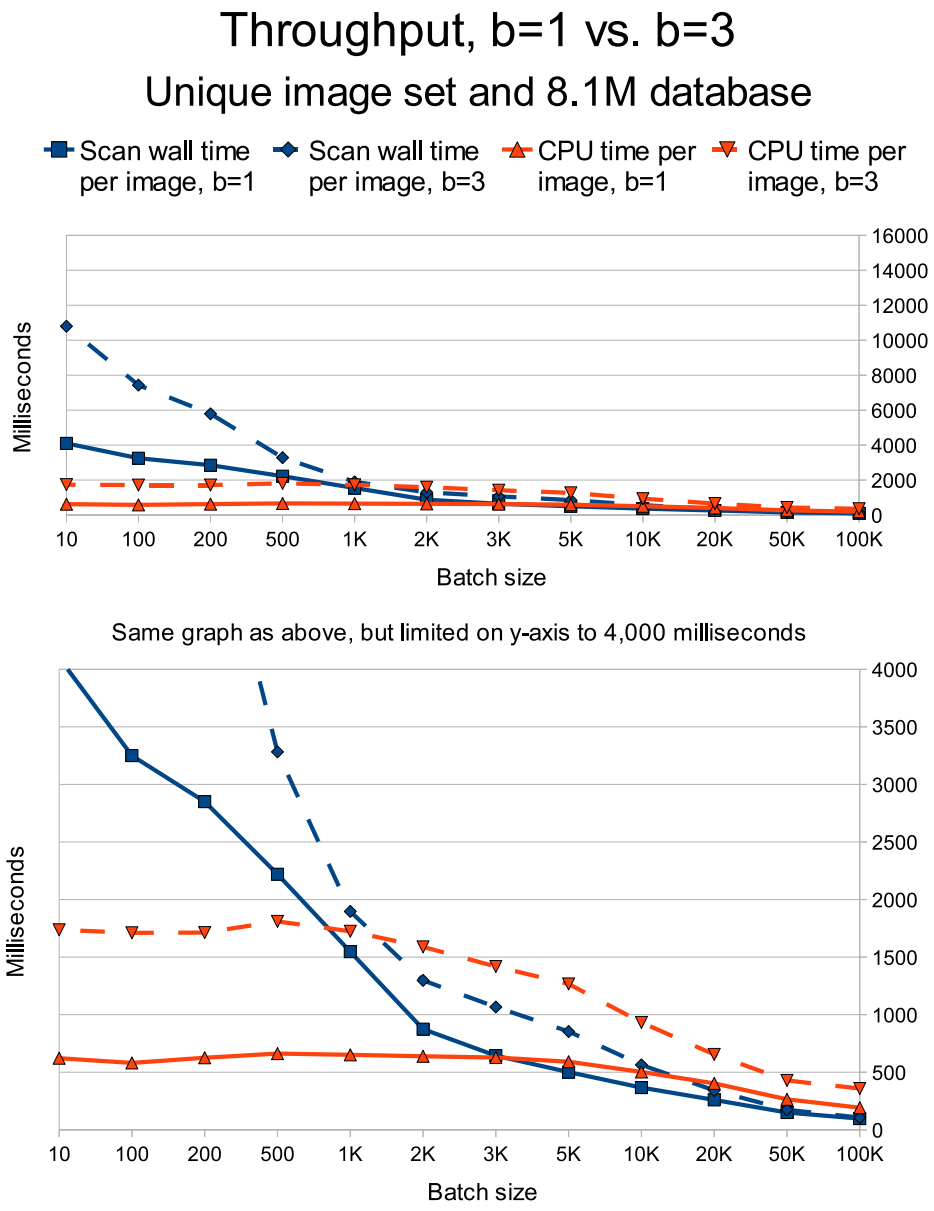

Figure 3.9: In this graph, the batches of query images come from the Unique set of images and we are searching the 8.1M database, showing both b=1 and b=3. This is essentially the same kind of graph as we showed for the 1.6M database in Figure 3.8.

shifted from being I/O bound to be CPU bound on a single core (i.e. without parallelization we would be CPU bound).

3.4 Discussion

In this chapter, we have shown that the eCP indexing process can be parallelized and that it scales very well on a powerful multi-core machine (see Figure 3.5). If we take care, and follow the recommendation for Hyper-Threading, we even manage to squeeze a little extra work out of the second logical core in our HT enabled CPUs.

Despite the good scalability, both the low IPC and the cache pressure we observed, columns 7-9 in Table 3.2, indicate that our pointer-based index structure can be improved upon. The key optimization is to focus the memory access during index traversal on blocks of memory that are small enough to be cached. However, because of how *treeA* works, this calls for a much larger memory footprint that may become a limitation for scalability.

The standard one-image-at-a-time search process will not benefit from additional cores as that process is I/O bound. However, we have proposed and evaluated a batching search process that trades response time for throughput. As the size of the batch grows, more and more clusters are requested from disk and the density and contiguity of the I/O shifts from being a random I/O pattern to a sequential read of all the data. The increased throughput initially comes from less I/O cost per cluster, with overlapping requests being merged and the in-order reading of clusters benefiting from OS prefetching.

As we grow the size of the batch further, we shift the process more and more from being I/O bound to CPU bound (on scanning the clusters). This is where multi-threading the search becomes important, as from this point on, the increased throughput is coming from making full use of the 24 logical cores available to us. We therefore multi-threaded the search process, harnessing the available CPU power for those batches that are large enough to need it.

As it turns out, we use almost all the available cores when we scan a batch of 100K query images from the Unique set (31.2 million query descriptors), using b=3 on the 1.6M database. We can see this clearly in the lower graph of Figure 3.8, where the dashed orange line, that represents the average CPU time per image, is going up because the search is using the HT-cores that "over" report the CPU time. This indicates that we are close to fully using all the available CPU power.

In those same graphs of Figure 3.8, we can also see that with a batch of 500 query images we are already getting twice the throughput of a 10 image batch and with a batch of 5,000 query images we are below 1 second per image, regardless of weather we do b=1 or b=3.

One possibility we did not explore, but is quite possible, is to include the *early-halting* policy into the batch search process. This could be a challenging and yet very interesting combination.

When using a multi-core architecture, there are a few issues that we should be aware of and that we can draw lessons from. They are the following:

1. The latency of the RAM is important. The latency can become a significant bottleneck when cache misses are frequent and pages of memory have to be brought in often.
2. Task management of threads is important: many small tasks are best. We tested several thread spawning and work load policies and the best one was to rely on a pool of threads that is created only once. Then, each thread is given a small task again and again. This way, the slowest of the sub-tasks (HT cores) can not significantly delay the overall running time of the full job.
3. Having taken care of the threads task-size, using Hyper-Threading can help, but not that much. The problem is basically this: If your task is very well optimized, there will be little or no unused CPU cycles for the second task to run. If however (as is our case) your code is blocking very frequently for a many cycles (like waiting from RAM) both your threads are likely to be waiting in a blocked state. In the future, it is not unlikely that we will see processors that will support a multiple (or even a dynamic setting) of how many logical cores will be available for each real core.
4. From a practical point of view, the traditional Unix tools for performance measurements are of limited use when measuring multi-core applications that are using the processor SMT cores.

In connection with the last point, we highly recommend a monitoring tool called *TipTop*, that is built on the Unix *top* command. *TipTop* gives non-expert users a few simple low level metrics that help them gage their code and making the right design decisions when implementing and optimizing on multi-core architectures [Roh11].

Chapter 4

Scalable CBIR, Distribution

In the previous chapter we showed that the parallelized eCP could harness the processing power of a multi-core machine quite well, indicating good scalability. We indexed a dataset of 8.1B SIFT descriptors extracted from over 25 million images, or 1TB of data on disk. However, even with all 24 logical cores (12 real), the database construction takes over 40 hours. The scalability of a single machine is limited by the amount of hardware we can put in a single box.

The web-scale CBIR systems have to be able to cope with hundreds of millions of images, or tens to hundreds of billions of descriptors. For eCP to be a viable solution for this amount of data, we need more computing power than a single machine can provide. Therefore, it is our goal in this chapter to harness the power of distributed computing. In Section 4.2 we develop a distributed eCP indexing algorithm.

The standard image-at-a-time search process is bound on random I/O and will thus not benefit directly from having more CPU power available. Distributed infrastructures can provides more than just increased computing power. Typically, but not always, each machine will have its own local secondary storage. For the search process, the main advantage a distributed system has to offer, and the one we will seek to exploit, is that there are several hard drives available. However, those drives are distributed over several machines and only connected over a limited and potentially costly network. Harnessing them will thus neither be straight forward nor a simple matter.

In the previous chapter, we proposed a parallelized batching search process that shifts the bottleneck from the random I/O toward sequential I/O and using more and more CPU power. This kind of search, if the batch is large enough, can take full advantage of both more disks and more CPU power. We develop a distributed version of the batch search process in Section 4.3.

We did look into distributing the one-image-at-a-time search. However, when we consider the gains of using an *early-halting* policy and/or SSD devices that we

demonstrated in Section 2.5.5 of Chapter 2, we dismissed ideas of this kind. We think it will be hard to get competitive response times in a distributed environment that has to deal with network delays and coordination overhead. Developing such a solution remains on our list of future work.

The final section of this chapter is devoted to summarizing the content of the chapter, drawing lessons from our work and experiments. This discussion is found is Section 4.4.

To make a web-scale distributed CBIR system, there are certain capabilities that are desirable. Among them are: automatic distribution, fault-tolerance, re-execution of failed subtasks and network and I/O management that prevents I/O bottlenecks, just to name a few.

Implementing this full range of capabilities is an enormous task using traditional distributed computing programming models, like Open MP. Therefore, we start the chapter with a background section, where we look at the state-of-the-art in automatically distributing systems that have most, if not all, of the desired functionality already built in to their systems.

4.1 Background

There are several programming models available that provide variable degrees of assisted and/or automated distribution of computations. In this section we will introduce and discuss three such frameworks.

We start with two systems the exploit "data independence" to automate the distribution of calculations. First we have the Hadoop framework that is based on the Map-Reduce programming paradigm. Hadoop is a large framework developed in JAVA in an open source community under the guidance of the Apache Software Foundation or ASF.[1] We then briefly describe a similar system from Microsoft, called Dryad.

The third system we will discuss, called GraphLab, takes a very different approach. This system was proposed to address the needs of algorithms that do not have data independence that can be exploited to automate the distribution of the calculations. What is proposed instead is a priority based sub-task execution with an elaborate locking mechanism, where the user needs to a priori define the dependencies in the form of a graph.

We will however start our discussion with an introduction to the Map-Reduce programming paradigm that Hadoop is based upon.

1. ASF is most famous for its HTTP servers, but in 1999 it turned into a not-for-profit community that actively supports several open source projects.

4.1.1 Map-Reduce

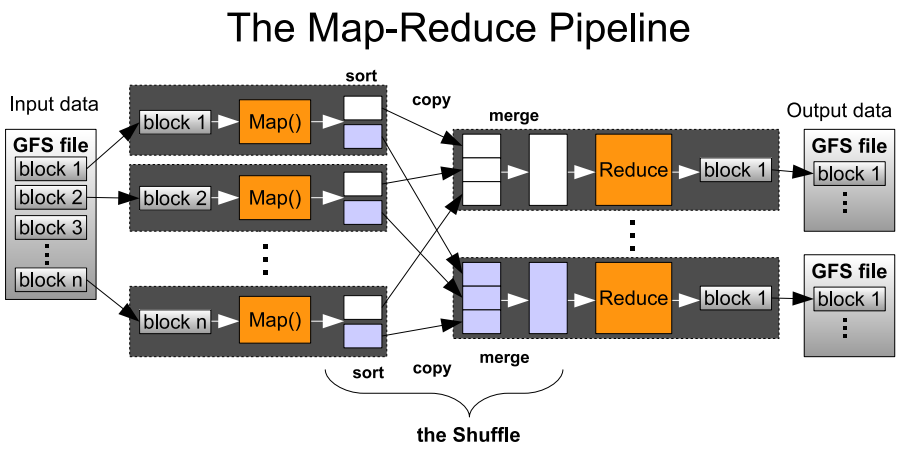

Figure 4.1: The Map-Reduce Pipeline, showing the flow of data from GFS input file(s) to GFS result file(s).

The Map-Reduce programming paradigm [DG08] was published by Jeffrey Dean et al. (Google) in 2008. It is designed to efficiently process extremely large datasets on a cluster of machines. This is achieved by exploiting data independence to split the work into subtasks that can be automatically run in parallel in the distributed environment.

In Figure 4.1 we have a picture of the so called Map-Reduce pipeline, showing the flow of data from input to output. The developer is tasked with implementing only two functions, the **Map** and the **Reduce** functions. The job's input data is distributed in blocks to the participating machines using the distributed Google File System or GFS [GGL03]. When a job is launched, the system automatically spawns as many *Map* functions as there are blocks of data to process. Each *Mapper* reads the data iteratively as a key-value pair record. It then processes it and, if necessary, outputs a new key-value pair record that is bound for a **Reduce** function. All records with the same key go to the same Reduce task.

The framework thus includes a sort-copy-merge data step that is called "the Shuffle", where data from several *Mappers* gets directed to specific reducers depending on their key (in Figure 4.1 this is indicated by different colors for the data that is in the Shuffle). 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 failed task by restarting them, possibly on a different machine and/or by reading a different replica of the block of data that is to be processed. The framework uses as little network bandwidth as possible by processing data where it resides or at the nearest available machine, using rack-awareness to pay attention to the network topology and minimizing reading over machine-rack boundaries.

4.1.2 Hadoop and HDFS

Hadoop is an open-source framework implementation of the Map-Reduce programming paradigm. The development is being done under the Hadoop project that is maintained by the Apache Software Foundation that is also 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 that is publicly available.

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 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.

Another feature that can be enabled is "speculative execution". When this is enabled, the *jobtracker* is allowed to schedule the same *Map*-task on multiple machines at the same time. Only the output of the task that is first completed is delivered to *Reducers*. This feature is especially useful near the end of jobs, when straggling tasks can potentially delay the entire process, while most of the *tasktrackers* are idle.

HDFS [Bor07] is based on the Google File System [GGL03] and 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 default blocks size is 64 MB, but this is configurable on a per file basis.

The architecture of HDFS consists of several datanodes storing the blocks of data and a centralized namenode that is responsible for keeping track of the file metadata as well as the locations of each files data blocks. HDFS handles failures through block-level replication (by default 3 copies are kept of each block). When distributing the replicas to the datanodes, HDFS employs a rack-aware policy: The first block is stored on the local disk of the originating datanode if possible, otherwise on a datanode of that is in the same rack; The second copy is always written to a namenode on a different rack, as part of an effort to maximize fault tolerance;[2] The third copy is then written to a different namenode on the same rack as the first copy. If the replication factor is set higher than 3, the version of Hadoop that we use (0.20.2) will randomly select a another namenode to store it on, even if that namenode has a copy of that block of data already.

