A Large-Scale Performance Study of Cluster-Based High-Dimensional Indexing

Gylfi Thór Gudmundsson, Björn Thór Jónsson, Laurent Amsaleg

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Large-Scale Performance Study of Cluster-Based High-Dimensional Indexing

Gylfi Þór Gudmundsson — Björn Þór Jónsson — Laurent Amsaleg

N° 7307

Juin 2010

Vision, Perception and Multimedia Understanding

ISSN 0249-6399 ISRN INRIA/RR--7307--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

A Large-Scale Performance Study of Cluster-Based High-Dimensional Indexing

Gylfi Þór Gudmundsson* , Björn Þór Jónsson† , Laurent Amsaleg

Theme : Vision, Perception and Multimedia Understanding
Équipes-Projets Texmex

Rapport de recherche n° 7307 — Juin 2010 — 20 pages

Abstract: High-dimensional clustering is a method that is used by some content-based image retrieval systems to partition the data into groups; the groups (clusters) are then indexed to accelerate the processing of queries. Recently, the Cluster Pruning approach was proposed as a very simple way to efficiently and effectively produce such clusters. While the original evaluation of the algorithm was performed within a text indexing context at a rather small scale, its simplicity and performance motivated us to study its behavior in an image indexing context at a much larger scale. We experiment with two collections of 72-dimensional state-of-the-art local descriptors, the larger collection containing 189 million descriptors. This paper summarizes the results of this study and shows that while the basic algorithm works fairly well, three extensions can dramatically improve its performance and scalability, accelerating both query processing and the construction of clusters, making Cluster Pruning a promising basis for building large-scale systems that require a clustering algorithm.

Key-words: Content-Based Image Retrieval Systems, clustering, multidimensional indexing, large scale

* School of Computer Science, Reykjavík University, Menntavegi 1, IS 101 Reykjavík, Iceland. gylfi03@ru.is
† School of Computer Science, Reykjavík University, Menntavegi 1, IS 101 Reykjavík, Iceland. bjorn@ru.is

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Étude de performance à grande échelle d'un indexation multidimensionnelle basée clusters

Résumé : Le *clustering* en grandes dimensions est une méthode employée par certains systèmes de recherche d'images par le contenu pour partitionner l'espace en groupes. Les groupes sont ensuite indexés pour accélérer le traitement des requêtes. Récemment, une approche dite "Cluster Pruning" a été proposée comme permettant l'obtention simple, rapide et efficace de ces groupes. Alors que son évaluation originale s'est effectuée dans un contexte d'indexation de textes et à une échelle réduite, sa simplicité et ses performances ont été une forte motivation pour étudier son comportement à bien plus grande échelle, et dans un contexte image. Nous menons des expérimentations où sont utilisés des descripteurs locaux d'image appartenant à l'état de l'art et de dimension 72. Nous traitons plusieurs collections de descripteurs, dont la plus grande en contient 189 millions. Cet article présente une synthèse des résultats de cette étude et montre que l'algorithme original fonctionne relativement bien. Toutefois, trois extensions simples permettent d'améliorer de manière très importante ses performances et son aptitude à passer à l'échelle, en accélérant tant le traitement des requêtes que le temps de construction des groupes. Dotée de ces extensions, l'approche "Cluster Pruning" devient alors une brique essentielle pouvant servir aux systèmes grande échelle nécessitant la création de groupes de points.

Mots-clés : Systèmes de recherche d'images par le contenu, partitionnement, indexation multidimensionnelle, grande échelle

1 Introduction

Recently, there has been a significant burst of research activity on data structures and algorithms for approximate nearest neighbor search in high-dimensional descriptor collections (e.g., see [4, 5, 9, 19]). Generally speaking, all these methods are based on some sort of segmentation of the high-dimensional collection into groups of descriptors, which are stored together on disk. At query time, an index is then typically used to select the single nearest such group for searching. The goal of the approximate search is to find a good trade-off between result quality and retrieval time.

1.1 Cluster-Based Retrieval

Several of the methods that have been proposed are based on using clustering algorithms to group the data. This line of work was pioneered by Li *et al.* [11], which proposed the Clindex framework, where a dynamic search algorithm could halt processing after reading a given number of clusters. They showed that good approximate results could be obtained by reading a small number of clusters, albeit for a very small collection. Their particular clustering algorithm did not scale well in practice, however.

Traditionally, clustering algorithms, such as k-means, find the "natural" clusters of the data, and produce large clusters (containing many descriptors) in dense areas of the high-dimensional space and small clusters (containing few descriptors) in sparse areas. Sigurðardóttir *et al.* [18] showed, however, for their particular collection, that large clusters are very detrimental to performance, and that excellent approximate results could be returned by simply bulk-loading the descriptors into an SR-tree and using the resulting leaves to create clusters of an even size. Indeed, when result quality was considered as a function of time, early results were much better with this simple clustering scheme than with a traditional clustering algorithm.

Chierichetti *et al.* [3] then proposed a very simple algorithm, called Cluster Pruning, which uses the initial steps of the k-means algorithm to select a number of random cluster leaders and assign each descriptor to a single leader. Like in [11], at search time, the nearest b clusters are read and used to produce the approximate results. To improve result quality, they proposed some parameters affecting the size of clusters and the depth of the cluster index.

1.2 Scalability

While the algorithm of Chierichetti *et al.* is efficient and effective, as predicted by the previous results, and their analysis is impressive, the performance of the algorithm was only studied using a small scale text collection. Its simplicity and performance was a strong motivation to study its behavior in an image indexing context at a larger scale, where secondary storage is needed.

State-of-the-art image applications typically use the SIFT descriptors [12] or variants thereof [7, 9]. These descriptors have two important properties that make them suitable for large-scale retrieval. First, they have been shown to scale very well with respect to result quality [10]. Second, each image is described by hundreds of descriptors, making approximate queries (and thus potentially Cluster Pruning) appropriate for these applications. Because each

image is described by hundreds of these high-dimensional descriptors, large-scale indexing and retrieval is absolutely necessary.

