The Quality vs. Time Trade-off for Approximate Image Descriptor Search

Rut Sigurdardottir, Hlynur Hauksson, Björn Þór Jónsson, Laurent Amsaleg

The Quality vs. Time Trade-off for Approximate Image Descriptor Search*

Rut Sigurðardóttir
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
rut01@ru.is

Hlynur Hauksson
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
hlynur01@ru.is

Björn Þór Jónsson
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
bjorn@ru.is

Laurent Amsaleg
IRISA–CNRS
Campus de Beaulieu, 35042 Rennes, France
laurent.amsaleg@irisa.fr

Abstract

In recent years, content-based image retrieval has become more and more important in many application areas. Similarity retrieval is inherently a very demanding process, in particular when performing exact *searches. Therefore, there is an increasing interest in performing* approximate *searches, where result quality guarantees are traded for reduced query execution time.*

The goal of approximate retrieval systems should be to obtain the best possible result quality in the minimum amount of time. As a result, typical indexing strategies divide the data set into many data chunks. Minimizing the search time suggests to generate uniformly sized chunks to best overlap I/O costs with CPU costs. Maximizing quality, on the other hand, suggests to strongly limit the intra-chunk dissimilarity of data.

The paper addresses the question to what extent guaranteeing the query processing time, using uniform chunk sizes, compromises the quality of the results, and vice versa. Using a large collection of 5 million 24-dimensions local descriptors computed over more than 50 thousand real life images, we show that minimizing the query processing time may in fact lead to better quality of the intermediate results.

*This project is part of the *Eff*^{2} project on *Efficient and Effective Image Retrieval* (see http://datalab.ru.is/eff2). The *Eff*^{2} project is a cooperation between researchers at the IRISA laboratory in Rennes, France, and Reykjavík University, Iceland, and is partially supported by Rannís Technical Research Grant 030290004 and ÉGIDE Jules Verne Travel Grant 4-2003.

1. Introduction

In recent years, content-based image retrieval has become more and more important in many fields, such as medicine, geography, weather forecasting and security. Content-based image retrieval is typically implemented by mapping the images to multi-dimensional descriptors, which are then used in similarity searches to determine which images in the collection are most similar to a query image.

Query processing in content-based similarity retrieval is inherently a very demanding application. While many advanced multi-dimensional indexing methods exist, they invariably run into the infamous *curse of dimensionality* when faced with large collections of high-dimensional descriptors [19, 5]. In the best case, their performance degrades gracefully to that of a sequential scan of the collection.

In many cases, it has been observed that most of the eventual top results are found in the early stages of the query processing and that most of the search time is in fact spent on *guaranteeing* that the result of the query is indeed the best result that can be obtained. Because of the inherent difficulty of obtaining such a guarantee, there has recently been a surge of interest in approximate query processing algorithms, which trade the guarantee of a best result for reduced query processing time [18, 12, 11, 6, 3].

One example of such a search scheme is the "clustering for indexing" paradigm, which was originally proposed in [15]. In this approach some mechanism is used to divide the descriptor collection into chunks of similar data. An approximate search algorithm is then used that 1) globally ranks the data chunks based on the distance from the query point, 2) reads and processes the data chunks in the order of

the global ranking, and 3) applies an aggressive *stop-rule* to determine when enough time has been spent on the query processing. A simple and natural stop rule is to process only the n nearest chunks, and then return the result that has been obtained at that time as the final result of the query. Typically, by processing only a few of the data chunks, a close approximation of the eventual result is obtained.

1.1. Quality vs. Time

Given the stop rule described above, the goal of the approximate retrieval system should be to obtain the best possible result quality (typically measured using precision and/or recall) in the minimum amount of time. A variety of methods to form the chunks of data may be (and have been) used, but this tradeoff between the quality and response time has not been studied in detail.

Since the number of chunks to read is fixed, the response time of the search is primarily determined by the CPU cost of processing the descriptors of the chunks. While this cost is typically quite significant, e.g. when using the common Euclidean similarity measure, it can potentially be overlapped with I/O cost. As a result, the way to guarantee minimal query processing cost is to produce uniformly sized chunks, to balance the I/O and CPU cost of the search.