Map-Reduce is a very simple yet extremely effective programming model adopted by many large scale applications designers today. On top of the core of Hadoop, described above, several specialized applications have been developed that aid developers to run specific tasks in a distributed environment with minimal effort. For example: a data warehouse system called Hive, scalable and fault tolerant database systems like Hbase and Cassandra; and a system that is targeted for machine learning algorithms called Mahout, just to name a few.

In addition to being used in cluster computing, Hadoop is becoming a de-facto standard for big-data applications running in the 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. For example, Amazon's Elastic Compute Cloud contains tens of thousands of virtual machines and supports Hadoop with minimal effort.

4.1.3 Dryad

Dryad [IBY+07] is an alternative to Map-Reduce, proposed by Michael Isard et al. of Microsoft Research. Dryad was targeted to run on Microsoft's cloud services and to be able to run applications developed in C# and managed C++. However, the last publication on Dryad was in early 2009 and the project seems to have been abandoned (at least temporarily) for the much more popular and freely available open-source Hadoop framework.

The main difference between Map-Reduce and Dryad is that Dryad allows for a more a flexible pipeline. In Dryad, the developer can define how many

2. Even if an entire rack of machines is lost, the HDFS files are still intact.

"functions" he wants to use and at each step, it is possible to use multiple input sources. In addition, the data can reside on various kinds of media ranging from local storage to FTP.

4.1.4 GraphLab

Both Dryad and Hadoop exploit the data independence, i.e. the fact that the calculations on the data are independent of one another, and thus the work can easily be parallelized and distributed automatically. There are many algorithms where the calculations are dependent on the data, i.e. the results of one calculation will affect the outcome of consequent calculations and therefore the order and even the number of calculations are dependent on the previous results. This is very common in machine learning algorithms and it is even applicable to the halting condition of the k-means clustering algorithm.

In light of how difficult it is to adapt machine learning algorithms to benefit from Hadoop, Yucheng Low et al. proposed an alternative system they called GraphLab [LGK+10, LGK+12].

The core of GraphLab is a user provided data graph, a directed acyclic graph or DAG, that encodes both the problem specific sparse computational structure and directly modifiable program state. I.e. the graph captures both the data dependencies and the nature of the computations. A shared data table is also used to capture any global state of the algorithm.

There are two ways to do computations, an "Update function and a "Sync mechanism". The "Update function" is a stateless user-defined function, much like the Map-Reduce *Map* function. The "Sync mechanism" is a three step process of Fold-Merge-Apply that can be seen as replacing "the Shuffle" and the *Reduce* function of the Map-Reduce. The Fold function sequentially aggregates data across vertices, the Merge function is an optional user provided function that combines results from parallel folds and finally the Apply function finalizes[3] the new value before it replaces the old one that is being updated.

In the DAG, the modifiable variables of the program state can be arbitrarily associated with both graph vertexes and edges. Around each vertex v a neighborhood Sv is defined as the vertices and edges (both inbound and outbound) connecting to v. The neighborhood data D_{Sv} is also defined, i.e. all blocks of data associated with vertices and edges in Sv. The scope of the user-defined Update Function is defined this D_{Sv}, i.e. the maximal range of data manipulation invoking the Update Function on v.

The consistency of the data is guaranteed by using the DAG and denying the parallel execution of overlapping scopes. However, with three models, Full (all edges and all vertices may be updated and are therefore locked), Edge (v and

3. Decay or discounting would be done at this step for example.

all edges may be updated and are therefore locked) and vertex (only v may be updated), the true effective range of the scope can be defined more specifically.

Graphlab provides several schedulers to run the jobs, like synchronized (everything updated in parallel at the same time), round-robin (fair) or prioritized scheduling where each task can reschedule itself with a new priority (fast convergence). What scheduler should be used will depend on the nature of the algorithm and what best suites the problem at hand.

Initially, GraphLab was only working on a single multi-core machine. It has since been extended to run in distributed environments [LGK+12]. Partitioning of the user-provided DAG between the participating machines is a potential problem as it can cause sever communication overhead if many edges fall across machines. GraphLab does not provide its own distributed file system, but has been adapted to run on HDFS.

4.2 Distributed eCP indexing

In this section we develop a distributed eCP indexing algorithm.

In Section 3.2.1 of Chapter 3 we showed that the eCP indexing process could be split into three phases, and that it was phase #2 where we do all the assignments of the descriptors to clusters that is the one that benefits most from being run in parallel. The task is therefore to run this phase in a distributed environment and hopefully taking full advantage of all the available hardware.

We will start by doing an overview where we analyze what problems we are facing and describe what assumptions we will be making. Then we develop the distributed eCP indexing and give the details of our implementation. We evaluate our algorithm in the following subsection and finally we conclude this section with a short summary and discussion its content.

We move to distributed eCP indexing next.

4.2.1 Distributed eCP indexing: Overview

We will start by listing both our assumptions and the problems we face and then we will discuss and come up with a solution to the issues raised.

The assumptions we make are:

1. The size of the collections at web-scale is such that both the raw data and the indexed database will have to reside on secondary storage.
2. No single machine has the storage capacity for web-scale collections. Only a NAS or a DFS have a sufficiently large storage capacity.

3. We will be able to use a large amount (≈100) of multi-core machines with a lot of RAM (but only a few GB of RAM per core).
4. Machines are interconnected via a relatively slow Ethernet network (typically 1Gbps).

And the problems we need to address are the following:

1. One of our main problems is to solve the issue of storage. A centralized storage, like the NAS we used in previous experiments, has the advantage of large capacity but its limited network link will quickly become saturated as we scale to a number of machines. The only realistic alternative is to use a DFS where the whole interconnectivity of the networks can be utilized.
2. We need to have task scheduling and management that keep the available computing power busy.
3. There is also the question of how to detect and re-launch failed tasks and provide fault-tolerance for data.
4. A final consideration is how do we keep down the network communication and overhead such that it does not become a bottleneck.

All of the above issues are addressed in the Hadoop framework and, as we saw in the previous chapter, the eCP indexing algorithm has the advantage of its clustering being fully data independent. The HDFS can solve our storage problem and Hadoop provides fault-tolerance, failure detection and re-start services as well. Last but not least, Hadoop provides a fully automatic distribution and parallel execution of the assignment tasks for us, making sure all the processing power is harnessed. Therefore, the obvious choice for us is to use Hadoop as the platform that we develop our distributed eCP on.

As has been discussed previously, eCP is representative of other state-of-the-art algorithms that rely on unstructured quantization of high-dimensional descriptors, like Video Google [SZ03] and the Product Quantizing NN-search described in [JDS11]. It is our hope that our work in extending eCP for Hadoop can function as a guide for future adaptations of other algorithms. While traditional indexing schemes based on k-means are iterative, eCP is not. Thus, it fits well with the single-pass all-in-one-go behavior of Haddop. Distributing iterative or data-dependent algorithms, like k-means or the NV-Tree, will be more difficult and perhaps a system like Mahout or GrapLab is better suited for such a task.

In Figure 4.2 we have a familiar picture from Chapter 3, where we show the three phases of the eCP indexing algorithm. In the picture we see how the Map-Reduce pipeline can be used to do both phase #2 and #3.

We can also see that phase #1, the building of the index hierarchy, is done outside the Hadoop process, before it starts. In Section 3.2.1 of Chapter 3 we

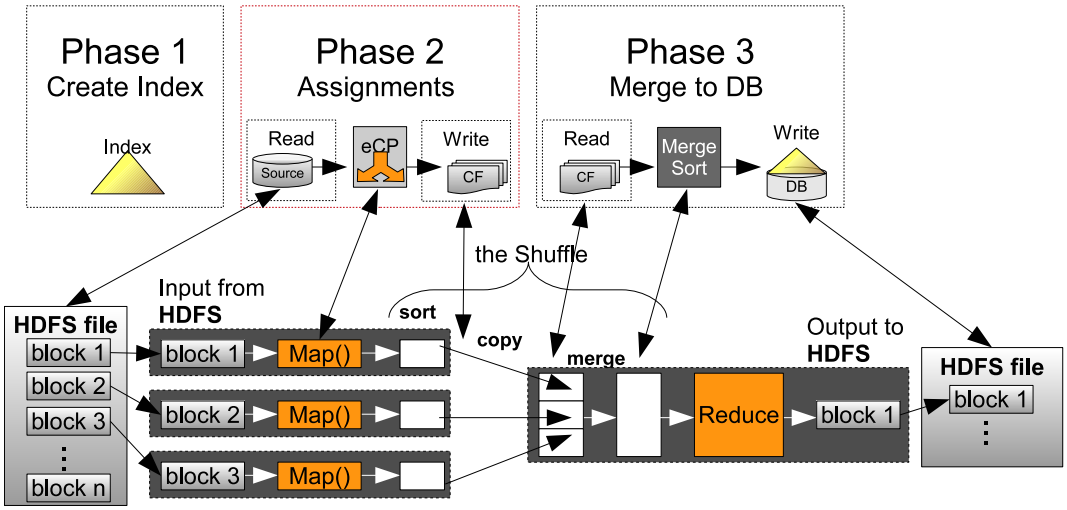

Figure 4.2: Here we see a familiar image of the three phases of eCP indexing and how they fit on the Map-Reduce pipeline. *Map* functions do the assignments of phase #2 and "the Shuffle" does the merge-sort of phase #3 for us. The data, both the raw descriptor collection (source) and the indexed database, is stored on HDFS.

gave justification why we did not parallelize this process and the reasoning for not distributing it are essentially the same.

The CPU intensive phase #2, that we previously parallelized, is now handled by the *Map* function. Unlike the parallelized indexing, the distributed *Map* functions do not share memory (even if they are running on the same machine). Therefore, each *Mapper* needs to load the index from file when it is invoked.

The raw collection of SIFT descriptors is stored on HDFS, where it is split up into blocks of data. For each block of data, a *Map* function will be invoked to process it. Therefore, the logical choice for eCP is to do the assignments in the *Mappers*. Once the *Mapper* has discovered the most similar cluster(s) for a SIFT descriptor, by traversing the index structure, it is emitted from the *Mapper* into "the Shuffle" as a key-value pair. The value is the SIFT descriptor itself (with vector and imageID) and the key is the clusterID of the cluster that the descriptor was assigned to.[4]

As it turns out, the sort-copy-merge process of "the Shuffle" can be used to do phase #3 for us. Each *Reducer* is responsible for a range of clusterIDs and "the Shuffle" will all the hard work for us, namely the sorting of descriptors and grouping them by the key (i.e. the cluster they are assigned to). Therefore, each *Reduce* function only has to do some simple bookkeeping, like count the number

4. If soft-assignment is used (a>1) the *Mapper* simply emits each descriptor a times, each time with a different clusterID as key.

of descriptors in each cluster etc., and pass the key-value pairs on to be written back down to HDFS as the final indexed database.

We should now have an overview of how the distributed eCP algorithm works on Hadoop. In the next section will give a more detailed description on how the distributed eCP indexing algorithm is adapted to and implemented on Hadoop.

4.2.2 Distributed eCP indexing: Implementation details

Despite how nicely the eCP indexing fits the Map-Reduce pipeline, there are several issues that still have to be addressed. We will discuss each problem that we have to solve as they occur as we move over the Map-Reduce pipeline from left to right, starting with the input data residing on HDFS.

4.2.2.1 Binary data and SequenceFiles

The raw collection of descriptors is binary data, 4byte imageID and 128byte descriptor vector. As the HDFS splits the data into fixed size blocks (default setting of 64MB) there is a risk that our binary descriptors may fall on a block-boundary and it data could get split between blocks. We cannot allow this to happen as the reading of the second block would be corrupted.

Hadoop provides special *SequenceFiles* for working with binary data. When using *SequenceFiles*, padding is used in the block-splitting process to protect the block boundaries, making sure that individual binary records are never split between two blocks. A *SequenceFile* consists of a header and at least one key-value pair record. The header contains metadata that HDFS uses to parse the records when they are read (for example by the *Map* functions). A *SequenceFile* record is fixed in size and is defined as a *key-value pair* (i.e. the defined input for *Map*-tasks). *SequenceFiles* also have several features (such as support for block compression and *sync markers* that for seeking to record boundaries) that make them an optimal choice for processing binary data with Hadoop.

We therefore implemented a conversion mechanism to create *SequenceFiles* from our original binary data. In the conversion process, we set the 4 byte imageID as the key and the 128 byte vectors as the value.

4.2.2.2 Indexing in the Map function: I/O access and RAM footprint

As we already described, each time a *Map* function is invoked it has to start by loading the index. There are two potential problems with this. The first is that we frequently load the index from disk and that file may exceed the size of the block of data to process. The second problem is each *Map* function keeps its

own private copy of the index structure. When the index structure is large, there is not enough RAM for all the *Mappers*.

As for the loading of the index structure from disk, we use a functionality of Hadoop called the "distributed cache" that is optimized to deliver large read-only files to all the participating nodes (or machines) before the sub-tasks of the job are executed. This prevents network hot-spots during the execution process as a copy of the file is already stored on the local disk. Also, as each machine is running several *Mappers* and the file with the index structure is read frequently, if there is RAM available, the OS is very likely to cache the file in memory for us.

The second problem is that despite the index being a read-only structure, by default, each *Map* function loads its own copy and thus we need enough RAM to hold as many copies of the index as there are cores in each machine. When the index is large (order of multiple GB) we simply run out of RAM and have to reduce the number of cores we use.

Instead of having an independent *Map*-task-per-core, what we need are multi-threaded *Mappers*. Hadoop actually supports threaded *Map* functions in the form of a class called *MultithreadedMapRunner*, where a pool of threads are used to do work otherwise done by a single thread. The threaded version is still processing only a single block of data, so the number of invocations (i.e. *Map*-tasks) is the same.

While we do solve the RAM limitation issue, and all cores could be fully utilized, we still do not get the decreased running time we were hoping for. The Hadoop version of eCP is written in JAVA and unlike our previous parallelized code, written in C++, we are having problems with parallel scalability. This is due to the overhead of locking mechanisms in JAVA (both in our code as well as in the Hadoop framework). For example, we observed better running times with running two 4-thread *Mappers* than a single 8-thread one, indicating the limitation of thread scalability. Essentially, the threaded-mappers are still a work in progress.