A major assumption made in the original design of Cluster Pruning is that CPU cost is dominant during the search. As a result of the decision to ignore disk cost, the optimal segmentation is to index a collection of n descriptors into $\sqrt{n}$ clusters containing, on average, $\sqrt{n}$ descriptors each; this division minimizes the total CPU cost of the retrieval. While the calculation of Euclidean distances is indeed CPU intensive, disk operations are also a significant source of cost, as shown in [18]. It is therefore necessary to study, for realistic workloads and data sets that need to be stored on disks, the optimal settings for the number of clusters and the resulting distribution of cluster sizes.

1.3 Contributions

In this paper, we study the performance of the Cluster Pruning algorithm in the context of a large-scale image copyright protection application. The copyright protection application has been studied significantly in the literature (e.g., see [1, 8, 9]) and good results have been obtained using a number of local descriptor variants. Furthermore, as queries are formed by modifying images in the image collection, there is no need for subjective judgment on similarity of images, greatly facilitating interpretation of results.

We study the effect of the various parameters of the Cluster Pruning algorithm, including index depth and cluster size, in this disk-based setting. Our results contradict some of the conclusions reached by Chierichetti *et al.* [3], due to the large scale of our experimental setup. While the basic algorithm still works fairly well, we propose three key changes which significantly improve its performance. First, a new parameter is needed to control cluster size on disk, to better balance IO and CPU costs. Second, a modification, which enables the use of the cluster index during the clustering phase, allows clustering the collection in a reasonable time. Third, by creating additional clusters and then reclustering the contents of the smallest clusters, cluster size distribution is improved which, in turn, improves search efficiency.

Note that, as mentioned above, there has been much recent research activity in the area of high-dimensional indexing. As a result, there are other competing approaches, which have similar theoretical properties, but may be appropriate for different applications (e.g., see [4, 10, 13, 15, 19]). In this paper, we do not attempt a comparison of all these approaches, as such a comparison would be extremely time-consuming, but focus instead on understanding the performance of one specific approach, the Cluster Pruning algorithm, for a particular workload setting. There is significant overlap between the ideas behind Cluster Pruning and the other approaches; Cluster Pruning can therefore be seen as a good representative for a whole family of algorithms where clustering is central. We thus believe that our analysis represents a very valuable contribution to the general understanding of disk-oriented cluster-based indexing.

1.4 Outline of the Paper

The remainder of the paper is organized as follows. In Section 2 we review the copyright protection application we use in our work. We then review the Cluster Pruning algorithm in Section 3. In Section 4 we propose extensions to

this algorithm for disk-based processing of large collections. In Section 5 we then run a detailed study of the impact of various parameters on performance. We discuss related work in Section 6, before concluding in Section 7.

2 Image Copyright Protection

The application we use as a case study is the well known image copyright protection application (see [9, 8]). It is very different from the one studied by Chierichetti *et al.*, where they used about 95,000 document descriptors with more than 400,000 dimensions. In order to set the context for the work, and for our examples, we now describe this application and our experimental environment.

2.1 Image Collections and Queries

We use two collections of images. The first collection contains 30K high-quality news photos, which are very varied in content. The second collection, which includes the first collection, contains about 300K such photos.

Queries are intended to simulate image theft. The standard method for this purpose is to generate modified variants of images in the collection using the StirMark software [14] and use those variants as queries. The goal is then to return the original image as a match, but no other images. For the purposes of our evaluation, 120 images were chosen at random from the collection, and modified with 26 different StirMark variants (the variants include resizing, cropping, compression, and some severe brightness modifications, see [9] for details), resulting in 3,120 query images.

2.2 Descriptors and Query Model

Each image is described with many local descriptors, each describing a small portion of the image. We use the Eff^2 descriptors, which are a variant of SIFT [12], but perform significantly better for this application [9]. An Eff^2 descriptor has 72 dimensions, each stored in a byte. Additionally, each descriptor stores the identifier of the image it was extracted from, for a total of 76 bytes. The small collection has a total of 20,445,871 descriptors, while the large collection has 189,605,419 descriptors. The collections thus require 1.5GB and 13.4GB of disk storage, respectively.

Beyer *et al.* [2] and Shaft and Ramakrishnan [17] have shown that the only way to obtain meaningful performance results for large-scale high-dimensional indexing, is to use real application data which has been shown to scale well in terms of retrieval quality. They have, for example, shown that the data distribution of most generated collections is such that those collections can neither yield meaningful results [2], nor be efficiently indexed [17]. Previous work has shown that SIFT descriptors do indeed scale well to large collections [10], and we believe that our collections are large enough for our conclusions to be quite general.

The descriptors from the images in the photo collections are stored in a large descriptor file, which is the input to the clustering process. When a query file is received, each of its q query descriptors is used in a k-nearest neighbor search:

the closest cluster representative is first found, the contents of the cluster fetched in memory and distances finally computed to get the k neighbors. In this paper, we use $k = 20$, but the results are not very sensitive to that setting for large collections. Each neighbor votes for the image it was extracted from. These votes are aggregated over the image identifiers, and the images with the most votes are returned as an answer to the query.

2.3 Metrics

The cost of clustering and search is measured through CPU time and IO time, but typically reported together as wall-clock time. The search time reported corresponds to the average time spent to perform each of the 3,120 queries. Quality, on the other hand, is measured as follows. For each of the 3,120 query images, it is clear which image should be returned as a match. We consider an image a "correct match" when the correct image has at least twice as many votes as the image with the second most votes. The percentage of such correct matches is our baseline quality metric.

Note that the quality results in this study are lower than reported in many other studies, for three reasons. First, some of the StirMark variants are very difficult to find and even an exact sequential scan does not find all the correct matches. Second, a few of the selected images have near-duplicates in the collection, and therefore are never found as a correct match using our simple measure. Third, our criteria of having twice as many votes is very strict; it is possible to find a match with a relatively small number of votes by applying post-processing to the top images (e.g., see [8, 12]), but for simplicity we avoid such post-processing. The point of this study, however, is not to show that the descriptors are effective at image copyright protection—this is already known [8, 9, 12]. The main point is to investigate the performance of the Cluster Pruning algorithm, and this simple definition of a correct match suffices for that purpose.