Uniformly sized chunks, however, may span arbitrarily large portions of the multi-dimensional space, depending on the density distribution of the data collection. Therefore, some of the query processing time may be spent on processing descriptors that are quite different from the query descriptor. The best way to guarantee result quality is therefore to limit the intra-chunk dissimilarity between descriptors, so that once a data chunk has been determined to be near the query descriptor, all of its contents are likely to be close as well.

Clearly, both quality and response time need to be considered, when dividing the descriptor collection into chunks. For example, by distributing descriptors to chunks in a round-robin manner, chunks of uniform size are obtained, but the quality will suffer. On the other hand, pairing each descriptor with one or a few of its closest neighbors will give very low intra-chunk dissimilarity, but the time to globally rank such small chunks will be excessive.

1.2. Contributions of the Paper

The main research question that this paper addresses is the following: *To what extent does guaranteeing the query processing time, using uniform chunk size, compromise the quality of the result, and vice versa?* In order to answer this question, we compare two different approaches to forming the data chunks that focus on guaranteeing response time and result quality, respectively.

For uniform chunk size, we have instrumented an SR-tree index [14] to generate chunks of uniform size, by varying the size of the leaves of the tree and generating one chunk from each leaf. The SR-tree makes some effort to keep similar descriptors together, but the emphasis is squarely on the chunk size.

To minimize intra-chunk dissimilarity, we have used the clustering algorithm of [3]. This algorithm focuses on guaranteeing the maximum diameter of the generated clusters, while making a best effort on the chunk size by allowing the user of the algorithm to control the number of clusters.

The main contribution of this paper is a detailed experimental study of the tradeoff between quality and query processing time of these two different approaches to forming chunks. We use a large dataset of 5 million 24-dimensional descriptors, created using one of the most advanced known image description schemes [1]. The results show that 1) the difference between the query processing time and quality of results using these two approaches is less than expected, and 2) focusing on the query processing time may actually give *better* early results than focusing on quality. Obtaining uniform chunk size is also far easier than guaranteeing intra-chunk dissimilarity.

These results indicate that for approximate similarity search, uniform chunk sizes should be the first priority, while making an effort to achieve good groupings within the chunks. Given the extremely long time required to generate high quality chunks for our collection, this is a very fortunate result, as performing such clustering for very large descriptor collections would simply be impossible.

1.3. Overview of the Paper

The remainder of the paper is organized as follows. Sections 2 and 3 review the two extreme chunk forming approaches employed in our performance study. Section 4 presents our experimental environment and Section 5 presents the experimental results. Finally, Section 6 discusses related work and Section 7 gives our conclusions.

2. Forming Chunks with the SR-tree

In order to build chunks of uniform size, we adapted the SR-tree of [14] to yield chunks, by making two minor changes to the code. First, we added a parameter to control the size of the leaves, and second, we added a method to generate chunks from the leaves, thus throwing away the upper levels of the tree. We used the static build method, as it was much faster and guaranteed uniform leaf size. Unfortunately, it requires the collection to fit in memory, which limited the data collection we could use.

The SR-tree produces roundish chunks of uniform physical size, and can be made quite scalable via standard sorting

and bulk-loading techniques. The chunks are not of high quality, however, as they tend to overlap significantly. Additionally, this approach does not handle outliers naturally (see Section 5 for a discussion on outlier handling in our experiments).

3. Forming Chunks with BAG

The second chunk forming strategy is a clustering algorithm called BAG[1] which is presented in [3]. BAG tries to create clusters of minimal volume in order to maximize the intra-cluster similarity of descriptors. It is derived from the first phase of Birch [20], and outputs hyper-spherical clusters, each being identified by its centroid and minimum bounding radius.

To produce clusters, the algorithm uses one key value, called MPI in the following, that represents the Maximum Possible Increment for radii. It starts by creating a cluster from each descriptor. Since the radius of each cluster is always maintained, they all have, at this stage, a radius of zero. Then, the algorithm scans the current set of clusters to see whether some clusters might be merged.

Two clusters can be merged if and only if the radius of the resulting cluster is smaller than the radius of the larger cluster plus the MPI value mentioned above. When two clusters are merged, the new centroid is computed as well as the new minimum bounding radius. Clusters that do not merge have their radius incremented by MPI (making their radius non-minimal).