4.2.2.3 Shuffling all the data

As we said before the *Mappers* output key-value pairs that contain all the data, both clusterID (key) and the SIFT descriptors and imageID (value). In the sort-copy-merge process of "the Shuffle" all of this data has to be moved across the network. Hadoop will do its best to fully utilize all the available bandwidth but Hadoop also supports compressing during this transfer. As we are using binary data, the amount of compression is limited, (20-30%) but this is an optimization that is highly recommended for data intensive tasks as it can save both on network load and transfer times.

4.2.2.4 Writing the output data

As we described in Section 4.1.2, HDFS default replication factor is to make 3 copies of each block of data. With rack-awareness enabled and configured (and a Hadoop-cluster that spans more than one rack), one of the replicas is always written to a remote rack. This is part of the fault-tolerance scheme, as this way, an entire rack of machines can be lost without compromising the integrity of the data. However, if the network link(s) between racks is limited, this can significantly delay the time it takes to do the writing. This can delay the completion of Hadoop jobs as the final output is written to HDFS with replication. It is however possible to set the replication factor on a per-file basis. Therefore, it is possible to use one replication factor setting for the output data, for example no-replication (replication factor 1), and another setting for the input file(s). Having a high replication factor for the input data is important because that will benefit Hadoop in finding available cores to run the *Map* functions where the data is located.

4.2.3 Distributed eCP indexing: Experiments

We will now look at the results of our experiments with distributed eCP. First however, we will describe the datasets and index configuration that we will use, and look at the hardware that we have available to us.

4.2.3.1 Datasets, index configuration and the Grid'5000

Dataset information				Index settings			
Set	Images	Descriptors	Size on disk	Clusters	L	Traversing	Index size
1	100M	30.2B	4TB	6M	5	296	1.8GB
2	25M	7.8B	1.0TB	1.5M	4	354	461MB
3	10M	3.3B	0.5TB	652K	4	285	193MB

Table 4.1: Dataset information and index configuration used in the experiments in this chapter.

Dataset Available to us is a very large (web-scale) collection of 30.2 billion SIFT descriptors extracted from a collection of 100 million images. This dataset was provided to us by Exalead as part of the Quaero project.[5] We have in fact already mentioned this dataset in Chapter 3, as the 8.1B descriptor dataset we

5. Quaero is a research and innovation program addressing automatic processing of multimedia and multilingual content.

used in our parallel eCP experiments, see Table 3.1 of Section 3.2.3, was a subset this 30.1B descriptor set.

> **Please note** that unlike the previous chapter, where we referred to the datasets by the index structure used, in this chapter we will refer to the datasets by their size. Each dataset will be indexed and searched with only a single index-structure and therefore the size is a more descriptive name.

The three datasets used in our distributed eCP experiments are described on the left side of Table 4.1. The first set, seen in row 1, is the full collection, while the other two are subsets of it. In row 2 we have a ≈25% subset that contains 7.8B descriptors. This is almost the same size as the 8.1B descriptor set we used before, but because the padding of the *SequenceFiles*, we reduce the number of descriptors to keep the file size below 1TB on disk. As we will discuss shortly, it is not possible to run a lot of experiments with the full set. We therefore need a small set to use for iterative experiments, where we tune parameters etc. In row 3 we have the set that was used for this purpose. It consists of 3.3B descriptors or ≈11% of the full set.

Index configuration For each of the three datasets we define a hierarchical index structure to use. The details of each index structure can be found on the right side of Table 4.1. The cluster *target-size* is 5,000 descriptors for each dataset and for the full set of 30.2B descriptors that results in a index of over 6M clusters on disk and thus the same number of *cluster representatives* on the bottom-level of the L=5 level deep index structure.

While this is not the largest index we have used, the ts=1,000 for the 8.1B set resulted in 8.1M clusters on disk, this is by far the largest amount of data that we will be indexing with such a wide and deep index.

The available hardware: Grid'5000 Grid'5000 [JLLa06] 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. For our experiments we have access to the three clusters of machines that belong to the Rennes-site of Grid'5000. In total the three machine-clusters have 129 machines but there are almost always some machines down due to failures or maintenance. The specifications of the hardware can be found in Table 4.2.

Each node of every cluster is connected to a Cisco Catalyst 6509-E router/switch that has 288 1Gbps Ethernet ports. In addition, the nodes of the Parapluie and Parapide clusters are interconnected with 10Gbps Infiniband connections (but

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

Table 4.2: Hardware specifications for the three machine cluster at the Rennes-site of Grid'5000.

those are not used in our experiments). For permanent storage we have access to a 7TB NAS volume that is connected to the clusters via the 1Gbps Ethernet of the Cisco Catalyst switch.

The Grid'5000 is run on a reservation based system where users reserve hardware ahead of time. There is both a limitation on time and number of reservation possible per-user. During the week-days, day-time reservations are limited to just a few hours at a time. Therefore, the only possible time to run our large-scale experiments on the full collection of data is during the week-ends when it is possible to make long reservations of a large amount of hardware. However, there is competition between users for the long reservation time-slots and each user can only have two active reservations at the same time. This limits our ability to run a large body of full-scale experiments.

4.2.3.2 Experiments and results

The goal of the first set of experiments is to evaluate the scalability of our implementation as we scale-out to using more and more machines from the Grid'5000. For this task we use the 3.3B descriptors set and range the number of machines from 20 to 50 computing nodes (that is 160-400 cores running *Map*-tasks).

In the second set experiments we scale-up, by indexing the 7.8B descriptor set. We also scale-out, by using all the machines available to us, or over 100 machines (that is over 800 cores running *Map*-tasks) from all three machine-clusters of the Rennes-site. [6]

Finally, in the third set of experiments, we scale-up again, this time going all the way and indexing the full 30.2B descriptor set using, again, all the machines available to us.

In all of the following experiments we use 128MB HDFS block size for our files

6. In a large collection of machines, like the Rennes site of Grid'5000 is, there are always a few machines down for maintenance. Therefore, the number of machines available to use may vary by a few machines at any given time.

Scalability of indexing

#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

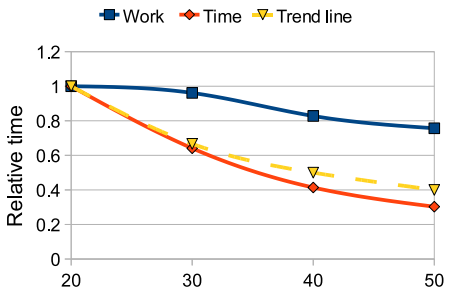

Table 4.3: Scalability experiments using the 3.3B descriptor set and using an L=4 deep index structure with 652K *leaders*. In orange we have wall clock time and in blue we have the amount of work needed (running time × #machines). In the graph on the right, the times are relative to the 20 machine setting and in yellow, we also show a trend line (the baseline setting of 20 machines/ the current # of machines).

stored on HDFS (both input and output). This is twice the size of the default setting of 64MB, but it is the recommended setting for data intensive jobs.

Evaluating the scalability Our first set of experiments is to determine the scalability of our distributed eCP implementation. We use the 3.3B descriptor set, described in row 3 of Table 4.1, and index it using 652K *cluster-representatives* (i.e. *leaders* on the bottom-level of the index), arranged in an L=4 deep hierarchical index structure.

Three machines are dedicated to manage the Hadoop cluster (for the task management and managing the HDFS). We run the indexing on 20-50 machines (*takstrackers*) using machines from the Parадent machine-cluster (see row 1 of Table 4.2). Each machine has 8 cores so we configure Hadoop to allow at most 8 *Mappers* and 2 *Reducers* to run in parallel at any given time on each machine (this is done by assigning 8 *Map*-slots and 2 *Reduce*-slots).

The input data, i.e. the 3.3B set, is stored on the HDFS and, with the 128MB block size, this 0.5TB file is split into 3,478 blocks. With a replication factor of 3, 10,434 blocks are created that require ≈1.3TB of disk space. The number of *Map*-tasks will equal the number of blocks, and thus 3,478 *Mappers* will run during the course of the job, each one loading the 193MB index file from the local disk.

The results of this set of experiments are shown in Table 4.3. We present the reported wall clock time of each job (orange) and we also sum up the total amount of work done (blue) by multiplying the running time with the number of

machines. On the left hand side we have a table with measured running times and on the right we have included a graphical representation of the relative times, using the 20 machine setting as the baseline reference point (1.0). In the graph, we have the number of machines used on the x-axis (from 20 to 50). We have also added a trend line (yellow), i.e. the baseline setting of 20 machines divided with current number of machines used in each setting.

The amount of work is an interesting measure, as it indicates how well we are scaling-out to using more and more hardware. If we are scaling linearly, the amount of total amount of work should remain the same (regardless of the number of machines used). If, on the other hand, we are have problems with scaling, the total amount of work should increase as we use more and more machines.

To our surprise, the results show that we are actually doing better than a linear scale-out, as the total amount of work is actually reducing as we use more machines. There seems to be some economy of scale going on and we think that there are several factors that are contributing to this phenomenon.

We should start by pointing out that we have more and more machines but the amount of data is always the same. The first factor is that with more machines, we also have more RAM to do caching, there is more overall bandwidth in the network etc. Any time wasted on waiting for the few and slow I/O devices (both network and disk) should therefore be reduced.

Another factor may be the length of the sequence of tasks that need to be done by each core. We can calculate how many *Map*-tasks each core will have to do (this is also referred to as the number of "waves" of *Map*-tasks). With 20 machines, each machine will have to process 174 blocks, and with 8 cores, that is a sequence of ≈22 blocks per core. With 40 machines, the length of this sequence is twice as short, only ≈11 blocks of data per core. Each machine will have to sort and cache less data before it is delivered to the copy-phase of "the Shuffle", this decreased the need to write intermediate data to disk and thus also to read it back from disk when the *Reducer* is invoked. Also, if there is a fixed overhead of doing the I/Os needed for each *Map*-task (i.e. reading the block of data and loading the index file), a shorter sequence could affect the total running time.

Other factors that may contribute are: that "speculative execution" is perhaps more effective when we have more machines available; as there are more *Reduce*-slots available, that may incline the *jobtracker* to start the *Reducers* earlier and/or more frequently and that can reduce the pressure on both disk and network use.

Scaling-up and -out, 7.8B descriptors indexed on > 100 machines In the second set of experiments we index the 7.8B descriptor subset that requires almost 1TB of storage space. On HDFS, with the 128MB HDFS block size, we get 8,178 blocks/*Map*-tasks to run and with replication factor 3, we get in total 24,534 blocks that requires almost 3TB of HDFS disk space.

We extended the deployment setup, using 57 nodes from Parадent, 15 from Parapide and 36 from Parapluie (see Table 4.2), for a total of 108 machines. Here again, 3 machines are dedicated to managing the system, leaving 105 *tasktrackers* to do the work. As before, each machine is configured for 8 *Map*-slots and 2 *Reduce*-slots.

Aside the obvious differences in the hardware and the size of the data set used here, we must highlight a key difference between this set of experiments and the previous one. This time, the number of *leaders* and the size of the index hierarchy is much larger, 461MB index file instead of the previous 193MB. The ratio between the HDFS block size and the size of the index is getting much larger and potentially, each *Mapper* will take longer to read the index file from the local disk and load the index hierarchy into memory.

Also, with the increased index size, each *Mapper* requires more RAM, leaving less of it available for the Hadoop framework and the operating system (for example to cache the index file for us). As we are using the same index depth L, the larger index also means that doing the assignments is more costly as traversing the index simply involves more distance calculations.

On top of that, the imbalance in the size of clusters in the index hierarchy is also a contributing factor for added index traversal overhead, especially in large and deep index structures. We should therefore not be surprised if the overall work needed for clustering the 7.8B subset is greater than what we observed for the 3.3B subset.

The result of indexing the 7.8B subset, using 105 machines (or 840 cores) is that it took 71 minutes to complete the indexing job. The total amount of work done was 7,455 minutes.

Compared to the previous set of experiments, using the 3.3B subset, the 7.8B subset is 2.36 times larger and the index structure is more the twice as wide (1.5M vs. 652K). To get some point of reference, we can compare the amount of work it took to do the indexing of the 7.8B set vs. the 3.3B set. The ratio of how much more work was needed to index the larger set ranges from 2.50 times more (20 machine setting) to 3.30 times more (50 machine setting). Overall, we cannot say that indexing the 7.8B set is significantly worse, even if the index structure requires 461MB of RAM and has to be loaded 8,178 times (once for each *Map*-task).

Indexing at web-scale In this third set of experiments, we take the scale-up all the way and index the full set of 30.2 billion SIFT descriptors, extracted from 100 million images. As before, we use all the machines available to us.

The eCP index structure for this task contains over 6 million *cluster representatives* at the bottom-level of the index and the L=5 level deep index structure

requires roughly 1.8GB of space, both in RAM and on disk (in the form of the index file each *Map*-task has to load).

This index structure is in fact so large that we are forced to reduce the number of *Map*-slots per machine by half, as there is not enough RAM to run both Hadoop and load 8 instances of the index. We therefore set the number of *Map*-slots per machine to only 4 and our 105 machine Hadoop cluster can now only run 420 *Map*-tasks in parallel, instead the 840 parallel *Map*-tasks we could run previously.

This limitation is the main motivation for our work on implementing a multi-threaded *Map* function, such that each core does not need its own copy of the index structure.

With the settings described above, it took about 600 minutes (10 hours) to run the job and cluster 30.2B descriptors. We calculate also the work done by all the machines, 600min. $\times$ 105 = 63,000min. Again, it is hard to compare the task of indexing the various datasets. To get some comparison we can again compare the amount of work done.

The full dataset, 30.2B descriptors, is 3.85 times larger than the 7.8B subset. Also, the 6M index structure is vastly larger than the 1.6M index structure used for the 7.8B or 3.75 times more clusters on disk. Indexing the full set took 8.45 times more work then we observed in indexing the 7.8B descriptor subset using the 1.5M index structure. Because we sum up the work done by multiplying the measured running time by the number of machines (not the cores) we should take into account that each machine only had half the number of cores available. When we do this, the ratio between indexing the full set vs. the 7.8B subset is to 4.23 times more work. The difference between the size ratio of 3.85 and the work ratio of 4.23 is tells us that indexing with the L=5 deep index of 6M clusters is only 0.38 (4.23-3.85) times harder than indexing with the L=4 deep index of 1.5M clusters. Overall, we are quite happy with this result.

