REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

The NV-Network: A Distributed Architecture for High-Throughput Image Retrieval

Friðrik Heiðar Ásmundsson

Master of Science
August 2006

Supervisor:
Björn Þór Jónsson, Ph.D
Associate Professor

Reykjavík University - Department of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

The NV-Network: A Distributed Architecture for High-Throughput Image Retrieval

by

Friðrik Heiðar Ásmundsson

Thesis submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of
Master of Science

August 2006

The undersigned hereby certify that they recommend to the Department of Computer Science at Reykjavík University for acceptance this thesis entitled **The NV-Network: A Distributed Architecture for High-Throughput Image Retrieval** submitted by Friðrik Heiðar Ásmundsson in partial fulfillment of the requirements for the degree of **Master of Science**.

____

Date

____

Björn Þór Jónsson, Ph.D, Supervisor
Associate Professor, Reykjavik University, Iceland

____

Laurent Amsaleg, Ph.D, Co-Supervisor
Assistant Professor, IRISA/CNRS, France

____

Hjálmtýr Hafsteinsson, Ph.D
Professor, University of Iceland, Iceland

Date

Friðrik Heiðar Ásmundsson
Master of Science

Abstract

Recently, the computer vision community has started a trend towards advanced image description schemes, where an image yields many local descriptors, each describing a small area of the image. These new schemes are not satisfactorily served by traditional multi-dimensional indexing methods and call for new and advanced database support techniques.

In previous work, we have proposed a new indexing structure, the NV-tree, which repeatedly segments the descriptor collection based on projections to random lines. Although using the NV-tree yields performance which is orders of magnitude faster than previous approaches, it is still unsuitable for high-throughput environments.

Therefore, we present the NV-Network which is a distributed architecture built around the NV-tree. The NV-Network mirrors the NV-tree across many worker machines and uses a coordinator machine to manage the system and balance the workers. Our system is designed to be scalable, effective and reliable to hardware failures. The system also has moderate hardware requirements, keeping the hardware cost to a minimum. Experiments show that using the NV-Network results in a very high throughput image retrieval, without any effect on result quality.

Útdráttur

Rannsóknir á sviði tölvusjónar hafa nýlega lagt áherslu á víðværar myndlýsingar þar sem hverri mynd er lýst af mörgum lýsingum. Fyrri vísar styðja ekki þennan fjölda af lýsingum og því er mikilvægt að næsta kynslóð vísa styðji þennan háa fjölda.

Í fyrri verkum kynntum við NV-tréð, sem endurkvæmt varpar lýsingunum í hólf byggt á vörpunum á slembi línur. Þó að NV-tréð sé margfalt hraðvirkara en fyrri vísar er það engu að síður of hægvirkt til að nota í afkastamiklum kerfum.

Í ljósi þess, kynnum við NV-Netið sem er dreift kerfi smíðað utan um NV-tréð. NV-Netið speglar NV-tréð og notar samhæfistjórnun til þess að stjórna kerfinu og dreifa álagi. NV-Netið er hannað til þess að vera skalanlegt, hraðvirkt og áreiðanlegt gagnvart vélbúnaðarbilunum. Kerfið gerir einnig lágar vélbúnaðarkröfur sem halda kostnaði í lágmarki. Mælingar sýna að NV-Netið gefur af sér hámarks afköst án þess að fórna gæði niðurstaðna.

Acknowledgements

When I started college, there was never any doubt what I wanted to do in the future; I wanted to study Computer Science at Reykjavík University. When beginning my studies here I encountered great people, and a working environment and atmosphere which met all my expectations, and then some.

I would like to thank Björn Þór Jónsson for his very helpful advice, his great support over the years and all the time he spent reading over this thesis and the publications we submitted during the last couple of years. I also thank Laurent Amsaleg for his work and input on the publications we have submitted.

I especially have to thank Herwig Lejsek for his friendship and the great teamwork when designing the prototype of the NV-Network. Our lively discussions over the years are to me a valuable memory; thanks!

Thanks to Árni for his input on various C++ topics, and thanks to the technical staff at Reykjavík University, especially Arnar and Marteinn, for their technical support and their great attitude. These guys are always willing to help! I also would like to thank the Netlab members for their input when designing the NV-Network and Ágúst for helping me setting up the web service.

Lastly, I would like to thank my wife for accepting my working hours, her parents for letting us stay at their home, and my parents for their support over the years.

Publications

A part of the background material in Sections 2 and 3 of this thesis are adapted from *"Scalability of Local Image Descriptors: A Comparative Study"* presented at ACM Multimedia held in October 23, 2006 in Santa Barbara, CA, USA.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Content based image retrieval (CBIR) systems are used to search for digital images by their contents. By "Content Based", we mean that the search extracts and uses information from the images themselves, rather than using keywords or other types of meta-data. These systems are applicable to institutions and users that gather such large collections of digital images that the traditional way of classifying them by keywords becomes too time consuming and hard to maintain.

Images are handled by the computer as a sequence of pixels, but pixel sequences are not amenable to searching. The images must therefore be pre-processed to allow more appropriate data representations. The pre-processing step typically encodes features such as shape, color and contrast into a vector of numbers. These vectors are called descriptors and are usually of high dimension.

Descriptors can be divided into two categories depending on how many descriptors are computed from each image. If only one descriptor is computed (or a few), it is called a global descriptor. On the other hand, if many descriptors are computed, they are called local descriptors. Global descriptors cannot detect similar elements between images, such as regions or objects; they are, as a result, not adequate for today's CBIR systems. More accurate results can be achieved by using local descriptors.

Local descriptors are computed by scanning the image for interest points, computing one descriptor for each. The interest points are chosen in such a way that similar images are likely to have similar interest points. After the interest points have been found, the descriptors are computed from the signal around each interest point. Performing a similarity search with local descriptors requires finding the nearest neighbors for each query descriptor and aggregating the results.

1.1 Scalable Database Support

Although the nearest neighbor query for each query descriptor may be run somewhat efficiently, the fact there may be hundreds of such query descriptors results in very inefficient query execution, even when using multidimensional indexing techniques. In fact, until recently, no indexing method had been shown to beat the sequential scan for image retrieval using local descriptors.

In recent work, we have proposed a new indexing technique, called the NV-tree. Our indexing strategy is based on repeatedly segmenting the descriptor collection into overlapping segments and projecting the data in the segments onto lines through high-dimensional space. While the index size grows, due to the overlapping segments, our approach requires only two disk reads per query descriptor, resulting in a fixed-time query evaluation, regardless of collection size. Although the NV-tree is already extremely fast, its performance and throughput can still be improved. It is currently I/O bound, and therefore it can be made even faster by distributing the disk operations among several machines.

There are primarily two ways to distribute the system: 1) by distributing disk operations to a set of independent machines (shared-nothing), and 2) by simulating shared-memory functionality to minimize disk operations. Because of the way local descriptors may be independently evaluated, this application lends itself very well to a shared-nothing architecture. Since shared-memory functionality is typically expensive (or complex), while building shared-nothing clusters is easy, we chose the shared-nothing approach.

1.2 Distributed CBIR

Since centralized CBIR systems have been found so difficult to implement efficiently, researchers have turned to distributed systems. In this area, most effort has been spent on algorithms required for Peer-2-Peer (P2P) retrieval applications. A P2P network is an administration free, ad-hoc network of equal peers that simultaneously function as both clients and servers to other peers. Every peer indexes its own collection and publishes it to the network. When a peer wants to search, it computes the query descriptors locally and broadcasts the query to all the peers in the network. Then each peer must perform a similarity search on its local data and return the results to the issuer of the query. Unfortunately, because there is no central management, there is no way to control a P2P network after it has started.

Compared to P2P systems, much less effort has been spent researching CBIR using a cluster of computers in a centralized environment. Since these systems are managed centrally, their collections can be administrated more securely, which is very important for many corporations and institutions. Unfortunately, because of the lack of scalable database support, designing centralized systems such that they are both efficient and effective, has been considered very hard. The only research activity we could find in this area was a scatter and gather architecture for a sequential scan [5]. This architecture had several flaws: first it did not scale well since all machines used a sequential scan; second, throughput decreased when many searches ran concurrently; and third, the hardware cost was very high, since very many machines had to be bought.

With the arrival of the NV-tree, an opportunity opened up for creating both an efficient and effective distributed CBIR system. Therefore, we have developed the NV-Network, which mirrors the NV-tree indices across many worker machines and distributes the query descriptors among these machines. Although this system is centralized, it overcomes the scalability issues of earlier systems since it is based on a scalable indexing technique. Additionally, since our system is using local descriptors, the queries can easily be split and distributed to the worker machines.

1.3 Contribution of the Thesis

In this thesis we present the NV-Network, which is a distributed architecture for high-throughput image retrieval with local descriptors. When designing the NV-Network, we made a list of requirements we wanted the system to have. First, it should be very scalable and able to handle many concurrent queries. Second, it should be easy to manage and administrate. Third, it should be as fault tolerant as possible with respect to hardware failures. Fourth, it should take advantage of fast hardware while allowing cheap hardware to co-exist. Fifth, we wanted to make as few changes to the NV-tree search as possible, and if possible, make the system independent of the underlying indexing technique.