Once all clusters have been analyzed (therefore either merged into larger clusters or having had their radius incremented), the algorithm reconsiders again the list of current clusters for potential new merges. Note that at each step, it is possible that no merge occurs (all clusters are too small and too far away) or that many merges take place. At the end of each step, the average number of descriptors per cluster is computed. Then, all clusters that hold less than some percentage of this average number (20% in the case of our experiments) are destroyed. All the descriptors held in destroyed clusters create again one-point clusters with a zero radius. These points might subsequently be merged into some neighboring clusters.

As time goes by, the algorithm reduces the number of clusters as their radius expands. Eventually, the number of clusters falls below a user-defined threshold and the algorithm terminates. At that time, the average population of clusters is computed. Clusters with too few descriptors are destroyed and the descriptors they were holding are considered to be outliers.

BAG does not use any indexing scheme to facilitate the merge process. Instead, it examines all existing clusters every time a cluster is checked for potential merges. As a result, the CPU time required for processing is very high and clustering descriptors takes a long time.

[1]This algorithm was not named in the original paper, so we refer to it using the first letter of the last names of the authors.

4. Experimental Environment

This section presents the experimental environment we used to investigate the tradeoffs between the two chunk-forming strategies mentioned previously. We first present the image description scheme we used to create our data sets. Then we detail the architecture used to index the chunks and the search algorithm.

4.1. Describing Images

The descriptors we used to describe our images rely on the fine-grained recognition scheme for grey-level images originally proposed by [9], extensively used and evaluated by [16] and extended to cope with color images in [1]. These descriptors are often called *local descriptors* because one descriptor encodes information that is local to a (small) area of an image. A descriptor is a 24-dimensional vector of floats and, in general, there are few hundreds of descriptors computed on each image. Images belonging to the collection are described off-line and typically stored sequentially in a single file. With this description scheme, similarity between images is implemented as a nearest-neighbors search in a Euclidean space [13].

These descriptors are very robust to image transformations and are able to detect similar elements in images despite orientation changes, translations, various illumination changes, partial occlusions, etc. [1]. They are particularly well suited to enforce robust content-based image searches for copyright protection [4].

4.2. Chunk Index Architecture

The chunk index consists of two files, a chunk file and an index file. The chunk file holds the descriptors computed over the whole image collection but these descriptors are grouped according to the specific chunk-forming strategy. All the descriptors belonging to one chunk are stored together on disk and the chunks are stored sequentially. The chunks are padded to occupy full disk pages.

The second file stores a simple index built over the chunk file. Each entry of the index stores the coordinates of the centroid of each chunk and the radius of the chunk, as well as its location in the chunk file. The order of the entries in the index is identical to the order of the chunks in the chunk file.

4.3. The Search Algorithm

For a given query descriptor, the search algorithm proceeds as follows. It first computes the distance between this query descriptor and the centroids of all existing chunks, and then ranks chunks according to their increasing distances. Based on this ranking, chunks are then accessed one after the other.

When a chunk is accessed, all the descriptors it contains are fetched into memory. The search then computes the distances between all the descriptors in the chunk and the query descriptor. This might in turn update the current set of neighbors.

Once all the descriptors of a chunk have been analyzed, the algorithm checks whether a new chunk must be read. The search might simply stop once n chunks have been processed or when a time threshold has been passed. If the search is asked to go to completion, however, it stops when k neighbors have been found and when the minimum distance to the next chunk is greater than the current distance to the k^{th} neighbor. This ensures that all nearest-neighbors have been found. Computing this minimum distance is the rationale for storing the radii of chunks together with their centroids.

5. Experimental Results

This section first describes our experimental setup, and then the two experiments we performed. The first experiment compared the response time and result quality of the two chunk forming strategies, while the second experiment was performed to determine the optimal chunk size to balance I/O and CPU cost. The section concludes with a discussion of the results.

5.1. Experimental Setup

We used the architecture described in Section 4 and the algorithms described in Sections 2 and 3. In the following we describe the descriptor collection used in our experiments, the query workloads studied, and the metrics gathered.

5.2. Descriptor Collection

We used a collection of 5,017,298 descriptors, derived from 52,273 real life images. Of these, 610 images come from a collection of still images[2], while the other images have been accumulated from various television broadcasts. As each descriptor has 24 dimensions, plus an identifier, each descriptor consumes 100 bytes, and the collection requires around 500MB of disk storage.