3 The Cluster Pruning Approach

In this section, we briefly describe the Cluster Pruning approach. We first describe the basic algorithm, and then three parameters affecting its behavior. We end by discussing the costs of the Cluster Pruning approach before summarizing the results reported in [3].

3.1 Basic Algorithm

Assume a collection $C = p_1, \ldots, p_n$ of n points in high-dimensional space. The clusters are then formed as follows. First, a set of $l = \sqrt{n}$ cluster *leaders* is chosen randomly from C. Then, each point p_i is compared to all l cluster leaders and assigned to its closest leader. Finally, once the clusters have been formed, a cluster *representative* is chosen, per cluster (the obvious choices are the cluster leader itself, the centroid of the cluster, or the medoid of the cluster).

At query time, the query point q is first compared to the set of l cluster representatives to find the nearest representative. Then, the query point is compared to all the points in that representative's cluster, to determine the k

nearest neighbors found in the cluster. Those neighbors are returned as the approximate answer to the query.

The choice of $l = \sqrt{n}$ clusters is made because the total number of euclidean distance calculations, which is $l + n/l$, is minimized when $l = \sqrt{n}$. On average, each cluster contains $\sqrt{n}$ points, resulting in a total of $2\sqrt{n}$ distance calculations. Assuming that the set of cluster representatives fits in memory, but not the descriptor collection, one disk read is required at search time.

3.2 Extended Searches: The *b* Parameter

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 b IOs and $(1+b)\sqrt{n}$ distance calculations. Using b, it is possible to dynamically change the query execution strategy, for example to read more clusters to improve results.

As b grows, however, returns are expected to diminish as the nearest neighbors are most likely to be contained within the nearest clusters [18]. Unfortunately, a suitable choice of b is difficult to determine dynamically, as the result quality is not known at run-time; instead the number of clusters required for acceptable result quality must be determined explicitly through experimentation.

3.3 Redundant Clustering: The *a* Parameter

Alternatively, it is possible to increase the quality of the results by assigning each data point to $a > 1$ clusters, and reading only $b = 1$ cluster at query time. Each cluster will then contain, on average, $a\sqrt{n}$ points, resulting in $(a + 1)\sqrt{n}$ euclidean distance calculations, but only one IO.

The clustering phase is always more costly with higher a (the average cluster size is proportional to a). Furthermore, it is not possible to change the a parameter once the clusters are formed, while the b parameter can be dynamically modified at query time.[1] The effect of the a parameter on query processing cost is more complex, and is studied in Section 5. In short, as a is increased, the size of the clusters on disk increases, as well as the time required to process them.

3.4 Recursive Clustering: The *L* Parameter

For large collections, $\sqrt{n}$ is a large number, resulting in excessive CPU cost and potentially even significant IO cost. The solution suggested by Chierichetti *et al.* is to recursively cluster the set of cluster representatives, using the exact same method. They introduce a parameter, L, to control the number of levels in the recursion; the default algorithm described above corresponds to $L = 1$.

The L parameter is used as follows during the clustering, which is performed in a bottom-up manner. First, $l = n^{L/(L+1)}$ cluster leaders are now chosen initially, resulting in l clusters containing on average $n^{1/(L+1)}$ descriptors. Cluster assignment then proceeds as before, as does the choice of cluster representatives. Once the cluster representatives are formed, however, they are considered as a collection of high-dimensional points, and clustered using $n^{(L-1)/(L+1)}$ representatives. This process is repeated recursively, and the outcome is an L-tier

[1] Note that while it is possible to have both $a > 1$ and $b > 1$, such settings will most likely result in several data points being read a times and are therefore not considered.

index of cluster representatives, where each representative always represents, on average, $n^{1/(L+1)} = \sqrt[L]{l}$ points at the next level. At query time, the total number of distance calculations is $(L+1)n^{1/(L+1)}$, while the number of IOs is at most L, assuming at least the top level fits in memory.

Note that the size of each cluster decreases rapidly as L grows. This method is thus effective at decreasing CPU cost, but potentially at the expense of additional IOs.

Example 1 *For a collection of 1 million descriptors, $L = 1$ yields a cluster index of 1,000 representatives with 1,000 descriptors per cluster on average. Searching this index, with $b = 1$, therefore requires $2 \times 1,000 = 2,000$ distance calculations per query descriptor. Using $L = 2$, on the other hand, yields 10,000 clusters with 100 descriptors per cluster, and searching requires $3 \times 100 = 300$ distance calculations.*

3.5 Cost of Cluster Pruning

During query processing, Cluster Pruning incurs costs for scanning the cluster index and processing clusters. While clustering costs do not affect search throughput, they are nevertheless important, as cluster generation must take reasonable time. We now briefly discuss the impact of a, b, l and L on the CPU and IO costs of querying and clustering.

Cost of Index Scan. Assuming the cluster index fits entirely in memory, the cost of the index scan is only CPU cost, which is $O(abL\sqrt[L]{l})$ (as before, either $a = 1$ or $b = 1$).

Cost of Cluster Scan. The CPU cost of sequentially scanning the b clusters is $O(abl)$. The IO cost of reading clusters is $O(b(C + al))$, where C is the cost of a random IO relative to a distance calculation (this cost depends on hardware, layout on disk, etc.).

Cost of Clustering. Assuming that the cluster index fits in memory, the cost of the clustering process is affected mostly by the a parameter. The CPU cost, however, consists of scanning the cluster index for each database descriptor to find the correct cluster, for a cost of $O(naL\sqrt[L]{l})$.

3.6 Summary of Previous Results

While the bulk of the results reported by Chierichetti *et al.* [3] were obtained using a collection of about 95,000 descriptors with dimensionality of about 400,000, it is still instructive to recall their results.

Their goal was to determine the parameter settings that gave the best result quality in the shortest time span. First, they found cluster centroids to be the best representatives, followed by the cluster leaders. For that small collection, $L = 1$ gave the best results, followed closely by $L = 2$. Higher values of L resulted in very poor results. They also found that for a memory-based setting using $a = 1$ worked best, as then b could be varied to increase quality, while for a disk-based setting using $a = 5$ and $b = 1$ gave the best results. Our results, on the other hand, indicate that for large collections, using $L > 1$ and $a = 1$ is always preferred.