We describe the architecture of the NV-Network in detail and give reasoning for various design decisions made during the development and testing phase. The NV-Network consists of four major components. The client issues the search; the server manages the search status; the coordinator manages the workers; and finally the workers perform the individual NV-tree searches.

To give an indication of how the system performs and behaves under various workloads we present extensive measurements using many different setups. We show that the NV-Network scales at least linearly with respect to the number of worker machines used, is scalable to hundreds of worker machines, achieves very high throughput, even when under very heavy load, and does not sacrifice search quality for speed.

1.4 Overview of the Thesis

The remainder of this thesis is organized as follows. Section 2 reviews the necessary background information and problems related to CBIR and Section 3 describes the NV-tree. Section 4 presents the NV-Network architecture in detail. Section 5 presents our experimental setup and performance results. Finally, Section 6 concludes and outlines future work.

Chapter 2

Background

We start by describing the current state-of-the-art local descriptor schemes in Section 2.1, followed by a discussion on current indexing techniques for CBIR using local descriptors in Section 2.2. Section 2.3 describes previous work for CBIR in a distributed environment.

2.1 Local Descriptors

Performing a similarity search in a CBIR system using local descriptors requires finding the nearest neighbors for each query descriptor. In order to do so, the system must compare each descriptor in the database to each query descriptor and compute the distance between them. When all the nearest neighbors have been found for each query descriptor, each of these neighbors gives a vote to the image it belongs to. The images are then ranked by how many votes they received during this process. The image which gets the most votes is considered to be the most similar to the query image, the image which received the second most votes is considered to be the next most similar image, and so on.

Local descriptors are typically computed on small areas of images in a two step process. The first step identifies points of interest in the image and the second step computes the values of the descriptors using the characteristics of the signal around each point of interest. Local descriptor schemes differ in the way points of interest are determined, in the number of points subsequently used and in the way the image signal is manipulated to compute each descriptor.

Several local descriptor schemes have been proposed. The ROT-DIAG-TRANS scheme by Amsaleg and Gros [1] uses the Harris point detector [14] and generates vectors of 24 dimensions. This scheme has been shown to be quite insensitive to affine transformations, rotations, cropping, color variation, mirroring, various illumination changes, partial occlusions, etc. It is, however, sensitive to image scaling, JPEG compression and various filtering methods [13].

Lowe [20] proposed the SIFT descriptors which transforms the image data into scale-invariant coordinates relative to local features. This means its features are invariant to image scaling, rotation, and robust to changing viewpoints. This scheme starts by progressively Gaussian blurring the original image at different scales, resulting in a series of Gaussian blurred images. Then, each pixel in these images is compared to its direct neighbors to find interest points at local extremas. To compute the descriptors around the interest points, a histogram of gradient directions is encoded into a 128-dimensional vector.

The PCA-SIFT descriptors proposed in [16] use the output of Lowe's SIFT descriptors and reduce the dimensionality using Principal Components Analysis (PCA). This is done by projecting the data to an offline trained eigenspace, which results in a 36 dimensional descriptor. This offline projection matrix for PCA-SIFT is built beforehand, using a subset of the descriptor collection.

Lejsek et al. [13] proposed a new descriptor scheme, the *Eff*^{2} descriptors, which borrow heavily from the SIFT methodology proposed by Lowe. These descriptors are constructed, however, with an aim for extreme scalability by restricting the number of descriptors, focusing the efforts on high-level descriptors and by studying the location of descriptors and adjusting parameters such that corresponding descriptors are closer to each other. In [13] the *Eff*^{2} descriptor scheme was compared to the three above mentioned descriptor schemes. There, it was shown that the *Eff*^{2} descriptor scheme yields the best results when studied using an image copyright violation workload.

Although the NV-Network can work with any descriptor scheme, we have chosen to use the *Eff*^{2} descriptor scheme in this thesis since it has been shown to give the best results.

2.2 Scalable Database Support

Up until recently, scalable database support for local descriptors has been sorely lacking. It was shown in [1] that advanced multi-dimensional indices, such as the Pyramid technique [3] and the VA-file [24], fail to beat a sequential scan of the whole descriptor collection. Two approaches specifically geared to local descriptor retrieval were proposed in [4] and [17]. The retrieval scheme proposed in [4] was based on pre-clustering the data and running approximate nearest neighbor queries, and was similar in spirit to the Clindex approach [19], while the scheme proposed in [17] was based on locality sensitive hashing [11]. Neither approach has been shown (nor is likely) to scale to very large collections.

As far as we know, only three approaches achieve efficient query processing for large collections of local descriptors. In [15], a system was proposed to verify, in real time, whether a video broadcast on TV comes from a reference collection containing over 40,000 hours of video. It uses a smart encoding of local descriptors together with a statistical similarity search which uses a Hilbert space filling curve. The processing time is sub-linear in collection size for reasonable collection sizes (albeit with an asymptotic linear behavior).

A second scheme that achieves efficient query processing is the PvS-framework [18]. The PvS-framework is heavily based on the OMEDRANK algorithm published by Fagin et al. in [9]. The PvS-framework consists of two key parts. First, it transforms costly nearest neighbor searches in multi-dimensional space into efficient unidimensional B+-tree accesses using a combination of random projections of vectors to random lines and partitioning of the projected space. Second, descriptor distance is computed efficiently using median rank distance, which approximates the expensive Euclidean distance function [9].

Based on the experience gained with the PvS-framework, we have designed a more sophisticated and general index which is also based on ranking, projections and partitions. This index, called the NV-tree, yields better performance and space utilization, is better able to capture the real distribution of data by self-tuning the projection and partitioning strategies, copes with on-the-fly updates of the descriptor collections, can be used stand-alone or by aggregating the results from two or more indices, and lends itself effectively to distributed processing to further reduce response times [12]. All in all, the NV-tree yields efficient query processing and good result quality with extremely large descriptor collections.

Because of the above mentioned qualities of the NV-tree, we have chosen it as our indexing technique to use in the NV-Network. Note that the NV-Network is independent of the indexing technique used. It is, however, beneficial for performance if the selected indexing technique offers a constant evaluation time per query descriptor, as the NV-tree does. The NV-tree is described further in Section 3.

2.3 Distributed CBIR

Most CBIR papers discussing distribution issues focus on algorithms required for Peer-2-Peer (P2P) retrieval applications. A P2P network is an administration free, ad-hoc network of equal peers that simultaneously function as both clients and servers to other peers. With each peer joining a P2P CBIR system, more images and processing power is added to the network. Each peer indexes its own collection and publishes it to the network.

P2P retrieval applications offer advantages of resource utilization, increased reliability and comprehensiveness of information [23]. They are however sensitive to query flooding, since each peer is forced to waste resources in handling irrelevant queries. The major problem of P2P retrieval applications, however, is identifying (or predicting) peers in a network which contain information relevant to the query. A first approach is to broadcast the query to all the peers in the network. Then each peer must perform a similarity search on its local data and return the results to the issuer of the query.

Broadcasting the query to all peers in the network brings an unbearable network and processor load to the network. Therefore doing efficient query routing for P2P CBIR systems is of great importance to make the network scalable. Balakrishnan et al. define this problem as follows [2]: “How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy?" Recent algorithms that try to address this problem include:

- Firework Query Model (FQM) proposed in [22] is a method for clustering together peers that share similar properties. When a peer initiates or receives a query message, the query is routed selectively according to the content of the query, to its designated cluster. Once there, the query message is broadcasted to all peers inside the cluster. This method was extended in [23] by allowing more than one signature to each peer, which resulted in a better result quality. Still, the main weakness of the FQM is the limitation of searching in only one target data cluster per query.
- PlanetP, proposed in [6] summarizes the content of each peer using a bit array called a bloom filter. This bloom filter is then distributed across the P2P network via a rumor spreading algorithm (see [7] for details). Using rumoring enables PlanetP to reduce the amount of information exchanged between nodes.

Since P2P networking involves the discovery of relevant information from within a potentially extremely large pool of peers, image retrieval quality will only be as good as the information inside the selected peers. So far, this problem has been circumvented by manually categorizing the images in every peer and clustering the collections based on their categories. Therefore, the search quality is heavily dependent on the quality and accuracy of the categorization. We find this a major downside with P2P CBIR systems.

If the problem of searching in the best possible peers is solved, researchers have questioned whether P2P CBIR systems are scalable at all! In a recent paper [21], the authors argued that the lack of efficient high-dimensional indexing techniques prevent the adaptation of P2P CBIR. With the arrival of the NV-tree, however, a solution may have become available.

Since the NV-Network is a centralized system, we tried to find systems more relevant than P2P applications. We found only one paper which studied the application of CBIR using a cluster of computers in a centralized environment [5]. There, the authors proposed a scatter and gather architecture for the sequential scan. The setup contained a single master machine and 25 worker machines, where each worker machine contained approximately 1.18 million global descriptors (or almost 30 million descriptors in total). When a query was performed, the master machine computed the query descriptor and broadcast it to all the worker machines. Each worker machine then searched its local collection using a sequential scan. This architecture had several flaws: first it did not scale well since all machines used a sequential scan; second, throughput decreased when many searches ran concurrently; and third, the hardware cost was very high, since very many machines had to be bought.