Table 1 shows information about the chunk indexes used in our study. The process to obtain these chunk indexes was as follows. The collection was first clustered using the BAG algorithm, yielding three different chunk indexes with varying sizes of clusters (SMALL, MEDIUM and LARGE). Because of the way the algorithm works, each clustering was generated from the other in succession. Outliers were then removed, as discussed in Section 3. The SR-tree was then used to form chunks of uniform size roughly equal to the average size of the BAG clusters. Table 1 shows how many outliers were identified for each chunk index, as well as the resulting number of chunks and their average size for both approaches.

Three things are worth noting about this process. Firstly, the SR-tree was used after removing outliers, as it has no mechanism for outlier removal. While this may seem unfair towards the BAG algorithm, we note that 1) outlier removal effectiveness is not under study here, and 2) we tested another simpler outlier removal scheme for the SR-tree, namely removing all descriptors with total length greater than a constant, and that method gave almost identical results. Additionally, knowing that no outliers are in the dataset would not change anything in the implementation of the BAG algorithm.

Secondly, the size given in Table 1 is uniform for all chunk-formations using the SR-tree. For the BAG algorithm, on the other hand, it is an average size. Due to the uneven distribution of the descriptors in the 24-dimensional space, there are a few chunks that are very large, and then there are many very small chunks. Figure 1 shows the size of the 30 largest chunks for each of the six chunk indexes of Table 1. The figure shows that the largest chunks are indeed very large (containing more than 500 thousand, 600 thousand and 1 million descriptors for the respective BAG chunk indexes of Table 1). As we will see later, this may have a serious effect on performance.

Thirdly, we note that the time it took to form the chunks was very different for the two approaches. The BAG algorithm took almost 12 days to generate the fewest clusters (as explained above, the other clusterings were generated along the way). The SR-tree, on the other hand, took about three hours to generate the chunk index with the most chunks, but less than two hours for the others. Surprisingly, the actual tree generation took at most 10 minutes, while the rest of the time was spent on calculating the centroid and radius of each chunk, as well as writing out the chunks.

5.3. Workloads

Queries to an image retrieval system will either have a good match in the collection or not. We have created two

[2] See http://www.irisa.fr/texmex/base_images/index.html.

Table 1. Properties of the BAG and SR-tree chunk indexes

	Descriptors and Outliers			BAG		SR-tree	
Chunk sizes	Retained Descriptors	Discarded Outliers	Percentage of Outliers	Number of Chunks	Descriptors per Chunk	Number of Chunks	Descriptors per Chunk
SMALL	4,471,532	545,766	12.2%	4,720	947	4,747	942
MEDIUM	4,595,312	421,986	9.2%	2,685	1,711	2,672	1,719
LARGE	4,652,022	365,276	8.0%	1,871	2,486	1,863	2,497

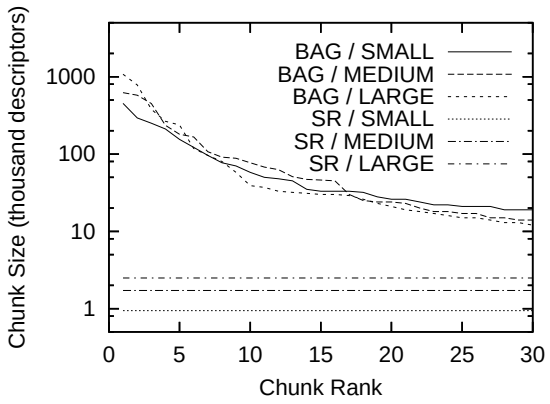

Figure 1. Size of the largest chunks

different workloads to simulate that. The first workload, called "DQ" (or "dataset queries"), consisted of 1,000 randomly selected descriptors from the descriptor collection. This workload simulated the cases when a match is found in the collection.

The second workload, called "SQ" (or "space-queries"), consisted of 1,000 queries, that were generated randomly from the 24-dimensional space as follows. For each dimension of the descriptors we analyzed the range of values in the descriptor collection. After discarding the top and bottom 5%, we stored the remaining value range of each dimension. The descriptors were then generated by selecting uniformly distributed values from these ranges. This workload simulated the cases when no match is found in the collection.