4 Cluster Pruning Extensions

The main emphasis of the original algorithm was to minimize the CPU cost of queries. We now propose four new design choices that affect performance significantly, when dealing with local descriptors in a disk-based setting.

4.1 Cluster Size Selection

The results in [18] indicated that cluster size is a key factor in the performance of cluster indexing, and that cluster size should be heavily influenced by the characteristics of the hard disk drive that descriptors reside on. In the original Cluster Pruning approach, however, there is a large difference in cluster sizes for $L = 1$ and $L = 2$, and both are independent of the IO granularity of the disk. While this behavior minimizes the CPU cost, increasing L leads to very small descriptor clusters on disk, which under-utilize the IOs, and a correspondingly large index.

Instead of choosing $l = n^{L/(L+1)}$ leaders in the first step, we propose to give the desired average cluster size and then determine the number of leaders as follows:

$$l = \left\lceil \frac{n}{\lfloor \textit{desired cluster size / descriptor size} \rfloor} \right\rceil \quad (1)$$

Using this new number of cluster leaders, the clustering proceeds as before. When $L > 1$, each intermediate-level representative still represents $\sqrt[L]{l}$ points at the next level.

Example 2 *Assuming a desired cluster size of 128KB (the default IO granularity of the Linux operating system) each cluster should contain* $\lfloor 128KB/76B \rfloor = 1,724$ *descriptors. For our small collection, the resulting number of cluster leaders would be* $l = \lceil 20,445,871/1,724 \rceil = 11,859$.

By decoupling the size of the clusters from the choice of L, we gain two major benefits. First, larger clusters lead to a smaller index that may fit entirely in memory. Second, as each cluster is larger, fewer clusters may potentially be read. While CPU cost is sacrificed, the IO cost is reduced resulting in lower overall query processing cost.

4.2 Choice of Cluster Representatives

Chierichetti *et al.* considered three potential choices for cluster representatives: the cluster leaders, the cluster centroids, and the cluster medoids (the descriptor closest to the centroid). Their conclusion was that the centroids gave the best performance, followed closely by the cluster leaders.

We, on the other hand, propose to use the cluster leaders, for the following reason. When cluster leaders are used, the bottom level of the cluster index is already known before descriptors are assigned to clusters. This, in turn, means that the upper levels of the cluster index can also be created before the cluster assignment. As a result, the entire cluster index can be created before cluster assignment and can therefore be used to direct the descriptors to the appropriate cluster during the clustering phase, resulting in a very significant reduction of clustering time.

Note that this optimization is not possible with the other choices of cluster representatives, as those are not known until the actual clusters have been created. While centroids may yield slightly better results (our initial experiments showed small benefits, if any), the difference in clustering time is so dramatic that it necessitates this choice.

When using an index during cluster assignment, however, it is not clear that the most appropriate cluster is always found for all descriptors. To increase the likelihood of finding the best cluster for each descriptor, we always create the upper levels of the index using $a = 3$. While this setting does increase the index size, it can still easily fit in memory.

4.3 Balanced Size Distribution

In [18], it was shown that the largest natural clusters of a descriptor collection might be as large as 5–20% of the collection, while many clusters were very small. Small clusters still require an IO operation, while contributing little to the result quality. Large clusters result in both a more expensive IO operation and additional CPU cost. Both small and large clusters, therefore, reduce query processing performance. Furthermore, large clusters tend to get selected more often for processing than the average cluster, which impacts query processing even further.

In theory, the random leader selection process should generate equally sized clusters. In practice, however, the reality is that several clusters are significantly smaller then the desired size and a few large clusters are an order of magnitude larger than the average cluster. While the cluster size distribution is much better balanced than for an algorithm which generates natural descriptor clusters, it is still possible to improve the distribution.

We propose a simple, yet surprisingly effective method to balance the size distribution. We intentionally choose $X\%$ additional cluster representatives in the initial step of the algorithm. At the end of the cluster creation process we then eliminate the corresponding number of the smallest cluster leaders by reclustering their descriptors into the l remaining clusters. In addition to the obvious advantage of eliminating the smallest clusters, the choice of additional leaders turns out to reduce the size of the largest clusters as the leaders now better represent the descriptor distribution.

We have chosen not to recluster the largest clusters. The reason is that since large clusters typically occur in dense areas of the descriptor space, it is likely that reclustering a large cluster would simply move all the descriptors to a single cluster (or a few), resulting in that cluster becoming equally large as the removed one, or even larger.

4.4 Handling Multiple Query Descriptors

As each query is represented by a few hundred descriptors, it is possible to optimize query processing significantly. Instead of processing query descriptors one by one, resulting in (potentially repeated) random IOs, all descriptors are considered in a batched mode. First, all query descriptors are compared to the cluster index to determine which clusters are needed. Second, only those clusters are read, in order, and their descriptors compared only to the query descriptors that found the corresponding cluster among its b closest clusters.

Clustering		Search Time (sec)		Correct Matches (%)	
L	Time (min)	$L=1$	$L=2$	$L=1$	$L=2$
1	1,287.0	2.09	1.41	76.2	74.7
2	64.7	2.10	1.42	75.5	75.2

Table 1: Impact of L on clustering and search performance (small coll., 128KB clusters, $b = 5$, $a = 1$).

This method is more efficient, as clusters are read once and the IOs are largely sequential.

It is, of course, possible to go even further and process multiple query images at the same time, but we do not consider such optimizations in this study.

5 Performance Experiments

In this section, we first analyze in detail the effects of the various parameters using the smaller descriptor collection. Then we compare the performance of the clustering and search algorithms for the small and the large collections, using settings determined from the experiments.

All experiments were run on DELL PowerEdge 1850 machines equipped with two 3.2GHz Intel Pentium 4 processors, 2GB of DDR2-memory, 1MB CPU cache, and two 140GB 10Krpm SCSI disks. The machines run CentOS 5.0 Linux (2.6.18 kernel) and the ext3 file system. The software was implemented in C++ and compiled using g++ 4.1.2.

5.1 Impact of Cluster Index Depth

We start by studying the impact of L on the performance of the clustering and search algorithms. In the Cluster Pruning algorithm, the choice of L during clustering and search can be independent; in fact Chierichetti *et al.* used $L = 1$ during cluster construction and $L \geq 1$ during search [3].