Chapter 3

The NV-tree

The NV-tree[1] (or Nearest Vector Tree) is a multi-dimensional indexing structure which provides efficient and effective support for local descriptor retrieval. The NV-tree returns approximate k-nearest neighbors of high quality using only a single I/O operation. This is done by partitioning and projecting the data space into sub-partitions recursively until each sub-partition can be fetched in a single I/O operation.

Section 3.1 details the NV-tree structure and its creation process. Section 3.2 describes how the tree is traversed to find the correct leaf-partition. Section 3.3 discusses how results from one or more NV-trees can be aggregated. Section 3.4 describes stop rules for early termination of query processing. Finally, Section 3.5 gives a summary.

3.1 NV-tree Creation

Similar to the PvS-index, the NV-tree is created using a repeated combination of projecting and partitioning of the high-dimensional descriptors. Initially, all descriptors of the collection are considered to be part of a single temporary partition. Then, the descriptors are projected onto a single line (chosen by picking from a pool of random lines the one that best distributes a sample of descriptors from the temporary partition). The high-dimensional descriptors are then partitioned, based on their projection to this line, into a set of new temporary sub-partitions.

This process of projecting and partitioning is repeated for all the new temporary partitions. The process stops when the number of descriptors in a temporary partition is smaller than an administrator-defined limit, designed to be disk I/O friendly. When this limit is reached, the descriptors in the temporary partition are projected again to a new line and the descriptor identifiers are written to a data file on disk according to the rank of their projected value.

Instead of storing each descriptor entry as a value/ID pair, we store only every 16th projected value, thereby fitting almost 100% more descriptor entries inside the partition. The

[1] A patent application is pending.

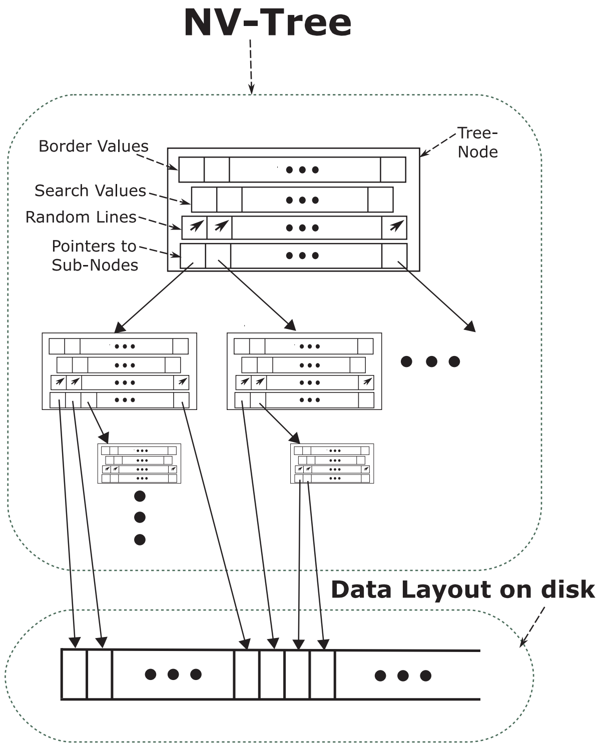

Figure 3.1: An unbalanced NV-tree.

effect of this approximation on the retrieved quality, however, is almost negligible since the final rank of the descriptor entries is already approximated by the last projection.

Two things are worth noting here. First, repeatedly projecting and partitioning descriptors tends to keep in a single partition the descriptors that are close in the feature space, thereby producing high quality results at query time, even at a very large scale. To further improve result quality, however, we introduce redundancy by allowing the partitions to overlap, thus making sure that near neighbors are likely to be together in at least one partition. Second, temporary partitions can either contain a fixed or a variable number of descriptors, resulting in a balanced or unbalanced NV-tree. The structure of the balanced tree must be determined before the index creation, while the unbalanced trees automatically tune projections and partitioning policies to better capture the real distribution of descriptors in space and yield results of higher quality.

Figure 3.1 illustrates the data structure of an unbalanced NV-tree. Each node in the NV-tree manages four arrays: the first array stores the splitting points of each sub-partition. The border values are necessary when updating the tree. The second array stores the 25% quantile values of each sub-partition. These quantile values are search values, which guide the index creation and search down the correct path in the tree. The third array stores the lines used for projecting each sub-partition. Finally, the fourth array stores references to all its sub-partitions, which can reference a sub-partition in the NV-tree or a sub-partition in a data file. Note, that sub-partitions inside the NV-tree do not contain any references to descriptors, making the tree structure small enough to easily fit it in memory. The sub-partitions stored in the data file, however, may be large depending on application and disk characteristics.

3.2 NV-tree Search

During the processing of a single query descriptor, the NV-tree is traversed (in memory) as follows. At each level of the tree, the query descriptor is projected to the line associated with the current node. Using the search values for that level, the search is then directed to the appropriate sub-level. This process of projection and choosing the right sub-level is repeated until the search reaches a leaf partition. This single leaf partition is fetched into memory and the search returns the k descriptors closest to the final projection of the query point.

When the k approximate nearest neighbors have been found for each query descriptor, each of these neighbors gives a vote to the image it belongs to. The images are then ranked by how many votes they received during this process. The image which gets the most votes is considered to be the most similar to the query image, the image which received the second most votes is considered to be the next most similar image, and so on.

3.3 Aggregating Results

For some applications, more accurate results can be obtained if more than one NV-tree is used. In this case, results returned by individual NV-trees must be merged using some rank aggregation algorithm.

Many rank aggregation methods exists, but they all share the same goal, which is to combine several ranks into one final ranking which represents an optimal ordering of the ranks. The rank aggregation method that gives the best results is Kendall optimal aggregation. Unfortunately, computing a Kendall optimal aggregation of just four ranks is NP-complete [8]. An approximation to the Kendall optimal ordering is the Footrule optimal aggregation, which can be solved in polynomial time. A recent median rank algorithm published by Fagin et al. in [9] called OMEDRANK, approximates the Footrule optimal aggregation and is database friendly and efficient to compute.

In this thesis, we chose to use an approximation of the OMEDRANK algorithm for calculating the final ranking. The final ranking is computed by traversing each NV-tree until the relevant leaf-partitions have been retrieved. Each of those partitions are then used as input to the approximate OMEDRANK algorithm, which works in the following way:

For each leaf-partition we project the query descriptor to the line associated with the partition to find a starting position. When this has been done for all partitions, we start two cursors for each partition at its starting position, respectively reading successively lower and higher values. The cursors are used in a round-robin fashion, to simultaneously traverse all partitions and retrieve descriptor identifiers. The algorithm keeps track of how often each descriptor is encountered while the cursors are moved. When a particular descriptor has been seen in more than half of the partitions, it is returned as the nearest neighbor. Processing then continues, until the k nearest neighbors have been returned.

To increase the cache hit rate, both cursors are moved 16 positions at a time instead of one, therefore giving 32 descriptor votes at each iteration. Note though, that this optimization, coupled with the approximate storage structure results in an aggregation which is an approximation to the OMEDRANK algorithm.

3.4 Stopping Rules

We have observed that images that match a query image well, will typically receive tens or hundreds of votes. Two unrelated images, on the other hand, may have a few random descriptors in common among the hundreds of descriptors created, but these random votes are generally few and far between. Since the query processing time with the NV-tree is constant, it is solely dependent on the number of query descriptors processed. Therefore, we consider stopping as soon as the search is either confident that a match has been found or that no match is likely to be found with further processing.

Assuming that there is no matching image in the collection, any descriptor is equally likely to be returned as a neighbor to a specific query descriptor. Thus, the probability of any descriptor from a particular image matching the query descriptor is $p = k/n$, where k is the number of nearest neighbors returned and n is the number of images in the collection. The likelihood that this particular image gets more than x votes after evaluating m query descriptors then follows the binomial distribution $Bin_{m,p}(x)$. As a result, the probability that *any* image gets more than x votes by coincidence is given with the following formula:

$$P(max(X_1..X_n) > x) = (1 - Bin_{m,p}(x))^n \tag{3.1}$$

By choosing a fixed probability value P for the likelihood $P(max(X_1..X_n) > x)$ we can, for a given m, calculate x_m, which is the number of votes that an image must have such that the odds of the image being a random image are equal to P. When considering intermediate (or final) results, this method is used to determine which images are highly likely to be matches, which images are highly unlikely to be matches, and which images fall somewhere between the two.

Consider Figure 3.2, which demonstrates the processing of a fictional query image with 1,000 query descriptors against the collection used in most of our experiments. The figure shows two lines that divide the figure into three distinct parts. The upper line corresponds to a likelihood of $P = 1/1,000,000,000$ that an image is a random image; any image scoring above the line will be considered a match to the query image. The lower line corresponds to a likelihood of $P = 1/20$; any image scoring below this line will be considered not to be a match. Images scoring between the two lines are called “undecided”.