A careful analysis of the logs show that 99% of the reduce tasks where completed after 520 minutes, and the remaining 1% took an additional 80 minutes to complete. The reason behind this behavior is in part the uneven size distribution of clusters. But the main explanation is to be found in the content of the dataset. Upon further analysis of the data collection, we discovered that it contains hundred thousands of identical distracting images that turn out to come from a set of explicit web sites that all have different URLs but are redirecting to (or have duplicates of) the same images. This is unfortunate, but it is a good example of what happens in the real world when indexing images collected of the Internet. It would have been possible to filter these images out, but this would have required a specific ad hoc process that we may very well integrate in the future. The direct impact of so many 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 is what is causes the last 1% reduction taskt to take

so long.

It is also interesting to analyze a bit the loading of the index structure, as seen from the machine level. The input data for the 30.2B descriptor set consists of 4TB of of data that is split into over 32,000 128MB blocks in HDFS. That also means, as we know, that over 32,000 *Map*-tasks will be run by our 105 machines. Each *Map*-task must start by loading the 1.8GB index file from the local disk, and thus the index is loaded in total more than 32,000 times during the course of the job. On average, the 1.8GB index file is loaded 305 times per machine, or 76 times per core. Thus, on average, the index file is read from the local disk of each machine every 118 seconds.

If the local disk can be read at 70MB/sec. (this is probably overly generous) it would take about 26 seconds to sequentially read the file from disk, once. Doing it 305 times might take as much as over 8,000 seconds. Clearly, it is of great importance to leave some memory available for the OS to cache our index file, such that it must not be read from disk every time a *Map*-task is invoked. This is yet another strong motivation to reduce the memory footprint of our algorithm by implementing a multi-threaded *Map* function.

Larger HDFS block size In the experiments so far, we have been using the recommended setting of 128MB HDFS blocks. When the index file is very large, intuitively, we would like to make the block size larger, such that each mapper would process more data for every time the index had to be loaded and also to reduce the total number of times the index has to be loaded from disk.

We ran some preliminary experiments where used larger HDFS blocks. The settings and dataset are the same as in our second set of experiments, the "Scaling-up and -out...", i.e. the 7.8B subset and 1.5M index.[7] We increased however the HDFS block size to 512MB.

The indexing took 69 minutes and the amount of work done was 7,314 minutes of work, compared to 71min. and 7,455min. of work before. I.e. there was no significant gain observed.

We also indexed the full 30.2B set, using the same settings for that set (6M index and 4 *Map*-slots etc.) as we used before, except we use 512MB blocks on HDFS.

This time it took 507 minutes or 53,742 minutes of work. While this is a improvement over our previous 600min./63,000min. result, we also did other optimizations of Hadoop settings as well. For example, we increased the *Map-side* sort buffer and we used compression in the shuffling process. Both optimizations

7. This time, 1 extra machine was available, giving us 106 instead of 105.

of these settings should facilitate the issue we observed with the 1% last reducers taking such a long time. Perhaps a more fair comparison would be to the 520min./54,600min. mark when the experiment using the 128MB block size was 99% done.

In any case, and to our surprise, increasing the block size did not provide the improvement we had hoped for. For the smaller index (461MB), the OS file caching could simply be so good that loading the index is not a big factor, but for the 1.8GB index of the full set we were expecting a more significant difference. Perhaps, because we reduced the *Map*-slots to only 4, there is sufficient RAM available to OS to cache even the 1.8GB index file. A deeper analysis of what is going on is necessary, but it is not easy to monitor and analyze such a long a widely distributed task.

In our discussion and analysis of the distributed batch search, later in this chapter, we will encounter a very similar problem. For now, we will leave this as an unsolved mystery.

4.2.4 Distributed eCP indexing: Summary and discussion

There is a myriad of additional indexing experiments that we would have liked to do, especially on the web-scale, 30.2B descriptor, dataset. However, as we described in Section 4.2.3.1, experiments that require long reservations of Grid'5000 hardware are limited to run only in week-ends. In addition, we are not the only researchers that want or reserve the hardware of the Grid'5000 for long periods of time and thus there where only a few week-ends that all the machines of the Rennes-site where available to us.

Running the 30.2B set with a smaller number of machines was not feasible due to both disk space limitations, we need a minimum of 16TB of HDFS disk space, and because even the maximum week-end reservation time would not suffice to run the full scale job with only a few machines. This is not only because the added job running time, but because we need a lot of time to deploy the Hadoop cluster and get the 4TB of input data from the NAS, our permanent storage, to the HDFS via the limited network link between our NAS and the machine clusters. This 4TB transfer takes about 8 hours, one way.

Essentially, we can say that running the experiments on Hadoop was the easy part of our work, it is all the "everything else" that is making things complicated.

In our first set of experiments, and as we can see in Table 4.3, we showed that our distributed eCP can scale-out, to use more hardware, very well. In fact, we were surprised by the fact that the total amount of work necessary is reduced as we use more hardware and the response time is actually below the reference trend line (see graph on right hand side of Table 4.3).

We also showed that our distributed eCP can scale-up to index larger datasets. First by indexing the 7.8B set on 105 machines, using 840 cores. And then by indexing the full set of 30.2B descriptors on 105 machines, using 420 cores. When we indexed the full set, with the 1.8GB index structure, we did run into a scale-out problem. As we have limited amount of RAM per machine, and in this version of distributed eCP, each *Mapper* needs to load an instance of the index structure, we had to reduce the number of *Map*-slots by half, from 8 to 4.

However, as we discussed in Section 4.2.2.2, we are working on a multi-threaded *Map* function that does the descriptor assignments with multiple cores but using only a single instance of the index structure. This has proved to be a non-trivial task as unlike the parallelized eCP that we developed in Chapter 3, the multi-threaded implementation of distributed eCP in JAVA, and the threaded integration with the Hadoop framework, is not giving the same good parallel scalability as we got before. Using multi-threaded *Mappers* does however solve our scale-out issue and and getting better performance out of the multi-threded JAVA implementation is a work in progress.

We will now change the topic to searching and develop a distributed version of the batch search that can run on Hadoop.

4.3 Distributed batch search

In Section 3.3 of Chapter 3 we introduced a batch search process that sacrifices individual query-image response time for throughput of the entire batch. We showed how large batches, tens- or hundreds of thousands of images, shift the search process from being bound on random I/O toward sequential I/O and using ever more computing power. The largest batches of 100,000 unique query images, see Figure 3.8 in Section 3.3, was right at the brink of using all the 24 logical cores (12 real + 12 HT) of the machine we used for the experiments. Searching a larger batch on that particular database, using those settings, would need more processing power to keep reducing the average response time per image. Also, even if the average response time per image of parallel eCP's batch search is low, the total running time of the large batches is still high. For example, the 100,000 image batch that we see in Figure 3.8 took 12,123 seconds using b=1 and 16,443 seconds using b=3.

It is therefore motivating to harness the power of distributed computing to do batch searching.

In this section we will start with giving an overview of how we implement distributed eCP batch search on Hadoop, followed by a section with implementation details and a discussion on the specific problems of adapting the batch search process to the Map-Reduce paradigm.

We then shift the focus on to evaluating the distributed eCP batch search. The quality of web-scale search is evaluated by searching the 49K query set and the CopyDays query set, the same image query sets that we used in Chapter 3, on the 30.2B descriptor database that we built in our distributed indexing experiments (see Table 4.1). We also evaluate the distributed search process on Hadoop, where we try to gage the scalability, efficiency and identify limiting factors in running on the Map-Reduce paradigm.

We conclude this section with a summary and discussion on the topics covered.

4.3.1 Distributed batch search: Overview

Hadoop, being a batch processing framework, requires significant amount of time just to launch a job and is therefore never going to be good for answering individual image queries. It is however well suited for processing massive tasks that require a lot of data and processing power. As we saw in Chapter 3, an image query batch typically consists of 10^4–10^7 query descriptors and searching a web-scale database, that consists of several Terabytes of data, requires both the reading of a lot of data and a lot of CPU power to do the numerous distance calculations. Searching large batches of query images, on web-scale databases, should therefore be a prime task adapt to Map-Reduce and run on Hadoop.

Already in Section 2.3.4 of Chapter 2, where we discussed image-level optimizations for the search process, we defined an order for the process of searching. The first step is to discover all the cluster requests, such that redundant requests can be merged and the data on disk can be processed in order. While this gave limited benefit for an individual query, this is the key idea behind the batch search process we developed in Section 3.3 of Chapter 3.

In the Map-Reduced batch search we follow the same execution order and in fact we discover all the cluster requests for all the query descriptors outside the Map-Reduce pipeline. The result from this process is stored as a lookup-table where each query descriptor that requests a specific clusterID can quickly be looked up and retrieved.

In Figure 4.3 we can see how we implement the batch search process on the Map-Reduce pipeline. The input data for the batch search pipeline is the indexed data of the database. The database is stored as *SequenceFiles* on HDFS, where the content is stored as key-value pairs in HDFS data blocks. The key is the clusterID and the value is the SIFT descriptor (4byte imageID + 128byte vector).

The *Map* functions is tasked with populating the query descriptors k-nn by scanning the cluster content that is read from the HDFS data block. However, the *Map* function also needs access to the query descriptors and to know what cluster(s) each descriptor should be scanned against. We solve this by first loading

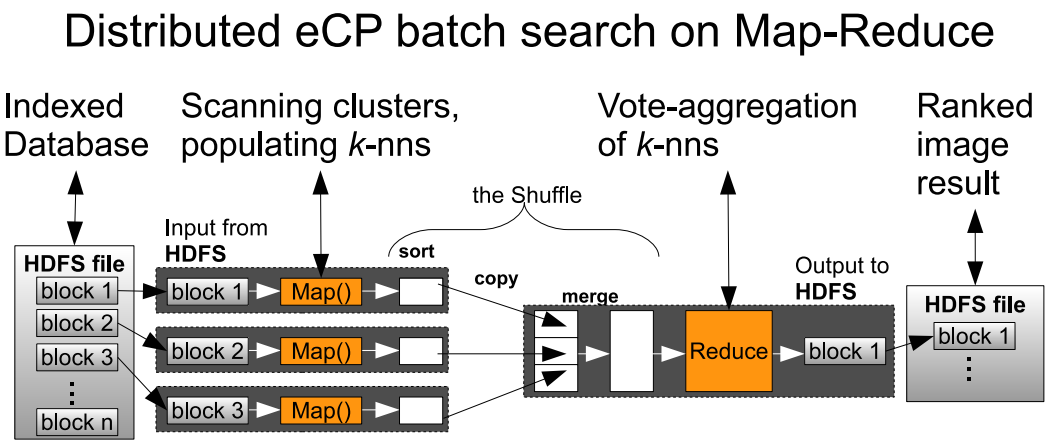

Figure 4.3: Here we see how the distributed eCP batch search can be adapted to the Map-Reduce pipeline. Outside the pipeline, a lookup-table is created that contains query descriptors and the clusters they request for scanning. The *Map* functions uses the lookup-table, populates the k-nns by scanning the clusters in its block of data and emitting the completed k-nns. "The Shuffle" groups k-nns of each query image for us (the query-imageID is used as the key) and the *Reduce* function does the vote-aggregation, creating the final ranked image-level result.

the lookup-table that was created a priori. The process of loading this lookup-table is very similar to the way we load the index structure in the distributed indexing process. When the k-nn has been populated, the *Mapper* emits it as a key-value pair, using the query-imageID as the key and the k-nn as the value.

The shuffle process (sort-copy-merge) picks up all the k-nns from all the *mappers*, groups them by query-imageID for us, and passes them on for *Reduction*. The *Reduce* functions task is then to take the data from "the Shuffle" and do the vote-aggregation for those images it is responsible for and produce the final ranked image-level results.

4.3.2 Distributed batch search: Implementation details

The Map-Reduce version of batch searching does not use the index structure nor does it need the file offset information that was previously used to retrieve individual clusters from disk. This is because the distributed part only scans the clusters and does the vote-aggregation. The index traversal is done in an a priori step that creates a lookup-table, mapping clusterIDs to the requesting query descriptors. As for the information about where the clusters are on disk, this information is not needed. All the data will be read by the Map-Reduce process, regardless of cluster requests, as that is what the Map-Reduce paradigm

is optimized for. What we do is simply to keep the clusterID in the database data, where we use it as the key for the key-value pairs that will be read and passed to the *Map* functions.

This would be a major issue if we intended to search small batches, but we are not building such a system. We are building a system for processing very large batches, where most, if not all, the clusters will be relevant to some query descriptor (we can see this in the two "Q-Desc. overlap" columns in Table 3.4 of Section 3.3.3 from Chapter 3).

As we saw in the overview section, we load the lookup-table in the beginning each *Map* function from the *distributed cache* (i.e. reading the local disk), much like we loaded the index structure in the *Map* function in the distributed eCP indexing. There is a risk that we may run into the same scalability issues as we had with the indexing *Mappers*. Also, the loading large files for each invocation of a *Map* function could be a potential bottleneck.

In our first implementation, that we will be testing, there is a single large lookup-table loaded by each *Mapper*. As the key-value pairs are processed by the *Mapper* (reading the block of data one pair at a time), the key (i.e. the clusterID) is used to look-up the queries in the lookup-table and those query descriptors k-nns are populated.

However, unlike the indexing process, the whole lookup-table is not needed by every *Mapper*. Only the information for those clusters that reside in the *Mappers* block of data will be needed by that particular *Mapper*. The problem is to know what clusters are in each block of data. If we make sure to keep all the clusters of the database HDFS files in ascending (or descending) order (by clusterID), it becomes sufficient for the *Mapper* to check the first key-value pair. From that clusterID of the first key, the *Mapper* can know the range of clusters it can expect to see in its block of data and thus only load the relevant subset of the full lookup-table. We are still working on such a version of the distributed eCP search.

We should also note that a pre-processing step is needed in our *Reduce* functions, that merges multiple k-nns for the same query descriptor, before the vote-aggregation process. This is because some clusters may fall across HDFS block boundaries (i.e. some of the descriptors are at the end of one block, while the rest are at the beginning of another) and then two half completed k-nns will be created by two separate *Map*-tasks. Also, if $b>1$ is used, this is bound to happen frequently.

For this reason, our *Mappers* should include in the emitted k-nns both the distance values to the neighbors and information that uniquely identifies the query descriptor the k-nn is based on. This allows the *Reducers* to quickly identify multiple k-nns for the same query descriptor and to merge them with minimal effort.

4.3.3 Distributed batch search: Experiments and results

We start by evaluating the quality of the large-scale databases that we built with the distributed eCP algorithm (see Table 4.1). For this purpose we search it with the CopyDays (3055 query images) and 49K (48,883 query images) query sets that we also used in Section 3.3.3 of Chapter 3. We then evaluate the efficiency by increasing the number of query images in the batch.

Search quality In this set of experiments we deploy our Hadoop cluster on 108 machines from the Rennes-site of the Grid'5000. Three machines are reserved for cluster management and thus 105 machines are running *tasktrackers* and each such machine is configured to have 8 *Map*-slots and 2 **Reduce**-slots (same as we did for the indexing experiments).