In this experiment, we generated $l = 11,859$ clusters with an average size of 128KB (1,724 descriptors), using $L = 1$ and $L = 2$, and then searched $b = 5$ clusters for each query descriptor, both using $L = 1$ and $L = 2$. Table 1 summarizes the results. As the first column of the table shows, cluster creation is much more efficient using $L = 2$, taking only about 5% of the time required for $L = 1$. The next two columns, for search time, show that searching a two-level index is also significantly faster than searching a single level index, although the difference is much less pronounced.

The last two columns show the search quality. Not surprisingly, the best quality is obtained through clustering and searching using $L = 1$, which returns 76.2% of the correct matches (recall that our definition of a correct match is very strict). The most efficient combination, using $L = 2$ for clustering and search, returns 75.2% of the correct matches. The difference is only 30 images, or less than 1% of the query set size. Given the tremendous efficiency gains, which will only become more important as the collections grow larger, the loss of quality is acceptable. We therefore only consider clustering and searching with $L = 2$ in the rest of this section.

l (clusters)	Average Cluster Size (KB)	(desc.)	Creation Time (min)
2,964	512	6,898	23.3
5,928	256	3,449	38.2
11,859	128	1,724	66.0
23,719	64	862	97.8
47,438	32	431	146.0

Table 2: Impact of average cluster size on clustering time (small coll., $L = 2$, $a = 1$).

Note, again, that when studying the performance impact of L, Chierichetti *et al.* clustered the collection using $L = 1$ but searched it using $L = 2$ [3]. This is indeed the worst combination, according to our results.

5.2 Impact of Average Cluster Size

We now study the impact of the l parameter determining the number of clusters created, thus affecting the average cluster size. Table 2 shows the clustering time for a range of cluster sizes. As expected, having more (smaller) clusters results in a longer clustering process, as each descriptor must be compared to a greater number of representatives.

The impact on search time and result quality, however, is more complex. The expectation is that searching smaller clusters will be faster, but that the results may be poorer, in particular with very small clusters. On the other hand, while increasing average cluster size will initially yield better results, the expectation is that a "law of diminishing returns" will reduce the additional benefits beyond a certain point.

Figure 1 shows the average time required to search for each query image. As the figure shows, searching is most efficient for the smallest cluster sizes. For clusters of 32KB and 64KB the difference is negligible as the cost of selecting from the large number of cluster cancels out the reduced cost of reading and scanning the clusters. As clusters grow, however, the differences become more pronounced.

Interestingly, scanning two clusters ($b = 2$) with average size of 128KB is less time-consuming than scanning one cluster ($b = 1$) of 256KB; the same holds for 256KB clusters and 512KB clusters. This is because, with the smaller clusters, it is more likely that at least one of the clusters is in memory. Thus, reading additional clusters impacts efficiency more positively than having larger clusters.

Figure 2, on the other hand, shows the result quality of the search, for the same values of l and b. Note that, for clarity, the y-axis focuses on the range from 60% to 80% of correct matches. This figure again confirms our intuition and shows that most of the quality is achieved with clusters of 64–128KB. Combining the two figures and Table 2, we conclude that the best combination of clustering time, search performance and result quality is achieved using an average cluster size of 64KB or 128KB; we use 128KB in the remainder of our study.

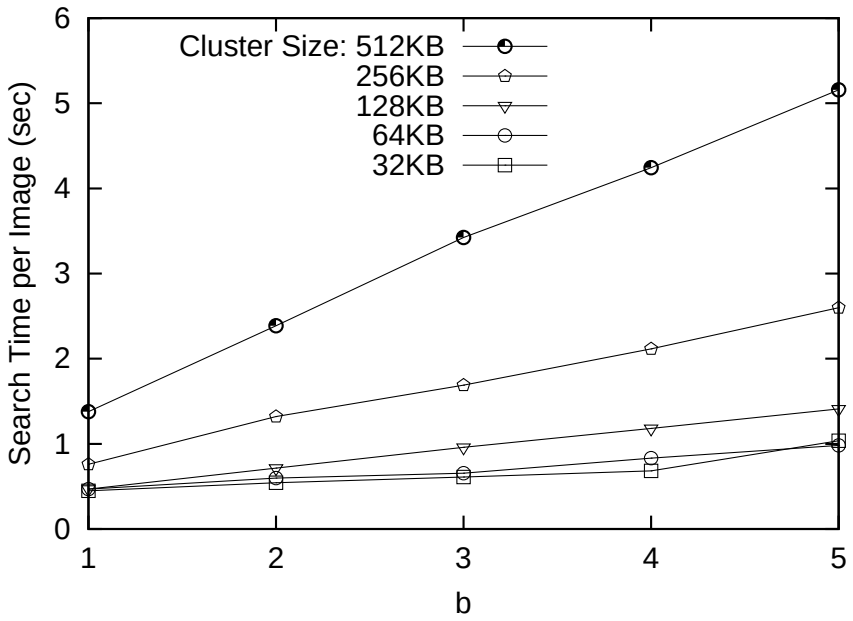

Figure 1: Impact of average cluster size on search time (small coll., $L = 2$, $a = 1$).

Note that the original algorithm at $L = 2$ would have created about 74,000 clusters of about 16KB each; as our results show, those clusters would be far too small and many.

5.3 Impact of Redundancy

We now turn to the trade off between the a and b parameters. As mentioned above, the expectation is that they should yield results of similar quality. This is confirmed by our results (not shown); for $a > 1$, only about 10 more matches are found than for the corresponding b.

With respect to search performance, the intuition is that using a should be slightly more efficient as it requires fewer (but larger) random disk operations, while using b is more flexible as b can be decided at query time. Our results, however, do not confirm this intuition. Figure 3 shows the impact of a and b on search performance for two different memory settings. Consider first the results when the main memory allocation is 2GB. As expected, the results are identical for $a = b = 1$, as this is the same configuration. Once $a > 1$, however, the performance becomes much worse than for corresponding settings of b. The primary reason for this difference is that when $a > 1$ clusters become much larger and therefore fewer can be cached in memory. Thus, each query must read most of its clusters from disk, while buffering performance is affected less by b.