To give an example of this classification, an image that has received its tenth vote after processing 100 query descriptors will be considered a match. If, however, the tenth vote is

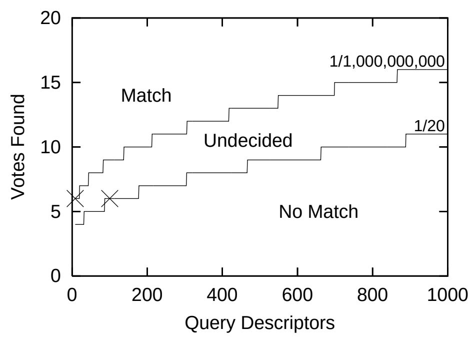

Figure 3.2: Visualization of the stopping rules.

seen after processing 500 descriptors, it is considered undecided. Finally, if the tenth vote is seen only after processing 900 descriptors, the image is considered not a match.

Furthermore, Figure 3.2 has two "×" marks that are notable as they signal the application of two distinct stopping rules.

Stop Rule 1 (Matches Found): The first mark, on the upper line, indicates that this stop rule starts considering intermediate results when 8 query descriptors have been processed. After that, whenever a situation arises where (at least) one image is considered a match, and no other images are undecided (i.e., all other images are decidedly non-matches), query processing is terminated and the results returned.

Stop Rule 2 (No Matches Found): The second mark, on the lower line, indicates that this stop rule starts considering the possibility of not finding any matches when 100 query descriptors have been processed. After that, whenever a situation arises where all images are considered non-matches, query processing is terminated and the results returned, indicating a failure to find a match.

As long as any images are labeled undecided, further processing is required to determine whether they are a match or not. Note that if processing continues until all query descriptors have been considered, the images that pass the upper line are considered to match the query image; all other images are considered non-matches. In fact, we use this method to determine the result quality of the different descriptor schemes in our experiments.

While applying the stop rules may obviously improve performance, it may also potentially affect the result quality. On one hand, Stop Rule 1 may generate false positives. This possibility arises when a random image generates, by chance, a number of its votes early in the search. As the preceding discussion points out, however, we take two precautions in order to avoid this possibility. First, we require a very low threshold of $P = 1/1,000,000,000$, which means that matching images are extremely likely to be at least partially similar to the query image. Second, we take care not to apply this rule until at least eight query descriptors have been processed, as before that time one or two random votes might trigger a false positive.

Stop Rule 2, on the other hand, may generate misses, when for some reason most of the votes are accumulated late in the query processing. As the description of Stop Rule 2 points out, however, we also take two precautions in order to avoid misses. First, we require only a threshold of $P = 1/20$ to make images undecided, which means that only images that have very low scores are ruled out entirely. Second, we take care to apply this rule conservatively, typically once 100 query descriptors have been processed.

3.5 Summary

In this chapter we have presented the NV-tree structure, described its creation process and showed how it is traversed. When a leaf-partition has been retrieved, the k nearest neighbors are the ones located closest to the final projection of the query point. When aggregating results from more than one NV-tree, some rank aggregation methods must be used to merge the sub-partitions retrieved from each NV-tree. In this thesis, we chose to use an approximation of the OMEDRANK algorithm.

Since the query processing time with the NV-tree is constant, it is solely dependent on the number of query descriptors processed. Therefore, we presented two stop rules for early termination of the query evaluation. The two stop rules are used to determine when a) a matching image is most likely found, or b) no matching image is likely to be found at all. While neither of these conditions is determined, processing continues. The stop rules are first and foremost a performance issue, trading improved query performance for potentially weaker results. Both rules, however, have two parameters (the probability threshold and the starting point) that can be used to make them stronger or weaker as applications require.

Chapter 4

The NV-Network

In Section 4.1 we list our design goals for the NV-Network. Section 4.2 describes the architecture of the NV-Network in detail. A summary is given in Section 4.3.

4.1 Design Goals

Our overall goal for the NV-Network was to build a system that combined the benefits of the PvS-framework with the ones of distributed systems. During the design phase, we made a list of requirements we wanted the system to have. In the following, we discuss these goals briefly:

- **Scalability**: It should be possible to add hundreds of worker machines to the NV-Network while getting at least a linear performance increase in terms of throughput with respect to the number of worker machines.
- **Effectiveness**: The search effectiveness of the NV-Network should be no less than that of the NV-tree.
- **Reliability**: The NV-Network should detect automatically if a worker machine fails and take appropriate actions. Additionally, the search quality should be unaffected by hardware failures as long as at least the coordinator and one worker machine are working normally.
- **Ease of use**: As with most distributed networks, ease of use and maintenance is of high importance. The NV-Network should be easy to use and maintain and have low setup overhead.
- **Cost**: The NV-Network should make very moderate hardware requirement while taking advantage of fast hardware.

The remainder of this section describes an architecture which fulfills all these design goals.

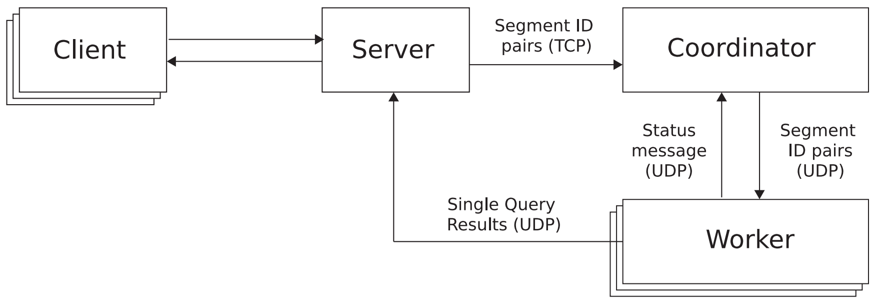

Figure 4.1: A component layout of the NV-Network.

4.2 Architecture

In the introduction we mentioned that the NV-tree is I/O bound. Therefore, by replicating the NV-tree indices to one or more disks, we increase throughput accordingly. Since a single machine is only able to handle a limited number of disks, a better approach is to replicate the NV-tree indices to independent machines. This is the basic idea behind the NV-Network.

Many iterations on the final architecture were made and numerous prototypes were developed. In the end, we came up with an architecture that we were satisfied with and met our design goals. This architecture is shown in Figure 4.1. It consists of four major components. The clients issue the search, the server manages the search status, the coordinator manages the workers, and finally the workers perform the individual NV-tree searches.

Following is an overview of what happens when a new search is issued:

1. The client issues a search by computing the local descriptors from the query image and sending them to the server.
2. The server computes the pairs of segment identifiers (IDs) which need to be accessed for each query descriptor. They are then sent to the coordinator.
3. The coordinator assigns each pair of segment IDs to the best worker.
4. The workers perform the NV-tree search for each set of segment ID pair they receive and send the results back to the server.
5. The server aggregates the results and sends them back to the client.
6. The client displays or forwards the results.

This architecture has some desirable benefits. First, to enhance performance, worker machines may be added accordingly. Second, the workers can be easily managed and administrated by making them send alive messages periodically to the coordinator. Third, by implementing a smart load balancing unit inside the coordinator, we are able to take advantage of fast and expensive worker machines while allowing cheap worker machines to coexist. Fourth, we do not need to make any changes to the NV-tree, as we simply contain it inside the worker machines, also making the other components independent of the of the underlying indexing technique.

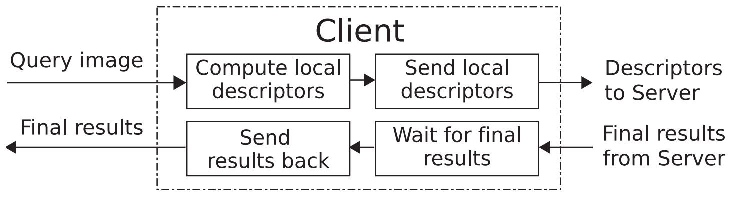

Figure 4.2: A detailed look inside the client Component.

The network communications between these components use either the TCP or the UDP protocol. From Figure 4.1 we see that all the communications to and from the workers use the UDP protocol, since managing a TCP connection to each worker would bring considerable overhead to the coordinator. Although UDP does not provide the reliability or the ordering of messages that TCP does, as datagrams may arrive out of order or go missing without notice, they are faster and more efficient for many lightweight messages. Additionally, since the server can resend lost packages, because it knows when a packet has been lost, we decided to use UDP for the majority of our network communications. The communications between the client, server and coordinator, however, all use the TCP protocol. These messages occur much less frequently than the worker UDP messages and contribute only a small fraction to the overall network traffic.

The remainder of this Section is organized as follows. In Section 4.2.1 we start by describing the client, followed by the server in Section 4.2.2. The coordinator is explained in much detail in Section 4.2.3. Finally, we describe the worker in Section 4.2.4.

4.2.1 Client

The client component computes the local descriptors from the query image and sends them to the server using TCP. The client then waits for a response from the server containing the final results.

We have implemented the client component both as a C++ program and as a HTML web service, with an interface very similar to a Google image search. Figure 4.2 shows the C++ version of our client.