In first experiment we use the CopyDays query set and search both the 7.8B and 30.2B databases. The databases are stored on HDFS where they are split into 128MB blocks.

The quality is shown in Figure 4.4, where we have the average quality by variant as well as the average score (on the far right). As before, we only consider a "correct match" for the query image when the original image is the top-voted database image in the ranked result.

As we can see, there is almost no difference between searching the full 30.2B (green line) sized set and the 7.8B (red line) subset. On the far right we have the overall results, averaged over all the variants, for both sets. For the 7.8B set it is 82.68% and 82.16% for the full 30.2B set. We can also see that eCP returns high quality results, except for some severely attacked images, such as when 80% of the original image is cropped and then it is rescaled to its original size, or when strong manual variants are applied.

We also evaluated the quality for the 49K query set on the 30.2B database. Overall we had a P@1/R@1 score of 91.65% using b=1. This is only a few percent lower than we saw in Chapter 3, see Table 3.3 of Section 3.3.3. The difference is that this time the size of the database is more than 3.7 times larger.

The Hadoop job of searching of all the CopyDays query images (955K query descriptors) in a single batch on the 30.2B database took 1,623 seconds, or 531 milliseconds per image, and on the 7.8B database it took 388 seconds, or 127 milliseconds. It is interesting to note that the same batch size running on our parallelized C++ eCP batch search and searching the similar sized 1.6M database (8.1B descriptors and same ts) took 2,977 seconds, or 974 milliseconds per image. Even with the overhead of launching the Hadoop job etc., a batch of "only" 3055 images (or less than 1M descriptors) is still giving almost twice the throughput than we got running on a single multi-core machine with very high-end NAS disks

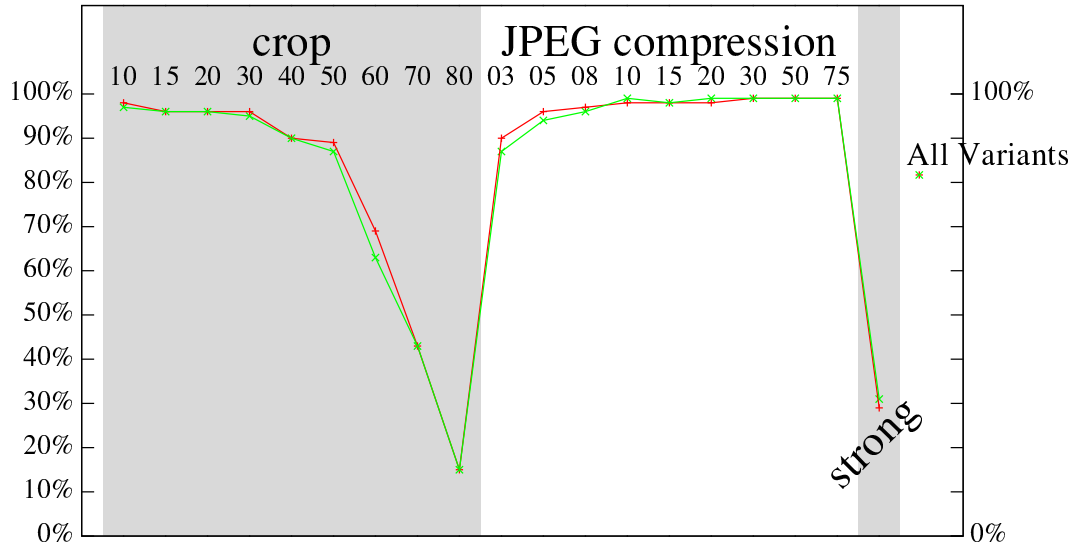

Figure 4.4: Here we have the quality results by variant for the CopyDays set, searching both the 30.2B (green) and the 7.8B (red) descriptors databases. As we can see, there is almost no difference between searching the full 30.2B sized set and the 7.8B subset. On the far right we have the overall results, averaged over all the variants, for both sets. For the 7.8B set it is 82.68% and 82.16% for the full 30.2B set.

organized in a RAID5 disk volume. On the other hand, the distributed Hadoop cluster does consist of over 100 machines, 100 hard drives and over 800 cores.

Large batch size Because we are loading the entire lookup-table in each *Map*-task, we are limited to a maximum batch size of 12,000 images, and even with that setting we had to reduce the number of *Map*-slots per machine by 2, from 8 to 6. Running the batch sizes we are designing the system for will have to wait until we solve the issues with partial loading of the lookup-table.

For this task we had available 90 machines from the Rennes-site of Grid'5000, 3 for managing Hadoop and 87 *tasktrackers*. Each *tasktracker* is configured with just 6 *Map*-Slots so we are using 522 CPU cores. We have also increased the HDFS block size of the input data (i.e. the database) to 256MB.

As we have fewer machines and different settings, we re-run as well the 3053 images of the CopyDays set for comparison.

Even though we have 18 machines less than before, the running time for the 3K images took 1,405 seconds, or 218 seconds less. The main reason for this difference is the use of larger HDFS blocks. It is more efficient as fewer *Map*-tasks are invoked and thus the lookup-table has to be loaded fewer times. Also,

using only 6 cores may cause less competition for the disk access and we also made some tweaking and tuning of other Hadoop settings, to optimize the performance between the initial experiment and this one.

The running time for the 12,000 batch size was 2.520 seconds. If we look at the average time per image, the 3K batch took 460ms. while the 12K batch took only 210ms.

Even if we are not able to do the really large batches, we already have the evidence for the potential running distributed eCP on Hadoop. We can look at the running times of the parallel eCP batch search for comparison. While we do not have a 12,000 image batch, we can compare to the 10,000 image batch running on the 1.6M database (that is most similar to our current settings). For the 49K set, the 10,000 batch size was running at 430ms. per-image and with the unique image set the average time per-image was at 490ms. The distributed eCP, running on 90 Grid'5000 machines and the 30.2B database requires only half that time, or 210ms.

With a better implementation of the eCP batch search, which does not suffer from memory limitations and can run the batch sizes that it was initially designed to do; we expect vastly improved throughput.

4.3.4 Distributed batch search: Summary and discussion

In this section we have shown that eCP can provide very good search quality at web-scale.

As part of the Quaero Project, we had to have an external evaluation of our results. The evaluation is based on both the 49K and CopyDays query sets. Of the three contributing partners to our sub-task, we had the highest precision/recall and mAP results. For the full scale 30.2B descriptor set, our mAP was 92.61% and we had a P@1/R@1 score of 91.44%. For Recall@10 our score had increased to 94.80%.

The throughput results of large batch size look promising, but the current implementation of our distributed eCP batch search is severely limited by not being able to process a batch larger than 12,000 images. Yet, searching a 3.72 times larger database using 90 Grid'5000 machines is already twice as fast as the results we observed when doing our experiments wtih the parallelized eCP batch search on the very powerful single multi-core server with top-of-the-line SAS disks in a RAID5 volume and a dedicated high-end RAID-controller.

4.4 Discussion

In this chapter we have developed a distributed version of both the eCP indexing algorithm and the eCP batch search algorithm. In both cases we adapt the algorithm to the Map-Reduce programming paradigm and we have evaluated them by doing experiments with a Hadoop cluster of over 100 machines, using over 800 CPU cores.

We indexed the web-scale dataset of 30.2 billion SIFT descriptors extracted from over 100M images using very similar settings as we did when we indexed the 8.1 billion descriptor subset of this collection back in Chapter 3. The *target-size* of clusters is only 5,000 descriptors (665KB on disk), creating 6,052,988 clusters on disk. The indexing with 100 machines took 10 hours. We also evaluated the quality of this database by searching both the 3055 query images of the CopyDays set and the 48,883 query images of the 49K set. Our quality results were also evaluated by a third party as part of the compliance with our participation in the Quaero project. Overall we had a 91.44% P@1/R@1, the best quality of the three contributing partners.

This results shows the eCP algorithms has the ability to cope with very large scale datasets and yet deliver very high quality results.

Regarding distributed eCP's ability to scale-out, i.e. to using more hardware, we showed in Figure 4.3 that the indexing scales even better then we expected. However, there is a limitation in our current implementation regarding the memory-footprint of the *Map* function, both for indexing and search. This limitation is caused by the auxiliary data that we need to load. In the indexing this data is the index-structure and in the batch search it is the lookup-table that tells us what query descriptors want to scan what clusters. However, in both cases we have proposed ways to adapt our algorithms to avoid these issues.

For the indexing algorithm we are working on a multi-threaded *Map* function that will eliminate the need to load an instance of the index for every core, just to keep all cores busy. We have already implemented a working version, but the efficiency of this version is still not adequate.

For the batch search algorithm, a multi-threaded *Mapper* will not be a sufficient as a long term solution. In large batches there are simply many query descriptors and even a single *Map*-task per machine will run out of RAM and thus a limit will be on the scalability. However, it is not necessary to load the entire table, as there are only so many clusters in each block of data. Once the *Mapper* reads the first entry, and knows what range of clusters he can expect from a single block of data, only the relevant parts of the lookup-table can be loaded and thus only a fraction of the RAM is needed.

Lessons we can draw from working with Hadoop on Grid'5000 In our discussion so far, we have focused on the distributed eCP algorithm, its evaluation and results. We would also take some time to discuss various lessons that we draw from our work with the Grid'5000 and the running of a Hadoop cluster.

Lesson #1. There is a potential for degrading performance when two conditions are met: (i) the input data occupies *many* HDFS blocks and hence, many *Map*-tasks will have to be run; And (ii), each *Map*-tasks needs to load large files from disk with auxiliary information at startup time (like we do with the index structure). It is key to reduce as much as possible the overhead payed by each *Map*-task at invocation time. We do this by using the *distributed cache*, reading data from the local disk, and leaving enough RAM available for the OS to cache the file(s) for us.

Another 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, i.e. above maximum recommended settings of 128MB. Setting this to 512MB or few even a few GB will in turn reduces the number of *Map*-tasks to invoke and thus potentially reduce the proportion of the time wasted when each *Map*-task starts. Note, however, that using large block sizes may effect other aspects of the Map-Reduce pipeline, some of which are not under our control. For example, the Shuffle process stores and copies data in blocks as well. 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 during deployment.

Lesson #2. By default, Hadoop's *Map*-tasks are completely independent of each other and do not support the sharing of resources like memory. Therefore we need to load the index structure in each one during our indexing. When this auxiliary information is large, each *Map*-task will consume a significant portion of the available RAM. This in turn can lead to us having to limit the number of *Map*-tasks, as to not exceed the available RAM, and thus also we are giving up part of the computing power that is available to us.

It is unfortunate to have to waste some of the processing power by leaving cores idle, all because there is no good way to share data between *Mappers* running on the same machine (especially *read-only data* like our index structure is). One possible solution, if the *Map*-tasks job is CPU intensive enough, is to use multi-threaded *Mappers*. This is way more complicated to implement but it is one option for using all the processing power of machine while circumventing Hadoop's (and JAVA's) inflexible architecture. However, as we have experienced, implementing an efficient multi-threaded *Map* function is no trivial task.

Lesson #3. In our experiments, we have observed that most of the *Mappers* read the data locally, only about 1% of reads are remote (i.e. the block of data to process is read over the network from a different machine). The level of replication is a factor in this, and we experimented with various replication settings. The conclusion of those experiments was that having no replication was a little worse (roughly 8 to 10% remote reads) but we did not observe any significant advantage of having the replication factor set higher than 2.

Rack awareness is also relevant to remote reads as with rack awareness, *Mappers* can be spawned as close to the data, preventing unnecessary network load over limited network links.[8] Rack awareness and replication are good for performance, and not only for coping with failures. We would also like to note that we did experience several nodes failures during our experiments and happily observed that Hadoop re-ran tasks, eventually completing all the runs and the job.

Lesson #4. We have also tried re-blocking the database before running searches. By re-blocking the indexed database files (that we use as input for the search jobs), we can evaluate the impact of using larger HDFS blocks during the search. This is an alternative method to our idea of "partial loading of the lookup-table" that was easy to run as it does not requirer any additional coding (re-blocking HDFS files can be done with a Hadoop command-line command and can be done on a per-file basis). The goal of this idea is to minimize the number of times we have to load the auxiliary by having fewer *Map*-tasks invoked during the search job and this is achieved by setting the HDFS block size to a larger value. Please note that this is essentially the only way we have to influence the number of *Map*-tasks Hadoop will invoke.

We tried this with our indexing code with inconclusive results, we did not observe a significant improvement. However, we also tested this on the searching jobs. When we use a block size of 512MB, we observed a significant reduction in response time for the search of a batch of 12,000 images where the roughly 3.7M query descriptors result in a 1GB lookup-table.

As we are searching with 8 *Map*-slots, the machines with the least amount of RAM per-core may have enough RAM available for the OS to cache the 1GB file. Therefore, when we load the lookup-table, we are frequently having to actually read it from the local disk and we are simply I/O bound on reading this file instead of the input data. An alternative to this approach would be to make sure the file is in RAM when it is accessed by writing it to RAM (like /dev/shm on Linux). On the other hand, that could also cause our system to start thrashing (due to the lack of RAM) or simply crashing with "out of memory errors".

The lesson is thus: if your task is under unavoidable RAM pressure because

8. However, all write operations are more expensive with rack awareness enabled.

of auxiliary data that has to be loaded from the local disk in each *Map*-task, increasing the HDFS block size of your input data can at least mitigate the problem by reducing the number of *Mappers* that are invoked.

Lesson #5. As we discussed in Chapter 3, the batch search process is dominated by I/Os until the batch has enough points to keep the CPU extremely busy doing distance calculations. Hadoop's architecture is such that all the blocks of data are read every time, but this time by multiple machines in parallel. Therefore, the search jobs take almost equally long regardless of the runs batch size (while the batch is small and we are not running into the RAM shortage issues). Using the 7.8B descriptor database and 512MB blocks, we have ran batches containing only one query image (312 query descriptors) to create a minimal baseline of how long it takes just to read the data. This took 323 seconds. With all the CopyDays images in a batch (955K query descriptors) the running time was 388 seconds. Thus, the additional 3052 query images (and almost all of the 955K descriptors) only added 65 seconds to the running time, and probably a good proportion of that time was spent on loading the much bigger auxiliary lookup-table file from disk. Much like we saw in the batch searching of Chapter 3, we think the CPU work is entirely hidden by the cost the I/Os.

Chapter 5

Conclusions

In this thesis we have developed the extended ClusterPruning algorithm or eCP. We have come a long way from the initial implementation, which could handle only a few thousand images and used only a single CPU core. But the basic and simple ideas behind the process remain unchanged.