To study the performance in a fair setting, we therefore reduced the memory allocated to the operating system to 750MB and repeated the experiment. With

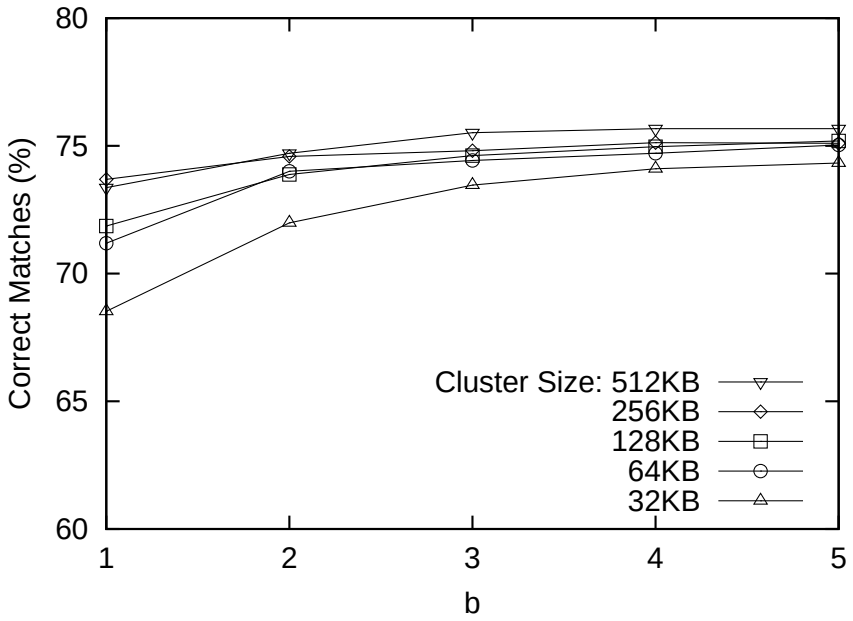

Figure 2: Impact of average cluster size on result quality (small coll., $L = 2$, $a = 1$).

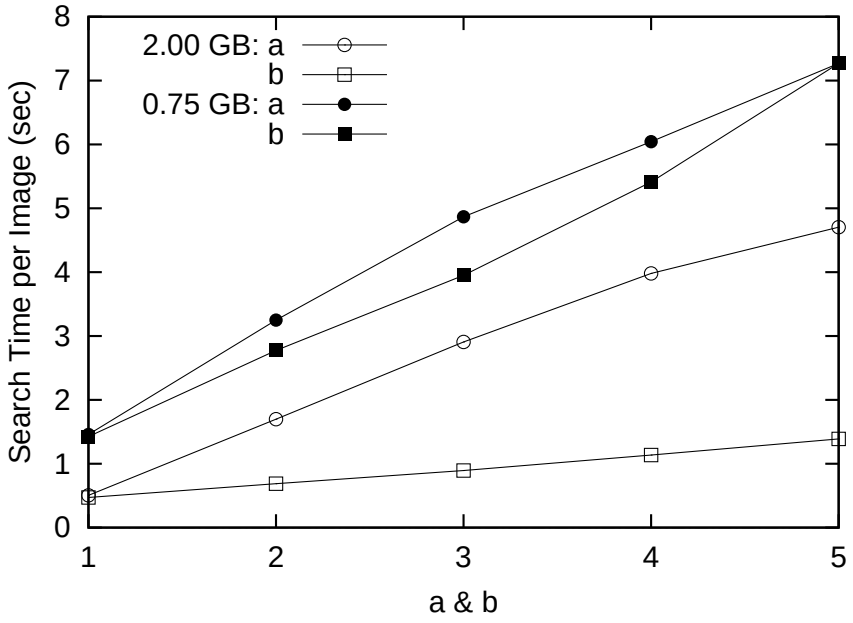

Figure 3: Impact of a and b on search time (small coll., 128KB clusters, $L = 2$).

this setting, both parameters are impacted by the buffer management performance, but varying b is still more efficient. We believe there are three main reasons for this. First, even though few clusters fit in memory, clusters are still smaller and the buffer manager is therefore more likely to find them in memory. Second, because each query consists of hundreds of descriptors, which read b times more clusters, and because clusters are read in sequence, disk reads are actually less random than expected. By varying a, on the other hand, fewer but larger clusters are read, and disk reads are spread over a larger area of the disk. Third, since clusters are often larger than the IO granularity of the operating system, each "logical" IO may result in many "physical" IOs. This occurs more often with the larger clusters generated using $a > 1$, which helps to explain the negative impact of a.

5.4 Impact of Cluster Size Distribution

The general idea for improving cluster size distribution is to intentionally choose $X\%$ extra leaders at the start of the clustering process. Once the collection has been clustered, we then remove the $X\%$ smallest clusters and insert their contents into the nearest remaining clusters. Figure 4 shows the resulting data distribution. The x-axis indicates how many additional clusters are created initially (percentage of the desired number of clusters). The y-axis shows the number of descriptors that fall into a given cluster size category; recall that the average size of clusters is 1,724 descriptors. As the figure shows, more than 10% of the data is initially ($X = 0$) either in very large or very small clusters, while only about 35% of the data is in the range from 1,000 to 2,000. As X increases, the largest and smallest clusters shrink, and contain about 4% when $X = 100$, while 60% of the data then falls within the range from 1,000 to 2,000.

Figure 5 shows the impact of varying X on the clustering time, search time, and result quality, compared to $X = 0$. As expected, clustering time increases as X is increased due to the additional distance calculations, and nearly doubles when $X = 100$. Search time, on the other hand, decreases due to the better size distribution of the clusters. Most importantly, however, result quality is only affect very slightly, as the number of correct matches only changes by ± 10.

5.5 Impact of Scale

The previous experiments have studied the impact of various parameters at a moderate scale (although a collection of 20 million descriptors is, after all, quite large compared to the typical collections studied in the literature). We have concluded that for optimal performance, we should set $L = 2$ and $a = 1$, generate clusters with average size of 128KB, and use b to improve result quality (optionally generating and then removing some extra clusters). We now apply these settings to a collection that is an order of magnitude larger and study the performance of the clustering and search algorithms with this larger collection. Note that in order to get a fair comparison of disk activity, we compared the search time of the large collection to the search time of the small collection with the 750MB configuration.