We use the client C++ program for all our experiments, since in that version we can select many query images at startup and let the client search them one by one automatically.

4.2.2 Server

Since we expect many clients to be searching at the same time, we need to manage the search status for each client. This is done by the server (see Figure 4.3).

The server can be configured in two ways; depending on how it aggregates the intermediate results from the workers. Either it uses the stop rules discussed in Section 3.4, and stops the search when the stop rule criteria has been fulfilled, or it waits until it has

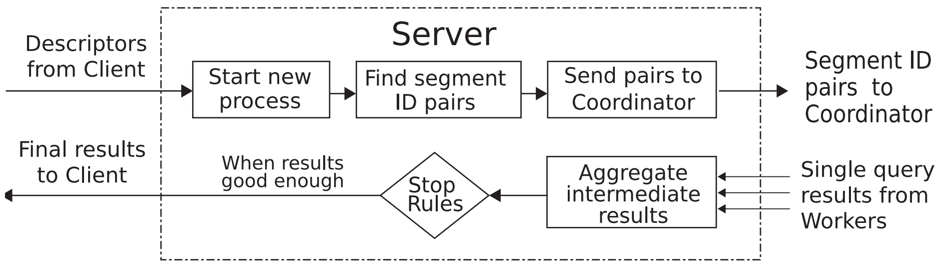

Figure 4.3: A detailed look inside the server Component.

received results for every query descriptor (Figure 4.3 shows the server using the stop rule configuration). If the server decides to stop the search, it sends a stop signal to the coordinator that invalidates the search.

When a client issues a new search, the server receives the query descriptors from the client. The server then selects a unique search ID and a unique port number that will be assigned to this search. The search ID is assigned incrementally by the system, while the operating system is asked for an unused port number. The server then forks a new process and sends it these two numbers as parameters.

Recall, that we have chosen to use two indices, since that results in optimal performance with respect to effectiveness. Therefore, to evaluate a single query using the NV-tree, two segments need to be loaded from disk. We call these two segments extracted for each query descriptor a *segment ID pair*.

After the server starts up a new search process, the process scans the query descriptors and looks up the set of segment ID pairs for each query descriptor. This is done efficiently and in-memory (only a few megabytes are required) by traversing the upper levels of the NV-tree.

After extracting the segment ID pairs for all query descriptors, they are packed into a message, along with the search ID and listening port number and sent to the coordinator. The search ID and port number are attached to all further network messages in the network for this search. This is necessary for two reasons:

- Each search thread at the server machine has a UDP listening port, where it waits for the results from the workers. When a worker has processed a single descriptor query, it sends the results back to the server, to the port specified in the message containing the original descriptor query. This listening port is therefore a unique number specifying which server thread should get the result message.
- The search ID is important when using stop rules. Consider this example: The server starts a new search thread and sends the query descriptors to the coordinator. Since the server is configured using stop rules, it decides to stop the search early. There might, however, be many workers still processing query descriptors for this canceled search. Now, at the same time, a new search thread is started at the server and that thread is assigned the same listening port as the canceled search. When the workers have finished processing the descriptors of the canceled search, they send the results back to the server to the port number specified. By using search IDs, the

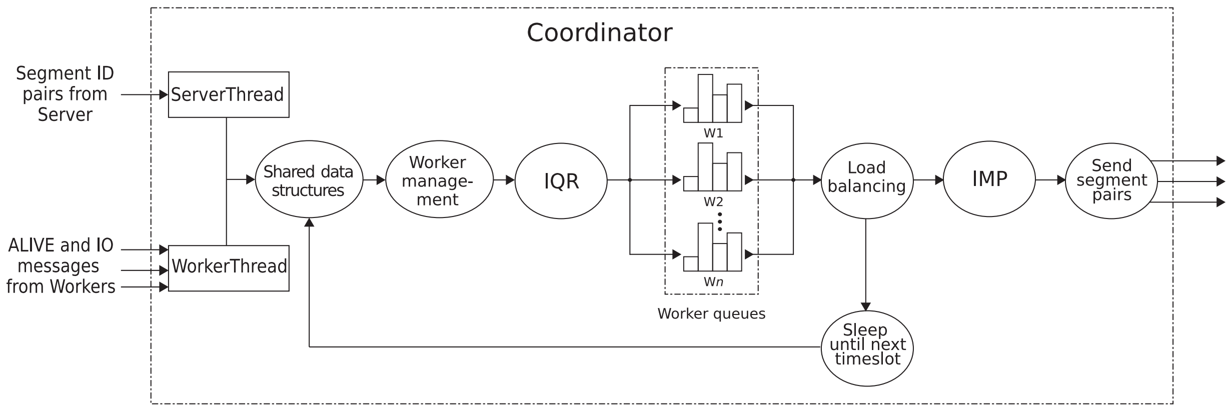

Figure 4.4: The Coordinator Component.

result messages can be checked for mismatching IDs, discarding the ones that do not belong to the active search.

When the Search is finished, the results are sent back to the client which issued the search.

4.2.3 Coordinator

The coordinator (see Figure 4.4) is the central piece in the NV-Network architecture. It handles the load balancing, scheduling, search assignment among the workers, and the dynamic insertion and deletion of new workers.

From Figure 4.4 we see the logical parts inside the coordinator component. The coordinator works similarly to a heart beat. The frequency of the heart beat is determined by a fixed interval we call a time slot. During each time slot, the coordinator performs its duties and sleeps until the start of the next time slot. To see visually what its duties are during each time slot, notice the loop inside the coordinator. This loop is traversed and all the parts visited are executed.

An overview of the operations of these parts is now explained briefly with an example. In this example, assume that one client has issued a search, the server has sent the segment ID pairs to the coordinator, and that the serverThread has received the segment ID pairs and added them to a shared structure. Also assume there are three workers in the system.

During each time slot:

1. The coordinator (main thread) checks the shared structure to see whether any memory changes have occurred at some worker. In this example, no memory changes have occurred.

2. The coordinator now checks the shared structures to see whether any new searches have been added. In this case one new search is found.
3. Next, the coordinator checks whether some worker has crashed. Each worker sends ALIVE messages periodically, and when some worker has not sent an ALIVE message in a certain period, it is removed from the system. In this example, we assume all workers are working normally.
4. Now, all the segment ID pairs are assigned a best worker, using a technique we call Intelligent Query Routing (IQR), which is described below. The coordinator maintains four queues of segment ID pairs for each worker in the system. When the IQR finds the best worker for a given segment ID pair, it is assigned to the appropriate queue owned by that worker.
5. When all segment ID pairs have been assigned to the best worker, the system calculates how many segment pairs should be sent to each worker in this time slot.
6. The coordinator now loops through the four queues for each worker and selects the appropriate number of segment ID pairs. The coordinator can assign different priorities to these queues, using a technique we call In Memory Preference (IMP), which is described below. After removing these segment ID pairs, they are sent to the workers.
7. The coordinator sleeps until the next time slot.

The above example demonstrates the flow between the different parts in the coordinator. We now explain these parts in more detail.

Listening Threads

The coordinator communicates with the server and all the workers. These communications occur frequently, especially when the system is under heavy load. Therefore, to more efficiently serve these communications, two listening threads are created when the coordinator starts up. These threads are:

- **serverThread**: This thread listens for incoming messages from the server. When the server sends the coordinator a new search, this thread will insert all the search information (segment ID pairs, search ID, server IP address and port) to a shared data structure. When the server wants to stop a search (because of the stop rules), its search ID is also sent to this thread which signals the cancellation of the search.
- **workerThread**: This thread listens for incoming messages from the workers. Each worker sends periodically an ALIVE message to this thread. Included in this message is the worker queue size so that the coordinator can more effectively load balance the system.

  Additionally, another type of message is sent when a worker has changed its memory contents. This message is called an IO message, listing all changes of the workers memory contents since the last time an IO message was sent. Each record in this message has three entries, and is in the following form:

{ *loaded/removed, segment ID, index ID* }

Therefore, each record contains the segment ID, the index ID and whether they were loaded from disk or removed from memory. With this information, the coordinator has a fairly up-to-date picture of which segments are in memory at each worker.

Shared Data Structures

The coordinator has several data structures which are shared between the serverThread, workerThread and the main process:

- **hWorker**: A hashtable which stores all necessary information for each worker. This information includes all the priority queues, the queue size at the worker side, how many segment pairs should be sent to the worker in each time slot, and the number of time slots since the last time the coordinator received an ALIVE message.
- **hMem**: A hashtable which stores which segments each worker has in memory. By probing this hashtable with the segment ID, we get a list of workers which have that segment in memory.
- **hSearchStatus**: A hashtable which stores the search status of each search in the system. This structure stores whether a search is running or has been canceled and how many segment ID pairs are located at the coordinator, waiting to be sent to a worker.
- **hPending**: A hashtable which stores which segment IDs are located (pending) at the coordinator, waiting to be sent to a worker.
- **qNewSearches**: A queue of new searches which have been received from the server since the last time slot.

Worker Management

The next part in the coordinator is the worker management, which discovers new workers and detects hardware failures.