5.4. Measurements

The experiments were run on a Dell workstation with a 2.8GHz Pentium 4 CPU, 1GB of main memory, and a 40GB ATA disk. Each query in the workload was run once to each chunk-index in a round-robin fashion (to eliminate buffering effects) and the results of the thousand queries were averaged for each workload.

Metrics were gathered for response time and quality. The primary response time metrics were number of chunks read and elapsed time (or wall clock time), while other metrics were examined to understand the results. The primary quality metric was the precision within the top 30 images (when the number of returned images is fixed, recall and precision are the same metric). These metrics were logged after the processing of every chunk. As we always ran queries to conclusion, we were able to measure the quality of intermediate results.

To measure precision, we first ran a sequential scan of the collection, and stored the identifiers of the returned descriptors in a file. We then read this file for each measurement and used the descriptor list to calculate the precision of the intermediate result.

5.5. Experiment 1: Chunk Formation Strategies

The purpose of this experiment was to measure the effect of emphasizing uniform chunk size or uniform intra-chunk dissimilarity. Therefore, we examined the number of chunks and elapsed time required to find nearest neighbors with each chunk index.

Figure 2 shows how many chunks were required, on average, to find any number of nearest neighbors, with all six combinations of chunk sizes and chunk-forming approaches. Overall, the figure shows that much fewer chunks must be read to find the nearest neighbors for chunk indexes created using the BAG algorithm. For example, reading 5 chunks will on average yield 25-28 nearest neighbors for the BAG chunk indexes, but only 16-20 neighbors for the SR-tree chunk indexes. This is not surprising, as with the BAG indexes most queries will search their own chunk first and find there a high number of nearest neighbors. Figure 2 also shows that the average size of the chunks has only a small effect on the number of chunks required to find nearest neighbors.

Figure 3 shows the same information, but in this case for the SQ workload. While the overall shapes of the curves is similar, the important difference is that searching the SR-tree chunk indexes now gives slightly better results—mostly because with the BAG chunk indexes many more chunks must be read to obtain the same number of descriptors. This is explained by the fact that several small chunks may be read, instead of a few larger ones.

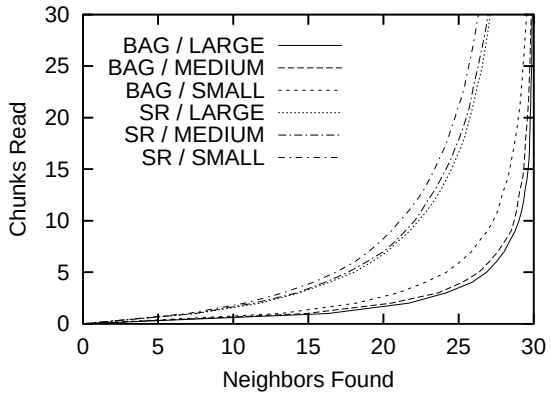

Figure 2. Number of chunks required to find nearest neighbors (DQ workload)

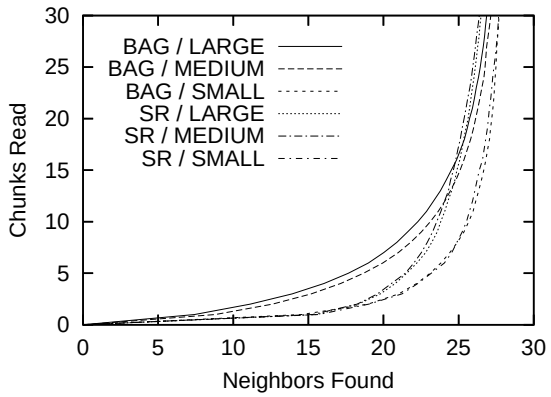

Figure 3. Number of chunks required to find nearest neighbors (SQ workload)

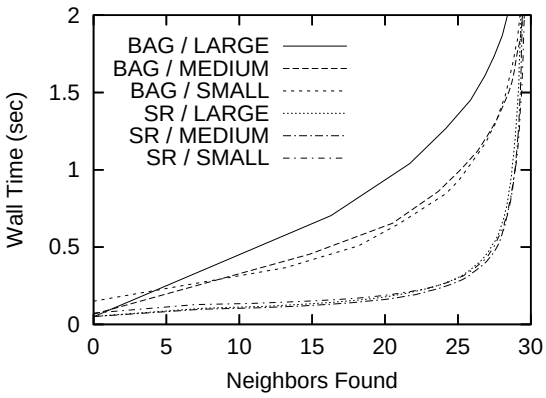

Figure 4. Elapsed time required to find nearest neighbors (DQ workload)