Since the collection is about 9.3 times larger, and cluster size is the same, there will be about 9.3 times as many clusters; as the depth of the index is the same, there will be about $\sqrt{9.3}$ times more cluster representatives at each level.

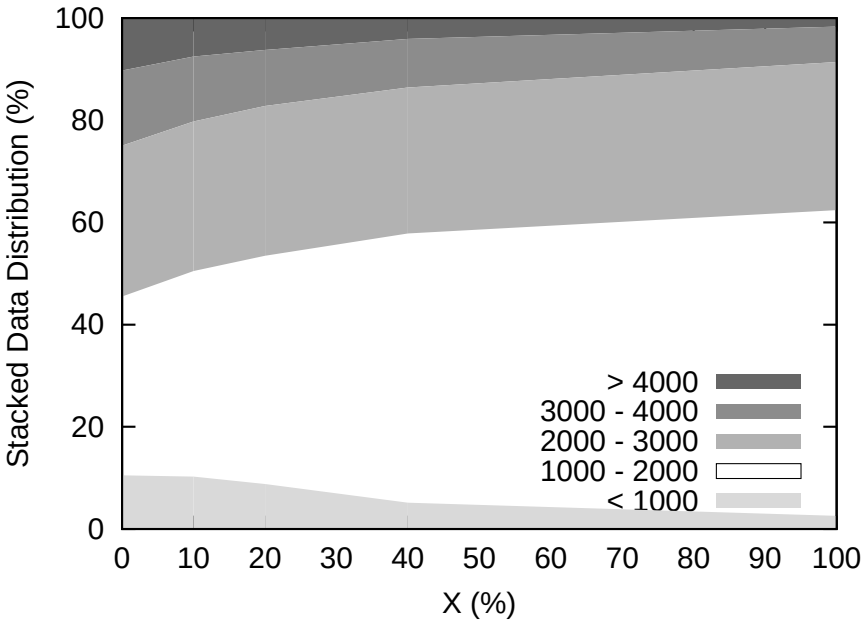

Figure 4: Data distribution for varying X (small coll., 128KB clusters, $L = 2$, $a = 1$).

Collection	Descriptors (millions)	Clustering Time (min)	Query Time (sec)	Matches (%)
Small	20.4	64.7	3.95	74.6
Large	189.6	2,344.7	8.82	74.3
Difference	≈9.3x	≈36x	≈2.2x	≈1x

Table 3: Comparison of the small and large collections (128KB clusters, $L = 2$, $b = 3$, $a = 1$).

We therefore expect that the cluster creation will take about $9.3\sqrt{9.3} \approx 28$ times more time, while the search should be affected much less. We also hope that the result quality will be largely unaffected.

Table 3 shows the results of the experiment. As the table shows, clustering is about 36 times more time-consuming, which is close to the expectation. Searching is just over 2 times slower, mostly due to the additional cost of scanning the index, but potentially also due to a slightly worse cluster size distribution. Most importantly, however, the table shows that only 10 images are lost when going to the larger collection, which is a reduction of about 0.3%. As each descriptor is compared to only $3 \times 1,724 = 5,172$ descriptors on average, when $b = 3$, or about 0.003% of the collection, this is an excellent result.

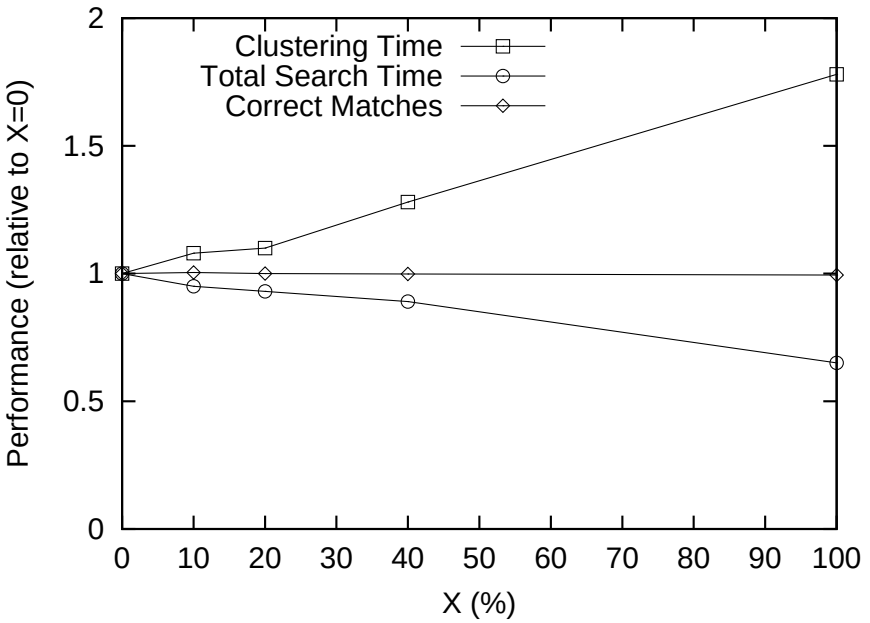

Figure 5: Relative performance for varying X (small coll., 128KB cl., $L = 2$, $a = 1$).

5.6 Summary of Results

Several lessons can be drawn from the above experiments. First, multilevel clustering is necessary when indexing large collections. It allows for very efficient clustering when the index is created before assigning descriptors to clusters. Note that at even larger scales, when scanning the index becomes costly, incrementing the depth of the hierarchy may be considered. Second, partitioning the collection into I/O sized clusters is best for efficiency. This, together with a more balanced distribution of clusters sizes reduces the time spent on I/Os. Third, reading more than one cluster at search time yields the best result quality. It also absorbs the inaccuracies of assignments of points to clusters and compensates for the losses in precision due to the multiple levels of the hierarchy. Furthermore, compared to large clusters, it increases the chances of finding a cluster in memory, avoiding I/Os. All in all, these extensions help Cluster Pruning to scale very well to quite large data sets.