Contribution

Our main contribution has been the development and evaluation of the eCP algorithm in the context of content based image retrieval, at very large scale. Both in terms of scaling-up, to index and search very large datasets (100 million images or 30.2 billion descriptors), and in scaling-out, both with a parallelized C++ version that can run on a powerful multi core machines and with a Map-Reduce version that can run in large distributed environments.

We also proposed a batch search process that sacrifices response time for vastly improved throughput. With the parallelized eCP batch search we experimented with batches of up to 100,000 images (31.2M descriptors) and observed average processing time per image of less than 100ms. The distributed eCP batch search looks very promising, but it true potential will not be known until we have implement and evaluated the "partial-loading of the lookup-table" variant that we have proposed and described.

We have also discussed, and gave examples, of how eCP can be easily altered to mimic the behavior of other established algorithms like BoF and the NV-tree. It is therefore our opinion that eCP is a prototypical clustering-based CBIR algorithm.

eCP will work equally well on large datasets as on small datasets. The ideas

behind it are simple and it is fairly easy to replicate and alter at will.[1] We use Euclidean distance, but this can easily be changed to any distance metric by changing a single function. It can also be applied to any high-dimensional problem, even in metric-space (as we do not use calculated centroids etc.).[2]

Therefore, we think it could make very good platform for research and development on CBIR related topics and even in other fields that have to tend with high-dimensional vectors. All the descriptors (vectors) are kept and k-nns return actual distance values. This makes it easy to monitor, evaluate and track any changes that are made either in images, index structure, clusters or any step in between.

It can also be used in the classroom for teaching purposes or as student projects. That is after all how this whole story started.

Lessons and conclusions

We have showed that eCP algorithm can cope very well with large datasets while still giving very good image-level search quality.

As part of the Quaero Project, the participating labs submitted their results to be evaluated by an external 3^{rd} party. The evaluation was done on both the 49K and the CopyDays image query sets, searching an indexed database of all 30.2B descriptors. We submitted the results for the 6M index described Table 4.1, see Section 4.2.3.1 in Chapter 4.

Of the three partners that participated, we had the highest search quality. Our mean average precession (mAP) score was 93.2% for the 49K set, 83.93% for CopyDays and 92.61% over both sets. The P@1/R@1 scores were 92.07% on the 49K set, 82.16% on CopyDays and 91.44% for both sets. Thus, almost all of our mAP score comes from the correct original image being the top voted one.

When we take into consideration that the images are only 150px. on the wider edge, how few descriptors are sometimes extracted from each image and the severity of some of our alterations, the search is no easy task.

The index structure we use creates 6M clusters on disk, of which we will only check one per query descriptor. The cluster *target-size* is 5,000 descriptors, so out of the 30.2B descriptors in the dataset, only 5,000 are scanned per query descriptor. The index structure is also 5 levels deep and traversing it requires an estimated 296 distance calculations. However, only 23 of those are actually scanning *cluster-representatives*, the other 273 distance calculations are to traverse the 4 upper levels of the index. I.e. out of the 6,052,988 clusters, only 23 of them

1. We have for example not used weighted voting nor have we used post processing steps like geometrical verification, but both can very easily be included with minimal effort.

2. We altered eCP to index and search 36-dimensional audio features in a single day.

are actually considered during the index traversal. In total, the cost per query descriptor is an estimated 5,296 distance calculations (traversing + cluster scan).

This is an unbelievably low number. One cannot help but be amazed that it actually works, but, it does.

We have also showed that eCP can scale-out to use more hardware, both multi-core and distributed.

Because the eCP indexing is both non-iterative and fully data independent, it scales and distributes quite well. In Chapter 3 we showed that the parallelized C++ implementation scales very well to multiple cores and therefore we are confident that the threading issues with the *MultiThreadedMappers*, that we discussed in Chapter 4, can be overcome.

For eCP, the classical single-image-at-a-time search is totally I/O bound. We did however show that both *early-halting* and better storage devices can significantly improve the performance, making it capable of real-time response (at least at small scale).

The batch search process, both parallelized and distributed, is a great option for search scenarios where throughput is more important than individual response time. In Chapter 3, we showed that a single powerful server, running very large batch, can get an average processing time per image below 100ms. Much for the same reasons as the distributed eCP indexing, the distributed eCP batch search, developed in Chapter 4, ran into scalability problems. However, we are certain that we can solve those issues with a better implementation. The 12,000 image batch we could search showed great potential and we thing a large-scale distributed eCP search is capable of extremely good throughput.

As we think eCP can be adapted to other domains, this vast potential for throughput could very useful, for example in a data mining task.

In Section 4.4 of Chapter 4 we gave several lessons that we draw from our experience of working with Hadoop and the Grid'5000. We will not repeat that discussion here, but we would like to emphasize that careful thought must be put into how algorithms are implemented and optimal performance may require some tuning of the default parameters of Haddop.

Because of the Grid'5000 reservation system, we had to deploy our Hadoop cluster for each set of experiments. It is interesting to highlight that it took us almost the same amount of time to transfer the 4TB of data to HDFS as it took to index the full 30.2B descriptor set. Big Data and Cloud computing are currently very popular topics and it is not uncommon to hear Hadoop mentioned in that same sentence. But one wonders how the "big data" is supposed to transferred to and from Clouds without excessive delays and a hefty price tag.

Open issues and future work

As we have pointed out throughout the previous chapters, there are many topics and avenues of research still unaddressed in, or related to, eCP.

Especially pressing are the *Map*-task scalability issues we encountered in Chapter 4 as we believe that neither the distributed indexing nor the distributed batch search have reached their potential.

We would also like to mention two other open issues that are related to distributed search. The first is to speed up the response times for the batch search by adapting an *early-halting* policy into it and the second is to be able to do responsive singe-image-at-a-time search, using the distributed disks without creating to much network communication overhead.

Another avenue of research would be to improve the quality of the deep index structure that eCP builds. We have the ability to trace the *path* of descriptors through the eCP index structure, and know both where two *paths* diverge and even quantify the margin of error at each sub-branch step (using the distances values). This information remains mostly unexplored, but it could be used to create a better and more reliable index structure, hopefully without adding too much overhead.

We have also proposed that eCP is a prototypical algorithm and the potential is there to change its behavior. We could for example reduce the amount of data retained in the clusters, mimicking either BoF, the NV-Tree or use some other idea. Regardless of the specific way we change eCP, a great advantage is that the current system would provide a very good baseline for comparison.

Applying eCP to other domains that require large scale search is also a very interesting task. Because eCP does not use domain specific information to compress or distort the data, any task that can be formulated as a set of high-dimensional vectors and where a distance function can be defined to measure similarity of those vectors, can be searched with eCP.

Résumé étendu

Introduction

Les volumes de données multimédia ont cru ces dernières années de manière spectaculaire. Facebook stocke plus de 100 milliards d'images, et près de 200 millions d'images sont ajoutées chaque jour. Cela oblige les systèmes de recherche d'images par le contenu à s'adapter pour fonctionner à ces échelles. Les travaux présentés dans ce manuscrit vont dans cette direction.

Deux observations essentielles cadrent nos travaux. Premièrement, la taille des collections d'images est telle, plusieurs téraoctets, qu'il nous faut obligatoirement prendre en compte les contraintes du stockage secondaire. Cet aspect est central. Deuxièmement, tous les processeurs sont maintenant multi-coeurs et les grilles de calcul largement disponibles. Du coup, profiter de parallélisme et de distribution semble naturel pour accélérer tant la construction de la base que le débit des recherches par lots.

Cette thèse décrit une technique d'indexation multidimensionnelle s'appelant eCP. Sa conception prends en compte les contraintes issues de l'usage de disques et d'architectures parallèles et distribuées. eCP se fonde sur technique de quantification vectorielle non structurée et non itérative. Durant l'indexation, un index hiérarchique est construit pour ensuite rapidement séparer les données en petits groupes. La recherche identifie à partir de la requête le groupe le plus similaire, y accède sur disque, l'analyse pour y trouver les k-plus-proches voisins.

eCP s'appuie sur une technique de l'état de l'art qui est toutefois orientée mémoire centrale. Notre première contribution se compose d'extensions destinées à permettre de traiter de très larges collections de données en réduisant fortement le coût de l'indexation et en utilisant les disques au mieux. La seconde contribution tire profit des architectures multi-coeurs et détaille comment paralléliser l'indexation et la recherche. Nous évaluons cet apport sur près de 25 millions d'images, soit près de 8 milliards de descripteurs SIFT. La troisième contribution aborde l'aspect distribué. Nous adaptons eCP au paradigme Map-Reduce et nous utilisons Hadoop pour en évaluer les performances. Là, nous montrons la capacité de eCP à traiter de grandes bases en indexant plus de 100 millions d'images, soit 30 milliards de SIFT. Nous montrons aussi la capacité de eCP à utiliser plusieurs centaines de coeurs.

Chapitre 2 : concevoir et développer eCP

Dans le chapitre 2, nous présentons le principe de notre algorithme d'indexation eCP. Cet algorithme comporte naturellement un aspect création d'index multi-

dimensionnel et aussi un aspect recherche par similarité grâce à l'index. Nous nous sommes fixés plusieurs contraintes pour mettre au point eCP. Nous voulons que cet algorithme puisse fonctionner efficacement à l'échelle du Web, c'est à dire sur des volumes d'images très importants (plusieurs dizaines de millions) en gardant tous les descripteurs extraits de ces images. Par conséquent, l'utilisation de disques est obligatoire dans notre approche puisque c'est là que sont stockés les descripteurs.

Notre algorithme s'inspire d'une approche proposée par Chierichetti *et al.* [CPR+07] originellement créée pour indexer de faibles volumes de données textuelles de très grande dimension. Ce premier chapitre motive d'abord notre désir de nous appuyer sur cet algorithme, puis analyse finement son comportement à grande échelle, révélant ainsi divers goulot d'étranglement. Cette analyse permet de proposer différentes extensions destinées à permettre l'indexation multidimensionnelle à très grande échelle. Ces extensions sont de nature différente :

1. l'algorithme original minimise les coûts de traitements, alors qu'il nous faut minimiser le coût des accès disques. Les formules de modèle de coût sont donc changées. Le nouveau modèle intègre le coût des entrées-sorties et permet de définir des *clusters* dont la taille en minimise le coût.
2. à très grande échelle il est obligatoire d'utiliser un processus hiérarchique pour rapidement déterminer dans quel *cluster* doit être stocké un descripteur. Cela permet de réduire fortement le nombre de calculs de distance à effectuer, qui diminue exponentiellement avec la profondeur de la hiérarchie.

Preuve de validité : indexer avec eCP

Un premier ensemble d'expériences est mené pour évaluer eCP et prouver la validité des extensions proposées. Nous avons indexé avec eCP un ensemble de 100 000 images (soit 110 millions de descripteurs SIFT) en utilisant un seul processeur et différents types de disques durs, magnétiques, basés sur des mémoire Flash, des disques SSD donc, une série de disques orchestrés en RAID et un volume NAS.

Ces expériences utilisent deux implémentations différentes du processus d'assignation des descripteurs aux *clusters*, qui se différencient par la manière dont elles gèrent les fichiers temporaires en cours d'indexation. La première, baptisée TF, tend à faire de très nombreuses entrées-sorties aléatoires mais est très simple à mettre en oeuvre alors que l'autre nommée CF privilégie les écritures séquentielles et lectures aléatoires.

Dans nos expériences, nous avons observé que la stratégie TF est limitée par les entrées-sorties et ne permet pas une indexation efficace. L'utilisation de disques SSD en lieu et place de disques magnétiques s'avère améliorer les

performances de manière spectaculaire, puisque le coût habituellement élevé des entrées-sorties aléatoires n'existe pas dans ce cas. Lorsque l'on utilise des SSD, alors TF n'est plus limitée par les performances des entrées-sorties.

L'autre stratégie, CF n'est pas confrontée à ces mêmes limitations. CF est limitée par les besoins en calcul, et cela quels que soient les supports de stockage utilisés. Il est facile de contourner ce problème en en développant une version parallèle. Les bonnes performances de cette stratégie nous la font adopter pour le reste du manuscrit.

Une des conclusions de cette première étude est qu'il est extrêmement bénéfique pour les performances d'utiliser des disques SSD en remplacement des dispositifs traditionnels magnétiques. Malheureusement, la capacité de ces nouveaux supports de stockage est encore limitée et leurs prix est encore fort. Ils seront certainement au centre des futurs systèmes de recherche d'images par le contenu, après que la technologie ait progressé.

Preuve de validité : la qualité de la recherche avec eCP

La véritable validation de la viabilité de eCP est la qualité de la recherche. Nous utilisons pour cela un scénario de détection de copie, où l'on considère la recherche comme étant un succès si l'image originale correspondant à la requête quasi-copie est retournée en première position dans la liste résultat. C'est ainsi le rappel à 1 qui est la métrique qualité qui nous intéresse soit R@1. L'évaluation a été effectuée en utilisant deux ensembles de requêtes, l'un avec 26 variantes Stirmark (redimensionnement, recadrage, compression, rotation, etc) de 120 images ou 3 120 images requêtes au total. L'autre est un ensemble beaucoup plus petit de seulement 533 images requêtes créées à partir de 11 images originales, également définies par des variantes StirMark. Il faut noter que certaines des variantes utilisées dans la première série sont des versions très distordues des images originales et sont donc très difficiles à trouver.

Le R@1 que l'on observe pour le premier jeu de tests varie entre 0,70 et 0,76, en fonctions de différents réglages que nous avons effectué. Le R@1 pour l'autre jeu de données est supérieur à 0.98. Globalement, ces résultats qualitatifs sont très bons, surtout parce que certaines images sont très abîmées.

Preuve de validité : le temps de réponse des recherches

Nous avons également montré que les temps mis par l'algorithme de recherche eCP, lorsqu'il fonctionne selon un mode une requête à la fois, est totalement dépendant des performances des entrées-sorties. Nous avons toutefois diminué ce temps en augmentant au maximum la séquentialité des accès, mais le gain reste faible. L'utilisation de disques magnétique permet de rechercher les images

similaires à une requête en 5,34 secondes et l'on descend à 1,84 en utilisant des SSD.

Le temps dans le cas magnétique n'est pas une surprise, les délais liés à la mécanique sont très importants. De plus, il y a près de 1 000 descripteurs dans une requête, et donc 1,000 *clusters* doivent être lus, faisant certainement autant d'entrées-sorties aléatoires. Les SSD n'en souffrent pas.