Turning to the elapsed time, shown in Figure 4 for the DQ workload, the story is quite different. As that figure shows, finding the first neighbors takes a much longer time with the BAG chunk indexes, than with the SR-tree chunk indexes. The reason is that while reading a large chunk is very beneficial for yielding many neighbors, it takes a great deal of CPU time to process such a large chunk. Since a single chunk is the natural granule of the search algorithm, results can only be returned once the required chunks have been processed. With the SR-tree chunk indexes, reading and processing each chunk takes only about 10 milliseconds, while processing the largest chunk of the BAG algorithm took as much as 1.8 seconds. Therefore many more chunks can be processed from the SR-tree indexes and more neighbors found within the first second.

Figure 5 shows the same information for the SQ workload. In this case, all approaches perform very similarly, as the BAG indexes are able to avoid reading the large chunks.

Figures 4 and 5 show that reading the chunk index takes about 50 milliseconds on average.[3] The figures also show, however, that the BAG indexes catch up after about two seconds. In fact the search is completed faster with the BAG indexes as shown in Table 2.

5.6. Experiment 2: Optimal Chunk Size

Having determined in the previous experiment that using uniformly sized chunks, created using the SR-tree, is preferable in terms of quickly obtaining reasonable approximate results, we wanted to know what chunk sizes gave the best performance tradeoff. We therefore used the SR-tree to

[3] Reading the chunk index of the BAG / SMALL configuration consistently took 150 milliseconds, compared to the 50 milliseconds for the other configurations. No explanation was found for this, but we suspect the disk layout of the index.

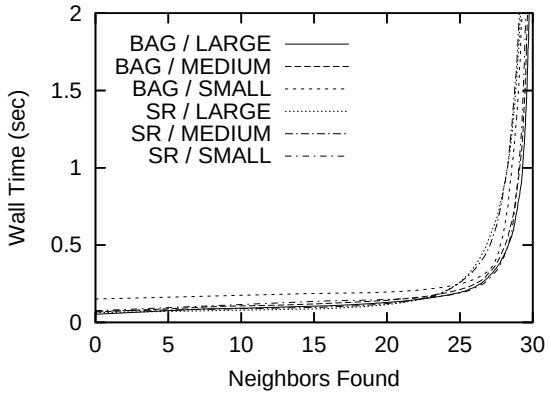

Figure 5. Elapsed time required to find nearest neighbors (SQ workload)

Table 2. Time to completion (seconds)

	BAG		SR-tree	
Chunk sizes	DQ	SQ	DQ	SQ
SMALL	39.5	44.6	45.0	45.0
MEDIUM	23.4	26.7	31.3	31.2
LARGE	16.7	20.3	25.2	25.5

create 16 chunk indexes with variable chunk sizes from the collection of 4,471,532 descriptors, and ran the same two workloads against each of these chunk indexes.

Figures 6 and 7 show the time required to find an increasing number of nearest neighbors for each of these chunk sizes, for the two workloads. As the figures show, the performance is quite similar for a wide range of cluster sizes, in particular when it is acceptable to expect not to find all neighbors. Chunks that are in the range of 1,000 to 10,000 descriptors all result in similar performance. This result is corroborated by the fact that the chunks of the previous experiment ranged from 942 to 4,747 descriptors.

5.7. Discussion

At least four important lessons may be learned from the results of these experiments. The first lesson is that relaxing the requirements for precise answers may yield significant improvements in response time. In our experiments, most of the 30 nearest neighbors were found in the first 1-2 seconds, while guaranteeing a correct result took between 16 and 45 seconds.

The second lesson is that the elapsed time is a more natural stop rule than the number of chunks read, as with the latter variably sized chunks may lead to variable query execution time.