Each worker must send an ALIVE message to a specified listening port on the coordinator regularly to inform it that it has not crashed (we found that sending ALIVE messages every 200ms is sufficient). When a new worker boots up and sends its first ALIVE message, the worker management will add it to the network automatically.

When these messages stop arriving, the worker is removed from the system and pending searches on the coordinator will be assigned to other workers (see Intelligent Query Routing for details on how this assignment is done).

Intelligent Query Routing (IQR)

Since the coordinator knows the memory contents at each worker, we can route each search to the worker which has the particular segments in memory. By routing each segment pair intelligently, we minimize the number of I/Os performed, increasing the overall performance of the system.

This routing is performed using a policy called Intelligent Query Routing (IQR). In general, IQR has a very simple job; it takes as input a set of segment ID pairs and assigns each pair to the worker, that is most likely to process the query the fastest, i.e. the best worker for this query.

For each worker, the coordinator maintains four queues of segment ID pairs. After IQR finds the best worker, it assigns the segment ID pair to the appropriate queue. These four queues store the following types of segment ID pairs:

- **The ALL queue**: Both segments are in memory at the worker.
- **The SOME queue**: One segment is in memory at the worker.
- **The PENDING queue**: At least one segment in the other queues shares the same ID.
- **The NORMAL queue**: Segment ID pairs with none of the above are assigned to this queue.

We now give an example of the assignment process (assume we have received a new search containing exactly 600 segment ID pairs):

1. For all the 600 segment pairs, we check whether some worker has *both* segments in memory. In this example, assume we find 50 segment pairs which fulfill this criteria. These 50 segment pairs are added to the ALL queue of their respective workers.
2. For the remaining 550 segment pairs, we check whether some worker has *either* of the segments in memory. In this example, assume we find 150 segment pairs which fulfill this criteria. These 150 segment pairs are added to the SOME queue of their respective workers.
3. The remaining 400 segment pairs, containing 800 segments, are nowhere in memory. However, if some worker has previously been assigned a segment (which has not been sent) which shares an ID with any of those 800 segments, we benefit from assigning this segment pair to that worker, since it only has to load that segment once. Each of the 800 remaining segments are simply probed against the hPending structure, and assigned to those workers which have the segment already pending. Assume there are 200 segment pairs which fulfill this criteria.
4. The remaining 200 segment pairs have no special properties, hence it does not matter to which worker they are assigned. The system assigns these 200 segment pairs across the workers equally.

If the stop rules are used, we scramble both the ALL and SOME queues after the IQR step. By scrambling these queues, we better interleave the active queries so they are not taken in FIFO order. By scrambling the queues, the workers are more likely to be evaluating queries for more than one search at a time, which potentially can increase the system throughput since only a subset of the descriptors need to be evaluated for a given search.

Load Balancing

If the system sends segment pairs faster than the worker can process them, large queues start to occur at the worker and the response time for new segment pairs will increase accordingly. For example, if all workers have 100 segment pairs queued, and each worker processes 10 of them per second, then the system cannot process any new searches for the next 10 seconds!

For this reason, the system keeps all workers moderately loaded. This is the responsibility of the load balancer, which tries to keep the local worker queues as small as possible, without having the workers wait for something to do.

The system load balances the workers by checking, at fixed time slots, how many segment pairs each worker has searched (thereby giving a measure of each worker's throughput) and sending it new segment pairs in proportion to its throughput. This is done by maintaining a *serve_count* number for each worker. If a worker gets too loaded, this number will decrease, but if the worker can handle more load, this number will increase.

The *serve_count* is governed by a smoothing formula, which smooths it over time. We use a smoothing formula because when segments are in memory they add an unrealistic measurement of the actual average throughput of the worker. Therefore, by smoothing the load factor we achieve more realistic performance measurements.

The smoothing formula is now explained. The coordinator records a variable called *avg_processed* for each worker which denotes how many searches the worker processes on average between each ALIVE message. This number is then smoothed with another variable, called *last_processed*, which denotes the number of searches processed since the last ALIVE message, These two variables are then used to update the *serve_count* as follows:

$$serve\_count = \left\lceil \frac{4 * avg\_processed + last\_processed}{5} \right\rceil \tag{4.1}$$

While the smoothing formula gives a good estimate of each worker's throughput, it does not work well with sudden changes in their performance. For example, if some particular worker suddenly starts a new (unrelated) process that hogs the hard disk, its performance will decrease dramatically. While this would be noticed by the load balancer, the smoothing formula would decrease the *serve_count* in smaller steps than what would be optimal. To compensate for these sudden changes, the load balancer calculates how many seconds it is likely to take each worker to empty its queue. This is calculated by multiplying each worker's *serve_count* with its local queue size. If this time is bigger than a second, the *serve_count* is decreased by 60%, and by 80% if it reaches two seconds.

To compensate for the dynamic performance difference between the workers during query evaluation, we add one final step to the load balancing which prevents workers from starving. This is done by 1) identifying the set of (starving) workers that have exhausted their queues, 2) ranking the other workers by their NORMAL queue size and 3) repeatedly reassigning segment pairs from the largest NORMAL queue until each starved worker in the set has been assigned *serve_count* segment pairs or until all NORMAL queues are empty. In other words, if a worker has been sent all its segment pairs, the load balancer steals unsent segment pairs from other workers and reassigns them. In this way, we fully utilize all workers at all times!

In Memory Preference (IMP)

While the load balancer decides how many segments should be sent to each worker at every time slot, it does not decide *which* segment pairs are sent. After the load balancer updates the *serve_count*, it is the job of the IMP to decide how many segment pairs from each of the four queues should be sent to the workers.

The intuition is that segments stored in the ALL or SOME queue should be prioritized over the ones in the other queues, since they can be searched considerably faster. IMP implements this by setting a lower limit (called IMP limit) on how many segment pairs should be removed from the ALL and SOME queue. It first empties the ALL queue and then the SOME queue until the IMP limit is reached. The remaining segment pairs that should be sent are removed from the four queues in a round robin fashion in the following order: ALL, SOME, PENDING and NORMAL.

Consider the following example where the IMP limit is 4, the *serve_count* is 10, and there are 3 segment pairs in the ALL queue and 10 in the SOME queue. In this case, because of the IMP limit, the ALL queue would be emptied and 1 segment pair would be removed from the SOME queue. The remaining 6 segment pairs would be taken in a round robin order as follows: 1 from SOME, 1 from PENDING, 1 from NORMAL, 1 from SOME, 1 from PENDING and 1 from NORMAL. Therefore, in this example, this particular worker would be sent 3 segment pairs from the ALL queue, 3 from the SOME queue and 2 from each of the PENDING and NORMAL queues.

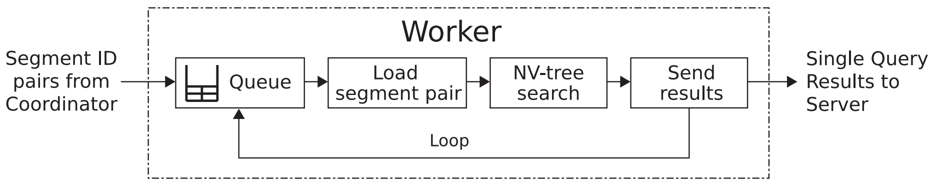
Worker
Segment ID pairs from Coordinator
Queue
Load segment pair
NV-tree search
Send results
Single Query Results to Server
Loop

Figure 4.5: A detailed look inside the Worker Component.

4.2.4 Worker

The last component in the NV-Network is the Worker (see Figure 4.5), which performs the actual search using the NV-tree. For keeping the NV-Network reliable during and after hardware failures, we have kept the workers as autonomous as possible. In our design they need not manage any information or state and are completely unaware of other workers.

When a worker receives a segment pair from the coordinator it adds it to a queue of segment pairs it stores locally (this queue is needed, so that the socket receive buffer on the worker does not fill up in case the coordinator sends segment pairs faster than the worker can process them). This queue is called the local queue of the worker.

When a worker searches, it removes the first segment pair from its local queue, loads each segment (possibly from disk), performs the search and sends the results to the server. If some segments were loaded from disk or removed from memory during the search, an IO message is sent to the coordinator detailing which segments were loaded or removed.

To minimize the network load, the worker groups these IO messages into a single UDP packet. The IO message is sent when either one second has passed since the last IO message, or when the size of the message is 1500 bytes, which is the maximum UDP datatagram size. The worker also sends ALIVE messages to the coordinator at regular intervals.

4.3 Summary

The NV-tree is an efficient indexing structure that responds very quickly to a query. Unfortunately, it is not efficient enough for high-throughput applications. We address this problem in the NV-Network by mirroring the NV-tree across many worker machines, splitting the query descriptors intelligently among them and sending each worker machine the query descriptors in a “good” order.

Since most distributed CBIR systems use P2P networking, they need (for efficiency reasons) to categorize their image collection, making the search quality heavily dependent on the quality and accuracy of the categorization. The search effectiveness of the NV-Network, on the other hand, is guaranteed to be no less than that of the NV-tree.