Preuve de validité : optimisations au niveau image

Nous avons également présenté et évalué une stratégie d'abandon précoce de la recherche. L'idée est de faire de multiples agrégations de votes pendant le processus de recherche et de stopper la recherche dès que le score d'une image domine largement les autres. Arrêtant la recherche, on évite de lire d'autres *clusters* et on économise de nombreux calculs de distance.

Pour rendre cela possible, nous utilisons des méthodes statistiques pour estimer quelle est la probabilité que l'analyse à venir de données puisse changer l'issue de l'agrégation actuelle des votes. Un score très fortement dominant indique que l'on a trouvé, et un score qui ne décolle pas bien que de nombreux descripteurs requête aient été utilisés indique qu'aucune image est similaire. Cette technique réduit très fortement les temps de réponse. Il a par exemple permis de descendre à des temps de traitement par image de l'ordre de 388ms pour les accès magnétiques et 250ms. sur le disque SSD, sans perte de qualité.

Preuve de validité : la qualité des recherches au niveau descripteur

Nous avons également analysé le comportement de l'algorithme lorsque l'on effectue des recherches au niveau de descripteurs individuels. Pur cela, nous avons utilisé la base ANN_SIFT qui contient un milliard de descripteurs SIFT pour lesquels les 1 000 plus proches voisins de 10 000 points requête ont été déterminés par un algorithme exact exhaustif.

Le rappel que nous observons en menant cette expérimentation n'est pas très bon. Il semble que cela relève plus d'un problème de définition de la base et de la vérité terrain que d'un problème intrinsèque à l'algorithme de recherche. En effet, les requêtes sont tirées au hasard, et donc sont parfois peu représentatives de réelles variations de similarité. Les plus proches voisins sont souvent très loin dans l'espace, accentuant l'importance du bruit et réduisant la qualité. Lorsque l'on se limite à observer les performances des recherches de points de voisinage relativement faible, alors les résultats sont excellents.

eCP, un algorithme représentatif

eCP est un algorithme d'indexation représentatif de ceux que l'on peut trouver dans la littérature. Il est simple de le modifier pour lui faire adopter un fonctionnement analogue à celui du NV-Tree ou des techniques basées sur le modèle BoF. Nous avons par ailleurs utilisé avec succès eCP sur d'autres descripteurs, de dimension 36, et caractérisant des flux audio.

Chapitre 3 : paralléliser eCP

Le chapitre 3 développe une version parallèle de eCP, tant pour l'indexation que pour la recherche.

Indexation parallélisée

Nous avons du diviser en trois phases le processus d'indexation pour ensuite le paralléliser.

Durant la phase #1, nous construisons de manière descendante la structure hiérarchique de l'index en choisissant au hasard des représentants de *clusters*. Ce processus est peu cher et ne nécessite aucune parallélisation.

Durant la phase #2, nous utilisons la hiérarchie construite pendant la phase #1 pour assigner chaque descripteur de la collection au bon *cluster*. C'est cette phase que nous allons paralléliser. D'abord, elle est très gourmande en CPU puisque de nombreux calculs de distance sont faits durant l'assignation. De plus, le processus est intrinsèquement parallélisable puisque les traitements sont indépendants les uns des autres. Il est donc aisé d'avoir plusieurs exécutions simultanées, chacune assignant des descripteurs indépendants dans des *clusters* temporaires. Il faut noter que diverses précautions doivent être prises pour ne pas introduire d'erreurs dues à des écritures concurrentes.

Durant la phase #3, les *clusters* temporaires doivent être fusionnés pour former la base finale. Cette phase ne fait aucun calcul, juste des entrées-sorties, et il est inutile de la paralléliser.

Indexation parallélisée : ajout de *hardware*

Nous avons évalué la capacité de eCP à fonctionner sur une architecture où l'on augmente le nombre de processeurs, afin de vérifier si l'algorithme se comporte bien lorsque l'on ajoute des dispositifs matériels. Nous avons utilisé une machine puissante ayant 12 coeurs physiques et 24 coeurs logiques. Nous avons indexé un ensemble de données fixé en faisant croître le nombre de coeurs utilisés, et nous avons mesuré les temps mis par les différentes indexations.

Nous avons observé que les performances de notre implémentation suit de très près la courbe optimale où le travail est divisé par le nombre de coeurs. eCP s'adapte bien à une évolution matérielle.

Indexation parallélisée : augmentation du volume de l'index

Nous avons également évalué eCP en variant les configurations de l'index, le créant plus ou moins grand selon les réglages. Nous avons utilisé plusieurs configurations, certaines réalistes, d'autres irréalistes mais juste destinées à mettre en avant le comportement de l'algorithme.

Nous avons utilisé un jeu de test de 8,1 milliards de SIFT et créé des index dont les *clusters* contiennent en moyenne 1 000, 5 000, 50 000 et 500 000 descripteurs. Cela change bien évidemment le nombre de représentants dans la hiérarchie et le nombre de *clusters* sur le disque. Dans le cas de *clusters* ayant 1 000 points, nous créons 8 122 260 clusters de 128Ko. L'index ainsi créé est appelé 8.1M. Sa profondeur est 5. L'index comportant des *clusters* de 5 000 descripteurs possède 1 624 452 *clusters* d'environ 665Ko, de profondeur 4. Il s'appelle 1.6M. Les deux autres index ont respectivement 162 446 et 16 245 *clusters*. Il est intéressant de noter que ce dernier, comportant peu de représentants, peut tenir dans le cache L3 de notre machine.

Nous avons ensuite lancé les différents processus d'indexation, en scrutant les performances au travers d'un outil spécifique aux machines parallèles, TipTop, et donnant des informations sur les défauts mémoire, le nombre d'instructions par cycle, etc. Il en ressort que les index ayant de nombreux représentants en mémoire sont plus lents que les plus petits, ceci s'expliquant par le plus grand nombre de défaut de cache à résoudre. Nous avons testé différentes variantes d'implémentation, une basée pointeurs et une autre basée copie. Cette dernière, bien que beaucoup plus gourmande en mémoire, est plus rapide à utiliser car elle évite le parcours aléatoire de la RAM et maximise le nombre de fois où l'on trouve directement en cache ce dont on a besoin. Cette solution ne passe pas à l'échelle lorsque l'on veut indexer de très gros ensembles de données où le nombre de représentants doit être très élevé.

Indexation parallélisée : qualité de la recherche

Nous avons également évalué la qualité de la recherche sur les différentes structures d'index (1,6M et 8,1M) pour les 8,1 milliards de descripteurs.

Nous utilisons pour cela deux ensembles de requêtes, CopyDays qui se compose de 3055 variantes de 127 images originales, et 49K qui se compose de 48.883 images requêtes créées à partir de 49 variantes de 1.000 images.

Pour CopyDays, et en utilisant l'index appelé 8,1M, le rappel est à 0,8448 si on analyse juste un *cluster* et à 0,8733 si on en analyse trois. Ce taux monte au

dessus de 0,9 si on exclue les variantes très dégradées particulièrement difficiles à trouver. Les performances en rappel pour l'index 1.6M sont respectivement de 0,8668 et de 0,9011. Pour l'ensemble 49K, le rappel est de 0,9358 avec l'index 8,1M et monte à 0,9503 avec l'index 1.6M.

Nous pouvons dire que la recherche est de très bonne qualité. Passons maintenant à ses performances.

Recherche parallélisée : faire des recherches par lots

Le chapitre précédent nous montre les limites de recherches individuelles. Ici, nous défendons une approche de recherche par lots où de très nombreuses images requête sont accumulées puis cherchées en une seule fois. Le traitement par lot accroît certes le nombre d'entrées-sorties à effectuer, mais il accroît aussi le degré de séquentialité des accès disques si l'on s'autorise leur réordonnancement. De plus, plusieurs requêtes peuvent accéder le même *cluster*, factorisant ainsi les coûts.

Plus la taille du lot augmente, plus les performances migrent d'un coût lié aux accès disques vers un coût lié aux calculs de distance. On se trouve alors dans une situation propice à la parallélisation. On maximise ainsi le débit, au détriment du temps de réponse.

Recherche parallélisée : mise en oeuvre

La recherche détermine d'abord, pour tous les descripteurs des requêtes du lot, quel est le *clusters* dont chacune a besoin. Ceci est mis dans une structure de données permettant de savoir, pour un *cluster* particulier, quelles sont les requêtes concernées.

Un pool de threads est ensuite créé. Un thread lit ensuite les *clusters* exigés dans l'ordre et les passe à d'autres threads de traitement qui font les calculs de distance entre les points du *clusters* et les vecteurs requête pertinents. On suit ici un protocole producteur-consommateur.

Une fois tous les *clusters* nécessaires traités, on procède à l'agrégation des votes.

Recherche parallélisée : évaluation des performances

Nous avons déjà parlé de la qualité des recherches. Nous discutons ici de leurs performances. Nous utilisons bien entendu les jeux de données présentés ci-dessus, à savoir CopyDays et 49k. Nous avons aussi utilisé un autre ensemble d'images n'ayant aucune correspondances dans la base, et composé de 100 000 images téléchargées de Flickr.

Chercher en parallèle les 3 055 requêtes de CopyDays permet de traiter chaque requête en 640ms. Nous sommes à 180ms avec les images de 49k. La réduction du temps vient du fait que le nombre de *clusters* demandés par ce dernier jeu est proportionnellement plus petit que dans le cas de CopyDays, la diversité des images étant moindre. Avec le lot de 100 000 images, le temps converge et se stabilise vers 100ms, ce qui correspond au temps nécessaire pour lire tous les *clusters* séquentiellement.

Chapitre 4 : distribuer eCP

Dans le chapitre 4, nous avons développé une version distribuée de eCP qui fonctionne sur le principe du paradigme Map-Reduce. Notre algorithme d'indexation est distribué, tout autant que notre recherche par lot. Nous avons évalué notre approche en utilisant Hadoop qui est une implémentation libre de Map-Reduce, en Java.

Selon le principe Map-Reduce, le programmeur ne doit développer que deux fonctions qui sont la fonction *Map* et la fonction *Reduce*. La fonction *Map* est exécutée pour chaque bloc de donnée source. Le résultat est envoyé au travers du réseau à des instances de fonctions *Reduce* qui finissent de les traiter. Une fonction *Reduce* particulière ne traite que les données ayant une clé de valeur spécifique, les autres données de clé différente étant orientées vers d'autres instances de fonction *Reduce*.

Indexation distribuée

Dans notre mise en oeuvre distribuée de l'indexation, les données d'entrée sont stockées sur HDFS et le processus d'assignation des descripteurs se fait dans les fonctions *Map*. Chacun assigne à des *clusters* temporaires les descripteurs stockés dans le bloc de données qui doit être traité.

Pour faire cette assignation, la fonction *Map* a besoin de charger la structure hiérarchique de l'index comme données auxiliaires. Nous réalisons cela efficacement via le système de cache distribué de Hadoop qui précharge les disques locaux à chaque machine. Ainsi, lorsqu'une fonction *Map* demande cette hiérarchie, l'accès est local et non au travers du réseau. Nous évitons ainsi un goulot d'étranglement.

Une fois attribué à un *cluster*, chaque descripteur est émis par le *Mapper* en utilisant l'identifiant du *cluster* comme clé. Tous les enregistrements émis avec la même valeur de clé se retrouvent traités par la même fonction *Reduce*. Celle ci se contente de les écrire sur disque.

Indexation distribuée : performances et résultats

Nous avons commencé par faire des expériences qui visaient à évaluer la capacité d'eCP à fonctionner sur une architecture distribuée de taille croissante. Pour cela, nous avons utilisé un petit jeu de test comprenant 3,3 milliards de descripteurs indexés dans 625k *clusters*. La hiérarchie correspondante est de profondeur 4 et occupe 193Mo de RAM. Nous avons utilisé un parc de 20 à 50 machines, soit de 160 à 400 coeurs.

Les résultats des expériences montrent que le coût global de l'exécution diminue lorsque le nombre de machine croît. Ceci s'explique par le fait que la quantité de données est constante alors que nous utilisons de plus en plus de matériel. Ainsi, il y a globalement plus de RAM pour plus efficacement cacher les données, plus de bande passante réseau, plus d'accès aux disques en parallèle, etc. Notons toutefois que les temps de traitement sont parfois augmentés parce que certaines machines sont particulièrement lentes.

Nous avons aussi réalisé des expériences où c'est la taille des données à traiter qui augmente. Pour cela, en plus de notre jeu de 3,3 milliards de descripteurs, nous avons indexé :

1. Un ensemble de 7,8 milliards de descripteurs indexés avec 1,5M de représentants, de profondeur 4, et dont la hiérarchie occupe 461Mo.
2. Un ensemble des 30,2 milliards de descripteurs indexé par 6,0M de représentants, de profondeur 5, et dont la hiérarchie occupe 1,8Go.

Pour indexer ces deux ensembles de 7,8 et 30,2 milliards, nous avons utilisé plus de 100 machines provenant de 3 grilles du site Rennais de Grid'5000.

L'indexation de 7,8 milliards de descripteurs n'a posé aucun problème.

Chacune des 105 machines a été configurée pour permettre à 8 *Map* et 2 *Reduce* de s'exécuter en parallèle par machine. Nous disposions ainsi d'au maximum de 840 *Map* en parallèle. L'indexation a demandé 71 minutes, soit 7 455 minutes de travail (71min.$\times$105 machines).

L'indexation de l'ensemble de 30,2 milliards de descripteurs a révélé un problème de passage à l'échelle de notre implémentation. Hadoop exécute toutes les fonctions *Map* sur une même machine de manière totalement indépendante. Aussi, chacune doit alors charger la hiérarchie nécessaire à la phase d'assignation. Une machine où 8 *Map* s'exécutent en parallèle doit réserver 8 segments mémoire, un pour chaque *Map*, où sera stockée exactement le même contenu, cette hiérarchie. Lorsque celle-ci est de taille importante, comme lorsque nous indexons ces 30,2 milliards de descripteurs, alors la mémoire disponible est insuffisante. En effet, chaque hiérarchie occupe 1,8Go. Il nous a donc fallu réduire le nombre de

fonction *Map* tournant en parallèle au sein de chaque machine de 8 à 4, augmentant la mémoire disponible pour chacune. Au total, nous avons ainsi pu exécuter seulement 420 fonctions *Map* en parallèle sur les machines 105 disponibles.

L'indexation de cet énorme jeu de données a pris 600minutes, soit 10 heures, et il a fallu 63,000 minutes de travail. Cette tâche a accompli 4,23 fois plus de travail pour indexer ce jeu de descripteur que le jeu de 7,8 milliards qui est lui 3,85 fois plus petit. Il faut bien entendu tenir compte du fait que cette tâche avait seulement la moitié du nombre de processeurs disponibles.