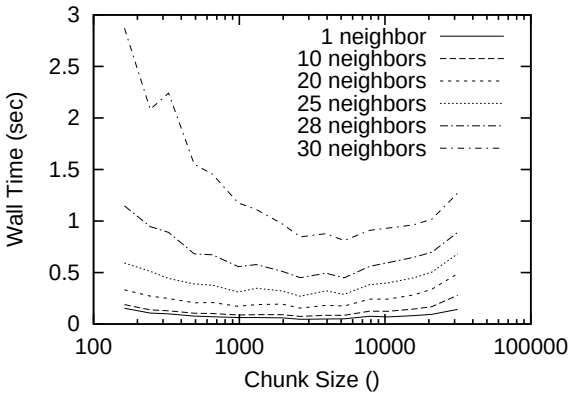

Figure 6. Effect of different chunk sizes (DQ workload)

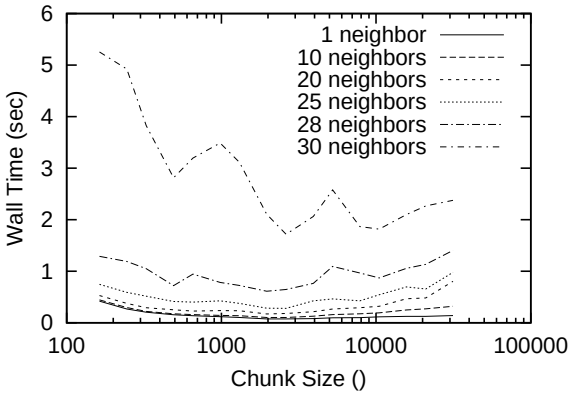

Figure 7. Effect of different chunk sizes (SQ workload)

The third lesson is that the query processing has similar performance and result quality for a range of chunk sizes. It is therefore not necessary to make all chunks the exact same size, but rather to avoid very small and very large chunks.

Finally, by comparing the results for elapsed time and recalling the excessive time taken to generate the BAG indexes, it is clear that the energy spent on creating dense chunks is largely wasted, and that an algorithm to form chunks for very large collections must indeed focus first on chunk size, and second on the intra-dissimilarity.

6. Related Work

Using clustering for indexing was originally proposed in [15]. They proposed to use the Cluster Forming (CF) algorithm to generate the clusters, and compared it to using the TSVQ algorithm [10]. Given a collection of about 450 thousand 48-dimensional histograms, CF outperformed TSVQ. The CF algorithm, however, may generate arbitrarily shaped clusters for two reasons. First, it is designed to detect clusters of arbitrary shape, and it has a very loose notion of adjacency of data. Second, the implementation had a hidden parameter to control the maximum size of clusters [17]. Given that segments of the multidimensional space are processed in the order of how many data points are contained within that segment, natural clusters may be broken up arbitrarily using this parameter. Thus, while it yields clusters of uniform physical size, they may have completely arbitrary shapes. Therefore we chose not to use CF for our study. We can point out, however, that CF is reasonably efficient—in [17] we have reported that building an index for a collection of 500 thousand 24-dimensional descriptors took 2–6 hours, depending on parameters.

Many approaches to approximate NN-searches have been proposed. Weber and Böhm with their approximate version of the VA-File [18] and Li et al. with Clindex [15] perform approximate NN-searches by interrupting the search after having accessed an arbitrary, predetermined and fixed number of chunks. Ferhatosmanoglu et al., in [8], combine this with a dimensionality reduction technique. With this technique, it is possible to improve the quality of an approximate result by either reading more chunks or by increasing the number of dimensions for distance calculations.

Other works use geometrical approximations of chunks for their NN-searches. They typically account for an additional ε value when computing the distances to chunks, making chunks somehow "smaller". In [18], Weber and Böhm present the VA-BND in which ε is empirically estimated by sampling database vectors. The AC-NN scheme for M-Trees presented in [6] relies on a single value ε set by the user.

Finally, some works enforce approximate NN-searches thru probabilistic approaches. DBIN [2], for example, exploits the statistical properties of data and clusters data using the EM (Expectation Maximization) algorithm. It aborts the NN-search when the estimated probability for a remaining database vector to be a better neighbor than the ones currently known falls below a predetermined threshold. P-Sphere Trees [12] investigate trading off (disk) space for time when searching for the approximate NN of query points. In this scheme, vectors belonging to overlapping hyperspheres are replicated. Hyperspheres are built such that the probability of finding the true NN of the query point can be enforced at run time by simply having the search identifying the nearest center and solely scanning the corresponding hypersphere. Both approaches are unable, however, to place any guarantees beyond the first nearest neighbor.