The NV-Network detects when a worker machine fails and takes the appropriate actions. Since all worker machines are equal (no machine is responsible for some specific crucial function) and the index collection is stored redundantly across all worker machines, we guarantee that the search quality will be unaffected by hardware failures, as long as the coordinator and at least one worker machine are working normally. Compare this to P2P systems or the system described in [5] where crashes severely affect search quality.

As with most distributed networks, ease of use and maintenance is of high importance. The NV-Network has a low setup overhead and requires almost no maintenance. When a decision has been made to buy more worker machines, it is only necessary to install the NV-Network software, connect the physical network link to the system and boot the machine up. If a worker machine needs to be removed from the system, it can simply be unplugged and removed.

The NV-Network has very moderate hardware requirements. Worker machines with just a few hundred MHz CPU are fast enough, since the NV-tree search is only I/O bound. Although the NV-Network makes no minimum requirements on the size of the main memory, is takes advantage of worker machines with large memory by sending more descriptors to those machines. The system requires, however, that the indices fit on the hard disk(s), likely making disk space the major price component in the system.

If many workers are connected to the system, then more information needs to be managed at the coordinator. Although no minimum hardware requirement is made to the coordinator machine, it is beneficial if it contains enough memory to manage the state of the workers. This makes the requirement directly related to how many worker machines are connected to the system and how much main memory they have. No special hardware requirement is made to the server machine.

In the following section, we turn to our performance analysis, which demonstrates the scalability and throughput of the NV-Network.

Chapter 5

Performance Analysis

In this section, we present results from our detailed performance study of the NV-Network. We first describe our experimental setup, including the image collections and hardware used for the measurements. Then we present several experiments, demonstrating various aspects of the NV-Network.

In Section 5.2 we present experiments showing the overall scalability of the NV-Network using two image collections, with and without the stop rules. In Section 5.3 we present experiments showing how the system scales when changing the number of workers and clients, respectively. In Section 5.4 we present an experiment showing how the system behaves if the main memory of the workers is large in comparison to the index size. In Section 5.5 we present an experiment with queries that have different ratios of matches in the database.

5.1 Experimental Setup

All the workers were run on up to ten Dell Optiplex GX240 machines, each equipped with a Pentium 4 1.6GHz processors, 256MB RAM and a single 250GB disk. The server, coordinator and all client processes were run together on a single Dell Optiplex GX240 machine, equipped with a Pentium 4 1.6GHz processor, 1GB RAM and a single 200GB disk. All eleven machines were connected through a 100Mbps ethernet switch.

We used two image collections; the SMALL collection, and the LARGE collection. For both image collections we used the *Eff*^{2} descriptor scheme for creating the descriptors. Table 5.1 shows the details of these collections. The SMALL collection consists of 29,277 high quality photos. We created 20,445,898 descriptors from this collection and created two indices in a balanced [11, 11, 11, 9] configuration, resulting in an total index size of 3,1GB. The LARGE collection consists of 287,269 high quality press photos from Iceland's largest newspaper, Morgunblaðið. We created 169,159,548 descriptors from this collection and created two indices in a balanced [11, 11, 11, 11, 11] configuration resulting in a total index size of nearly 41GB.

Collection	Nr. Images	Nr. Descriptors	NV-tree setup	Index Size
SMALL	29,277	20,445,898	[11, 11, 11, 9]	2 x 1,5GB
LARGE	287,269	169,159,548	[11, 11, 11, 11, 11]	2 x 20,5GB

Table 5.1: Image Collections.

During each experiment, we record several metrics. The primary metrics used in our study are:

- **Searches per second**: How many client searches the NV-Network can process in one second.
- **Descriptors searched**: The total number of descriptors the workers processed.
- **CPU usage**: The ratio between idle time and working time at the coordinator.
- **Hit ratio**: The ratio between total number of segments in memory and total number of segments processed.[1]

We fixed the number of query images per worker to 36 so all setups took similar time to complete. When stop rules were enabled, we increased the number of query images per worker to 108. The number of query images, therefore, varied depending on the setup. Since all workers had rather small main memory, their buffers filled up before their first query image was finished. We therefore did not run any warm up queries to initialize the workers, since that would have had little or no effect on the metrics. Four clients were used concurrently, querying random images taken from the other collection, such that queries have no match in the collection.

To measure the effect IQR and IMP had on performance we usually use three setups (unless otherwise noted) in our experiments:

- **Normal**: Both IQR and IMP are disabled.
- **IQR**: Only IQR is enabled.
- **IQR/IMP**: Both IQR and IMP are enabled.

Note that we do not enable just IMP as that will have almost no effect, since then it is unlikely that many segments are in the ALL or SOME queues.

5.2 Experiment 1: Overall Scalability

In this experiment we demonstrate how the system behaves when measuring the system with 1, 5, and 10 workers, respectively using the SMALL and LARGE collections with and without the stop rules.

[1] Our measure of hit ratio is slightly inaccurate, as it is maintained by information supplied by our own buffer manager, rather than the buffer manager of the operating system. This has a slight effect on results with relatively small collections (Section 5.4), but does not affect results with large collections.

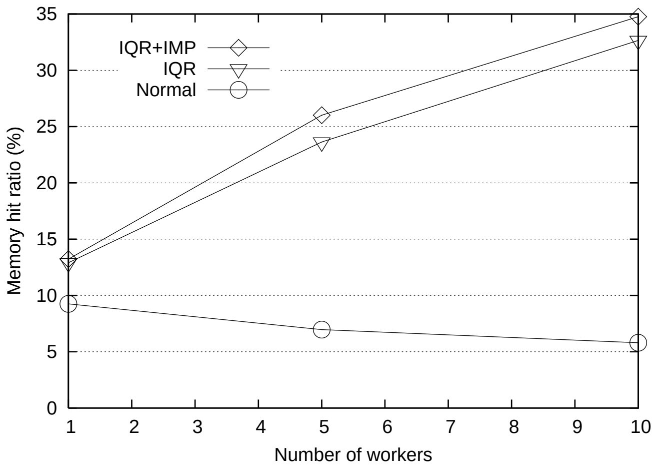

Figure 5.1: (Exp.1 SMALL) Ratio of segments in memory as the number of workers is increased.

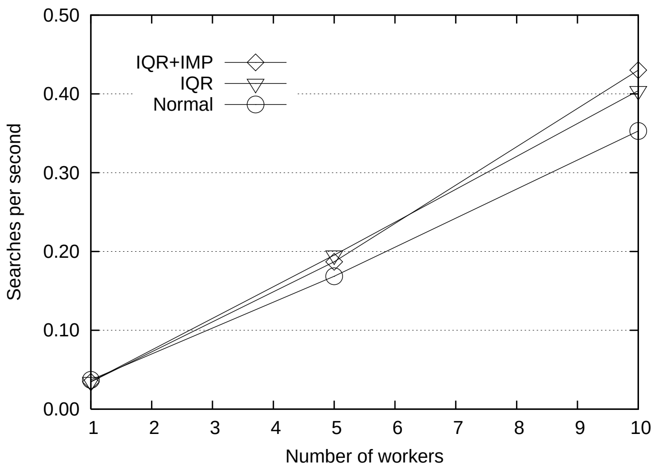

Figure 5.2: (Exp.1 SMALL) Searches per second as the number of workers is increased.

5.2.1 SMALL Collection

Figure 5.1 shows the memory hit ratio at the workers. When looking at the graph we see that the memory hit ratio varies between 6% and 35%. Overall we can note the following. When IQR is disabled, the hit ratio decreases when adding more workers. Enabling IQR, on the other hand, dramatically increases the hit ratio, especially when adding more workers. Finally, enabling both IQR and IMP gives the best results in all cases, though not by a large margin.

We now take a look at the system's throughput during the workload by studying how many searchers the system can evaluate per second. This is shown in the Figure 5.2.

Looking at Figure 5.2 we see that all setups achieve at least a linear increase in performance with respect to the number of workers. When IQR is disabled the increase is exactly linear, but enabling IQR results in more than linear increase. Also enabling IMP gives the best performance. When both IQR and IMP are enabled, the system can evaluate

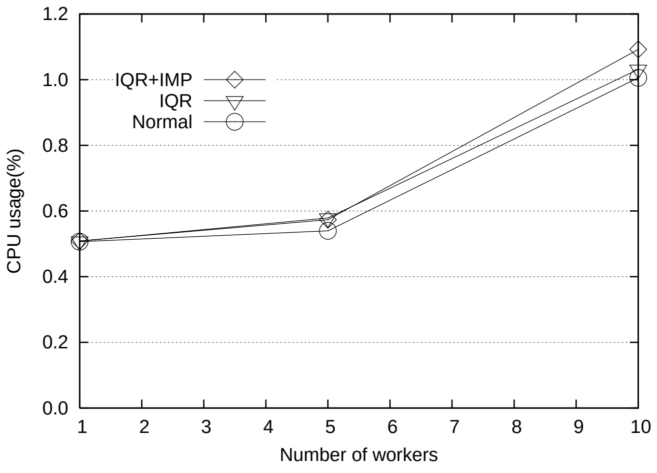

Figure 5.3: (Exp.1 SMALL) Coordinator CPU usage as the number of workers is increased.