6 Related Work

While there has been significant work on clustering data, the focus has typically been on identifying the natural clusters of the collection, rather than creating a cluster index for query processing. Aside from [11], using clustering for image retrieval has been investigated by the computer-vision community. One seminal approach to image retrieval, Video-Google [19], uses k-means to group descriptors into visual words, which are then indexed using information retrieval

techniques; in this case, the clusters are not used directly for query processing. Philbin *et al.* [16] concluded, much like Chierichetti *et al.*, that for this application result quality is enhanced when varying the extent of the search and/or the redundancy of the clustering.

Building on Video-Google, Nistér and Stewénius observed that the retrieval quality was increased when the visual vocabulary is significantly enlarged (to several millions) [13]. When k is very large, however, standard k-means fails. They thus proposed a hierarchical k-means approach, which is quite similar to Cluster Pruning, but builds clusters top-down. They first cluster data into a small number of partitions (typically 10) with the standard k-means. Then, they recursively build the next level of the cluster tree by applying again a k-means within each of the partitions independently, top-down. Eventually, it creates an L-levels hierarchy of k clusters per level. The cluster within which a query point falls is found by descending the tree. To compensate for assignment errors, data points may be assigned to more than one leaf. Nistér and Stewénius do not study the various options discussed in the Cluster Pruning approach. They subsequently addressed quality issues, by using multiple (15–20) clusterings together to ensure quality, requiring one disk IO per cluster for each query descriptor [6].

Accelerating the clustering of the data collection in the Video-Google context is also the goal of Philbin *et al.* [15]. Their clustering process is flat, similar to standard k-means. They basically reduce the number of representatives each point must be compared to, boosting the assignment and trading-off speed for (a small loss in) accuracy. They start by precomputing a large set of cluster representatives that get indexed into several randomized kd-trees. They assign a data point to its approximate closest representative by first probing each kd-tree with the point to cluster. They record the x best leaves for each tree, sorted on the distances to the separating hyperplanes. Then, the data point is assigned to the representative with the smallest such distance.

Overall, these methods [19, 16, 13, 15] have much in common with Cluster Pruning yet have quite specific properties. First, they never use the data *in* clusters, but rather the mapping between data points and cluster centers. Therefore, they are free to create poorly balanced clusters, and can rely on *tf–idf* schemes from information retrieval to compensate for differences in cluster cardinalities. Second, they also create a very large number of clusters since this, in turn, creates very sparse lists, as needed for efficient processing of inverted lists. Last, they are mostly main memory oriented. Therefore, an open question is whether Cluster Pruning and the extensions we propose here would be effective for applications like Video-Google.

7 Conclusions

Many content-based image retrieval systems and techniques rely on clustering to partition data, either for pre-processing or for data retrieval. Recently, the Cluster Pruning algorithm was proposed as a very simple, yet effective, approach for rapidly producing clusters of acceptable quality. Its simplicity and performance was a strong motivation to study its behavior in a large-scale image indexing context.

Building on Cluster Pruning, we have proposed three extensions which increase its performance at large scale. The first extension comes from the observation that disks can not be ignored and taking into account the IO granularity is a key factor to performance. This suggests to create clusters that contain, on average, enough data to entirely fill the operating system IO granule. The second extension comes from the observation that good search performance is obtained when clusters are better balanced. This can be achieved simply by creating extra clusters and reclustering the data in the smallest clusters. Third, many clustering algorithms have a high cost at cluster construction time because they cannot use any index to facilitate the assignment of points to cluster representatives. With Cluster Pruning, however, representatives are randomly picked beforehand. Therefore, we propose to use these representatives in a multi-level index to direct the assignment of data to clusters, dramatically reducing the clustering time.

Overall, we believe that, with our modifications, Cluster Pruning is a good basis for building large-scale systems that require a clustering algorithm. Not only is the algorithm fast, but it appears to produce clusters of acceptable quality, even at large scale.

References

[1] S.-A. Berrani, L. Amsaleg, and P. Gros. Approximate searches: k-neighbors + precision. In *Proc. CIKM*, New Orleans, LA, USA, 2003.

[2] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Comp. Science*, 1540:217–235, 1999.

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

[4] M. Datar, P. Indyk, N. Immorlica, and V. Mirrokni. *Locality-sensitive hashing using stable distributions.* MIT Press, 2006.

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

[6] F. Fraundorfer, H. Stewénius, and D. Nistér. A binning scheme for fast hard drive based image search. In *Proc. CVPR*, Minneapolis, MN, USA, 2007.

[7] Y. Ke and R. Sukthankar. PCA-SIFT: A more distinctive representation for local image descriptors. In *Proc. CVPR*, Washington, DC, USA, 2004.

[8] Y. Ke, R. Sukthankar, and L. Huston. Efficient near-duplicate detection and sub-image retrieval. In *Proc. ACM Multimedia*, New York, NY, USA, 2004.

[9] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. Scalability of local image descriptors: a comparative study. In *Proc. ACM Multimedia*, Santa Barbara, CA, USA, 2006.

[10] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE TPAMI*, 31(5), 2009.

[11] C. Li, E.Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clindex: Clustering for approximate similarity search in high-dimensional spaces. *IEEE Trans. on Knowl. and Data Engineering*, 14(4), 2002.

[12] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, 2004.

[13] D. Nistér and H.K Stewénius. Scalable recognition with a vocabulary tree. In *Proc. CVPR*, New York, NY, USA, 2006.

[14] F. A. P. Petitcolas et al. A public automated web-based evaluation service for watermarking schemes: StirMark benchmark. In *Proc. of Electronic Imaging, Security and Watermarking of Multimedia Contents III*, San Jose, CA, USA, 2001.

[15] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proc. CVPR*, Mineapolis, MN, USA, 2007.

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

[17] U. Shaft and R. Ramakrishnan. Theory of nearest neighbors indexability. *ACM Transactions on Database Systems*, 31(3):814–838, 2006.

[18] R. Sigurðardottir, H. Hauksson, B. Þ. Jónsson, and L. Amsaleg. A case study of the quality vs. time trade-off for approximate image descriptor search. In *Proc. IEEE EMMA workshop*, Tokyo, Japan, 2005.

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

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399