Recently, a very different approach to approximate searches has been published. With the Medrank algorithm [7], all descriptors are projected onto a set of random lines. Then, the database elements are ranked based on the proximity of the projections to the projection of the query. A rank aggregation rule picks the database element that has the best median rank as being, with a high probability, the true nearest neighbor of the query point. The next best median rank gives the second nearest neighbor of the query point and so on. One of the very nice properties of this algorithm is that it is I/O bound (and I/O optimal, as proven in [7]) because the algorithm is based on the aggregation of ranking rather than distance calculations.

7. Conclusions

In recent years, content-based image retrieval has become more and more important in many application areas. Similarity retrieval is inherently a very demanding process, in particular for *exact* searches. Therefore, there is an increasing interest in performing *approximate* searches, where result quality guarantees are traded for reduced query execution time.

The goal of approximate retrieval systems should be to obtain the best possible result quality in the minimum amount of time. As a result, typical indexing strategies divide the data set into many data chunks. Minimizing the search time suggests to generate uniformly sized chunks to best overlap I/O costs with CPU costs. Maximizing quality, on the other hand, suggests to strongly limit the intra-chunk dissimilarity of data.

The paper addresses the question to what extent guaranteeing the query processing time, using uniform chunk sizes, compromises the quality of the results, and vice versa. Using a large collection of 5 million 24-dimensions local descriptors computed over more than 50 thousand real life images, we have shown that minimizing the query processing time may in fact lead to better quality of the intermediate

results.

We have already described a collection of over 192 thousand images, resulting in over 220 million descriptors (over 22GB of data). We are planning to implement a multi-descriptor search algorithm for local descriptors and run against this collection. Our results indicate that we should use a clustering algorithm which keeps uniform chunk size as the first priority, but attempts to achieve the smallest possible intra-chunk dissimilarity. Given the extreme size of the collection, this is a very fortunate result, as focusing on the intra-chunk dissimilarity would be simply impossible.

References

[1] L. Amsaleg and P. Gros. Content-based retrieval using local descriptors: Problems and issues from a database perspective. *Pattern Analysis and Applications*, 4:108–124, 2001.

[2] K. Bennett, U. Fayyad, and D. Geiger. Density-based indexing for approximate nearest-neighbor queries. In *Proceedings of the 5th ACM International Conference on Knowledge Discovery and Data Mining*, San Diego, CA USA, 1999.

[3] S.-A. Berrani, L. Amsaleg, and P. Gros. Approximate searches: k-neighbors + precision. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, New Orleans, LA, USA, 2003.

[4] S.-A. Berrani, L. Amsaleg, and P. Gros. Robust content-based image searches for copyright protection. In *Proceedings of the First ACM International Workshop on Multimedia Databases*, New Orleans, LA, USA, 2003.

[5] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces—index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[6] P. Ciaccia and M. Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the IEEE Conference on Data Engineering*, San Diego, CA, USA, 2000.

[7] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Diego, CA, USA, 2003.

[8] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proceedings of the IEEE Conference on Data Engineering*, Heidelberg, Germany, 2001.

[9] L. M. J. Florack, B. M. ter Haar Romeny, J. J. Koenderink, and M. A. Viergever. General intensity transformations and differential invariants. *Journal of Mathematical Imaging and Vision*, 4(2):171–187, 1994.

[10] A. Gersho and R. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, 1991.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, 1999.

[12] J. Goldstein and R. Ramakrishnan. Contrast plots and P-Sphere trees: Space vs. time in nearest neighbor searches. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[13] R. Ý. Grétarsdóttir, S. H. Einarsson, B. Þ. Jónsson, and L. Amsaleg. The *Eff*^{2} image retrieval system prototype. In *Proceedings of the IASTED International Conference on Databases and Applications (DBA)*, Innsbruck, Austria, 2005.

[14] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Tucson, AZ, 1997.

[15] C. Li, E. Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[16] C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):530–534, 1997.

[17] Á. G. Valgeirsson, B. Erlingsson, Í. S. Einarsson, B. Þ. Jónsson, and L. Amsaleg. Using clustering to index image descriptors: A performance evaluation. Technical report, Reykjavík University, 2003.

[18] R. Weber and K. Böhm. Trading quality for time with nearest neighbor search. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, Konstanz, Germany, 2000.

[19] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, New York, New York, USA, 1998.

[20] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, 1996.