Notre conclusion est que l'indexation ainsi distribuée fonctionne très bien. Il faut toutefois régler ce problème d'empreinte mémoire pour les grosses hiérarchies d'index. Nous avons une idée sur la façon de remédier à ce problème, et nous avons déjà fait quelques expériences pour la valider. L'idée générale est de multithreader les fonctions *Map*, permettant d'exécuter moins de fonctions au sein de chaque machine, mais d'avoir des threads sur tous les coeurs, partageant la mémoire.

Recherche distribuée : principes et performances

La recherche distribuée suit la ligne de la version parallèle où nous avons observé les bonnes performances de la recherche par lots. Ici, ce sont les calculs de distance qu'il faut distribuer. Ils seront donc faits dans les fonctions *Map*.

Comme pour la version parallèle, nous construisons d'abord une structure de données permettant de savoir quels sont les *clusters* exiges par les requêtes, et inversement. Cette structure est distribuée à toutes les machines participantes et constitue ainsi les données auxiliaires à utiliser durant la recherche.

Chaque fonction *Map* lit le bloc de données qui lui échoit, vérifie si celui-ci contient des *clusters* exigés par les requêtes. Si c'est le cas, alors les calculs de distances sont effectués et les structures mémorisant les plus proches voisins mises à jour. À la fin, ces structures sont envoyées vers les fonctions *Reduce* qui font alors l'agrégation de votes.

Cette mise en oeuvre se heurte exactement au même problème que celui rencontré pour l'indexation, c'est à dire la consommation mémoire importante par machine. Ici, c'est la structure auxiliaire assurant la correspondance requêtes-*clusters* qui est en jeu. En présence de très gros lots à traiter, cette structure demande beaucoup de mémoire.

Une solution existe pour contourner ce problème. On peut aisément hacher cette structure de données et chaque *Map* n'accédera qu'aux paquets pertinents pour les *cluster* de son bloc de données en cours de traitement. Une mise en oeuvre complète de cette idée est en cours.

Les plus gros lots que nous avons pu traiter sans être pénalisés par ce problème de mémoire regroupaient 12 000 images. Dans ce cas, le temps moyen de

traitement d'une requête est de 210ms. Cela est bien meilleur que ce que nous avons pu observer sur le serveur parallèle. L'abondance de matériel sur cette grille est très avantageux.

Recherche distribuée : résultats qualitatifs

Nos travaux se placent dans le cadre du projet Quaero. Là, nos résultats qualitatifs sont évalués par un partenaire extérieur selon un protocole standardisé. Ce protocole s'appuie sur les ensembles CopyDays et 49k, décrits ci-dessus.

Nous avons noyé ces ensembles dans 100 000 000 d'images, soit les 30,2 milliards de descripteurs. L'index créé comporte 6M de *clusters*. Les résultats qualitatifs nous mettent en tête des participants à ces évaluations contrôlées. Les valeurs de mAP que nous obtenus sont 0,9320 pour le jeu 49k, de 0,8393 pour CopyDays et de 0,9261 pour les deux jeux, globalement.

Ce sont de très bons résultats, surtout si l'on réalise qu'il s'agit là de retrouver de toutes petites vignettes de 150 pixels, et donc ne créant que quelques dizaines de vecteurs, parmi 30 264 937 722 autres descripteurs, regroupés en petits paquets d'approximativement 5 000 vecteurs.

Chapitre 5 : conclusion et travaux futurs

Globalement, eCP fonctionne très bien et permet d'effectuer des recherches d'images par similarité à très grande échelle. Nous avons avec succès indexé 100 millions d'images. Pour atteindre cet objectif, nous avons successivement développé eCP et enrichi son algorithmique pour lui permettre de fonctionner d'abord de manière parallèle, ensuite de manière distribuée.

Différentes pistes permettent de prolonger ces travaux. Il faut bien entendu terminer les mises en oeuvre multithread des fonctions *Map* et aussi permettre aux recherches de ne charger qu'une partie de la structure *cluster*-requête.

Nous désirons aussi améliorer la qualité des recherches en changeant la manière dont l'index est construit. Il faut plus de précision dans l'établissement des relations de proximité. Nous pouvons certainement nous appuyer sur les notions de plus proches voisins partagés.

Finalement, le caractère assez générique de l'approche suivie par eCP nous permet de penser que l'on peut l'utiliser avec d'autres descripteurs, selon d'autres techniques (BoF, NV-Tree, etc).

Bibliography

[AJ10] Laurent Amsaleg and Hervé Jégou. ANN_SIFT1B, a set of 1 billion vectors and 10 thousand pre-computed k nearest neighbors. Online website, 2010. `http://corpus-texmex.irisa.fr/`.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. Commun. ACM, 18(9):509–517, September 1975.

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 11:21, 2007.

[BrJB09] Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In Proceedings of the Fourth biennial Conference on Innovative Data Systems Research. CIDR, 2009.

[CPR+07] Flavio Chierichetti, Alessandro Panconesi, Prabhakar Raghavan, Mauro Sozio, Alessandro Tiberi, and Eli Upfal. Finding near neighbors through cluster pruning. In Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '07, pages 103–112, New York, NY, USA, 2007. ACM.

[CPS+07] Ondrej Chum, James Philbin, Josef Sivic, Michael Isard, and Andrew Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval. In Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on, pages 1–8. IEEE, 2007.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. CACM, 2008.

[DKAF12] Thanh-Toan Do, Ewa Kijak, Laurent Amsaleg, and Teddy Furon. Enlarging hacker's toolbox: deluding image recognition by attacking keypoint orientations. In 37th International Conference on Acoustics, Speech, and Signal Processing, ICASSP'12, Kyoto, Japon, March 2012.

[Elk03] Charles Elkan. Using the Triangle Inequality to Accelerate k-Means. In ICML, pages 147–153, 2003.

[FKS03] Ronald Fagin, Ravi Kumar, and Dandapani Sivakumar. Efficient similarity search and classification via rank aggregation. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 301–312. ACM, 2003.

[GAJ12a] Gylfi Þór Gudmundsson, Laurent Amsaleg, and Björn Þór Jónsson. Distributed High-Dimensional Index Creation using Hadoop, HDFS and C++. In CBMI - 10th Workshop on Content-Based Multimedia Indexing, Annecy, France, 2012.

[GAJ12b] Gylfi Þór Gudmundsson, Laurent Amsaleg, and Björn Þór Jónsson. Impact of storage technology on the efficiency of cluster-based high-dimensional index creation. In Proceedings of the 17th international conference on Database Systems for Advanced Applications, DASFAA '12, pages 53–64, Berlin, Heidelberg, 2012. Springer-Verlag.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Rec., 26(4):63–68, December 1997.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In SOSP, 2003.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[GJA10] Gylfi Þór Gudmundsson, Björn Þór Jónsson, and Laurent Amsaleg. A large-scale performance study of cluster-based high-dimensional indexing. In VLS-MCMR Workshop with ACM MM, 2010.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD international conference on Management of data, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[HSDL12] Jonathon S. Hare, Sina Samangooei, David P. Dupplaw, and Paul H. Lewis. ImageTerrier: an extensible platform for scalable high-performance image retrieval. In Proceedings of the 2nd ACM International Conference on Multimedia Retrieval, ICMR '12, pages 40:1–40:8, New York, NY, USA, 2012. ACM.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[Jai08] Anil K. Jain. Data Clustering: 50 Years Beyond k-means. In Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I, ECML PKDD '08, pages 3–4, Berlin, Heidelberg, 2008. Springer-Verlag.

[JDS08] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In Andrew Zisserman David Forsyth, Philip Torr, editor, European Conference on Computer Vision, volume I of LNCS, pages 304–317. Springer, oct 2008.

[JDS11] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. IEEE Trans. on PAMI, 2011.

[JDSP10] Hervé Jégou, Matthijs Douze, Cordelia Schmid, and Patrick Pérez. Aggregating local descriptors into a compact image representation. In Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on, pages 3304–3311, 2010.

[JLLa06] Yvon Jégou, Stephane Lantéri, Julien Leduc, and all. Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. Intl. Journal of HPC Applications, 20(4), 2006.

[LAJA06] Herwig Lejsek, Fridrik H. Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. Blazingly fast image copyright enforcement. In Proceedings of the 14th annual ACM international conference on Multimedia, MULTIMEDIA '06, pages 489–490, New York, NY, USA, 2006. ACM.

[LGK+10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In Conference on Uncertainty in Artificial Intelligence (UAI), July 2010.

[LGK+12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A

framework for machine learning in the cloud. CoRR, abs/1204.6078, 2012.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision, 60(2):91–110, November 2004.

[MSGA13] Diana Moise, Denis Shestakov, Gylfi Þór Gudmundsson, and Laurent Amsaleg. Indexing and searching 100m images with map-reduce. In Procedings of 2013 International conference on Multimedia Retrival, ICMR, 2013.

[NS06] D. Nistér and H.K Stewénius. Scalable Recognition with a Vocabulary Tree. In Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, volume 60, pages 2161–2168, 2006.

[OD11] Stephen O'Hara and Bruce A. Draper. Introduction to the Bag of Features Paradigm for Image Classification and Retrieval. CoRR, abs/1101.3354, 2011.

[PAK97] Fabien A.P Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Stirmark: Image–watermarking robustness test. Online website, 1997. `http://www.cl.cam.ac.uk/~mgk25/stirmark.html`.

[PAK98] Fabien A.P Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Attacks on copyright marking systems. In Information Hiding, volume 1525, pages 219–239. Springer-Verlag, April 1998.

[PCI+07] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on, pages 1–8. IEEE, June 2007.

[PCI+08] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. IEEE, 2008.

[Roh11] Erven Rohou. Tiptop: Hardware Performance Counters for the Masses. Research Report 7789, INRIA, November 2011.

[SJHA05] Rut Sigurdardottir, Björn Þór Jónsson, Hlynur Hauksson, and Laurent Amsaleg. The quality vs. time trade-off for approximate image descriptor search. In Proceedings of the 21st International Conference on Data Engineering Workshops, ICDEW '05, pages 1175–, Washington, DC, USA, 2005. IEEE Computer Society.

[SMGA13] Denis Shestakov, Diana Moise, Gylfi Þór Gudmundsson, and Laurent Amsaleg. High-dimensional indexing on grid. In CBMI - 11th Workshop on Content-Based Multimedia Indexing, Veszprém, Hungary, 2013.

[SPS04] Tomás Skopal, Jaroslav Pokorný, and Václav Snásel. PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases. In ADBIS (Local Proceedings), 2004.

[SR06] Uri Shaft and Raghu Ramakrishnan. Theory of nearest neighbors indexability. ACM Trans. Database Syst., 31(3):814–838, September 2006.

[SZ03] J. Sivic and A. Zisserman. Video Google: a text retrieval approach to object matching in videos. In Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on, volume 2, pages 1470–1477, oct. 2003.

[TJA11] Romain Tavenard, Hervé Jégou, and Laurent Amsaleg. Balancing clusters to reduce response time variability in large scale image search. In International Workshop on Content-Based Multimedia Indexing, CBMI'11, Madrid, Espagne, June 2011.

[TTSF00] Caetano Traina, Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes. In Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '00, pages 51–65, London, UK, UK, 2000. Springer-Verlag.

[WJ96] David A. White and Ramesh Jain. Similarity Indexing with the SS-tree. In Proceedings of the Twelfth International Conference on Data Engineering, ICDE '96, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.

List of Figures

Abstract

The scale of multimedia collections has grown very fast over the last few years. Facebook stores more than 100 billion images, 200 million are added every day. In order to cope with this growth, methods for content-based image retrieval must adapt gracefully. The work presented in this thesis goes in this direction.

Two observations drove the design of the high-dimensional indexing technique presented here. Firstly, the collections are so huge, typically several terabytes, that they must be kept on secondary storage. Addressing disk related issues is thus central to our work. Secondly, all CPUs are now multi-core and clusters of machines are a commonplace. Parallelism and distribution are both key for fast indexing and high-throughput batch-oriented searching.

We describe in this manuscript a high-dimensional indexing technique called eCP. Its design includes the constraints associated to using disks, parallelism and distribution. At its core is an non-iterative unstructured vectorial quantization scheme.

eCP builds on an existing indexing scheme that is main memory oriented. Our first contribution is a set of extensions for processing very large data collections, reducing indexing costs and best using disks. The second contribution proposes multi-threaded algorithms for both building and searching, harnessing the power of multi-core processors. Datasets for evaluation contain about 25 million images or over 8 billion SIFT descriptors. The third contribution addresses distributed computing. We adapt eCP to the MapReduce programming model and use the Hadoop framework and HDFS for our experiments. This time we evaluate eCP's ability to scale-up with a collection of 100 million images, more than 30 billion SIFT descriptors, and its ability to scale-out by running experiments on more than 100 machines.

Résumé

Les volumes de données multimédia ont fortement crus ces dernières années. Facebook stocke plus de 100 milliards d'images, 200 millions sont ajoutées chaque jour. Cela oblige les systèmes de recherche d'images par le contenu à s'adapter pour fonctionner à ces échelles. Les travaux présentés dans ce manuscrit vont dans cette direction.

Deux observations essentielles cadrent nos travaux. Premièrement, la taille des collections d'images est telle, plusieurs téraoctets, qu'il nous faut obligatoirement prendre en compte les contraintes du stockage secondaire. Cet aspect est central. Deuxièmement, tous les processeurs sont maintenant multi-coeurs et les grilles de calcul largement disponibles. Du coup, profiter de parallélisme et de distribution semble naturel pour accélérer tant la construction de la base que le débit des recherches par lots.

Cette thèse décrit une technique d'indexation multidimensionnelle s'appelant eCP. Sa conception prends en compte les contraintes issues de l'usage de disques et d'architectures parallèles et distribuées. eCP se fonde sur technique de quantification vectorielle non structurée et non itérative.

eCP s'appuie sur une technique de l'état de l'art qui est toutefois orientée mémoire centrale. Notre première contribution se compose d'extensions destinées à permettre de traiter de très larges collections de données en réduisant fortement le coût de l'indexation et en utilisant les disques au mieux. La seconde contribution tire profit des architectures multi-coeurs et détaille comment paralléliser l'indexation et la recherche. Nous évaluons cet apport sur près de 25 millions d'images, soit près de 8 milliards de descripteurs SIFT. La troisième contribution aborde l'aspect distribué. Nous adaptons eCP au paradigme Map-Reduce et nous utilisons Hadoop pour en évaluer les performances. Là, nous montrons la capacité de eCP à traiter de grandes bases en indexant plus de 100 millions d'images, soit 30 milliards de SIFT. Nous montrons aussi la capacité de eCP à utiliser plusieurs centaines de coeurs.