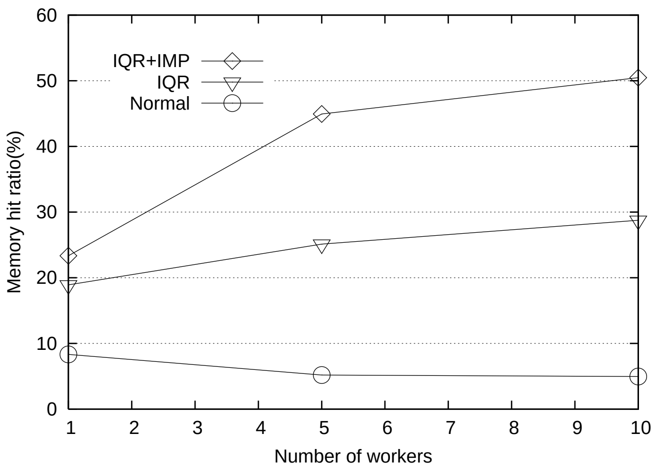

Figure 5.4: (Exp.1 SMALL+Stop) Ratio of segments in memory as the number of workers is increased.

2.1 searches per minute when using 1 worker, and about 26 searches per minute using 10 workers, resulting in about 12.3 performance increase.

Figure 5.3 shows the CPU usage at the coordinator while the experiment was running. From the figure we see that the CPU usage is between 0.5% and 1.1%, leaving much room for extra CPU work.

5.2.2 SMALL Collection + Stop Rules

In this experiment we use the same setup as in Section 5.2.1, except that we enable stopping rules. Figure 5.4 shows the memory hit ratio at the workers. The figure shows that the hit ratio is between 5% and 50% using the different setups, which is a quite dramatic difference from Figure 5.1. We observe that the hit ratio, when both IQR and IMP are disabled is almost exactly the same as in the previous experiment. The other two setups,

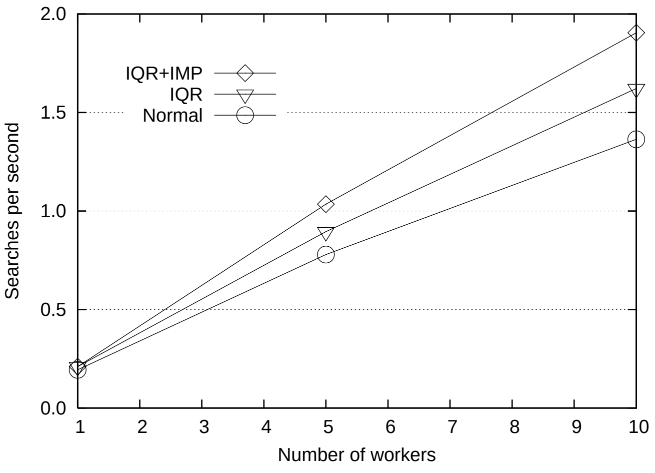

Figure 5.5: (Exp.1 SMALL+Stop) Searches per second as the number of workers is increased.

however, show significant improvements. Furthermore, in the previous experiment, enabling IMP added only a slight improvement to the hit ratio. In this experiment, however, enabling IMP increases the hit ratio by more than 70%.

This increase in hit ratio when enabling IMP is now explained. During every time slot, the coordinator sends segment pairs to the workers. When IMP is enabled it gives priority to segment pairs which are in memory, thus starving others. Since only a subset of the query descriptors need to be evaluated in this experiment because the stop rules are enabled, a much higher ratio of segments which are in memory are sent to the workers.

We now turn to Figure 5.5 which shows the system's throughput. We can see that the throughput has increased dramatically over the last experiment (Figure 5.2). Furthermore, since enabling IMP had such dramatic improvement on the hit ratio, we now see a very clear improvement in the system's throughput when enabling IMP. Here, using the best setup where both IQR and IMP are enabled, the system can evaluate 12 searches per minute with 1 worker, and about 115 searches per minute with 10 workers (instead of 2.1 and 26 searches per minute without the stop rules) . Unlike the last experiment, however, these setups result in linear (or less) improvement in throughput with respect to the number of workers. The reason for this effect is now explained.

Figure 5.6 shows the number of descriptors searched using the three different setups. While we expected that the number of descriptors should grow linearly (as the number of images grow linearly), the figure shows that the number of descriptors searched increases more than linearly. Recall, from Section 3.4, that we stop after evaluating 100 query descriptors because we are using queries that have no matches in the collection. When the server has received 100 results from the workers, it sends a stop signal to the coordinator, which invalidates all outstanding descriptors for that query. The server, however, cannot invalidate descriptors which have already been sent to the workers. Therefore, it is common that some workers evaluate descriptors after the query they belong to has been canceled, resulting in this slight performance decrease. This effect will escalate as more workers are used (Section 5.3.1).

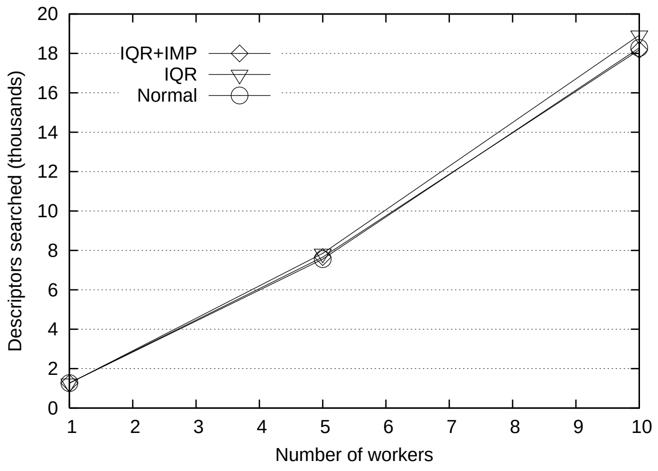

Figure 5.6: (Exp.1 SMALL+Stop) The total number of descriptors searched.

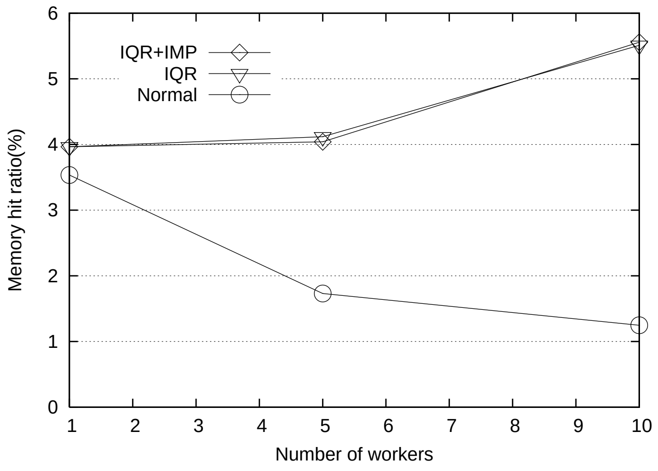

Figure 5.7: (Exp.1 LARGE) Ratio of segments in memory as the number of workers is increased.

5.2.3 LARGE Collection

We now turn to the LARGE collection, otherwise using the exact same setup as in Section 5.2.1. Figure 5.7 shows the memory hit ratio of the workers. From the figure we can see that although the characteristics are similar as in Figure 5.1, we achieve a much lower hit ratio; enabling IQR results in a hit ratio of only about 5%, which is much less than the 35% seen in Figure 5.1. Since the LARGE collection consists of 161,051 segments per index, instead of much lower 11,979 segments per index for the SMALL collection, the likelihood of segments being in memory decreases accordingly, which explains this decrease. In Section 5.3.1 we can see how the hit ratio is affected when scaling from 10 to 200 workers.

As a result of this low hit ratio, we see almost no throughput difference between the three setups, as can be witnessed in Figure 5.8. When comparing Figure 5.8 to Figure 5.2 in Section 5.2.1 the system's throughput has decreased substantially. Although the hit ratio

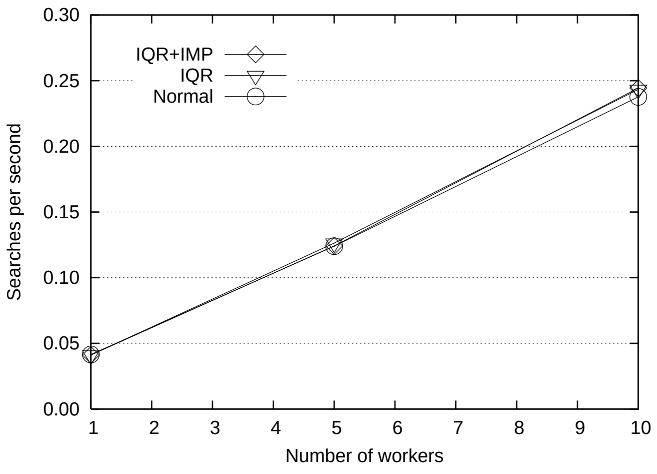

Figure 5.8: (Exp.1 LARGE) Searches per second as the number of workers is increased.

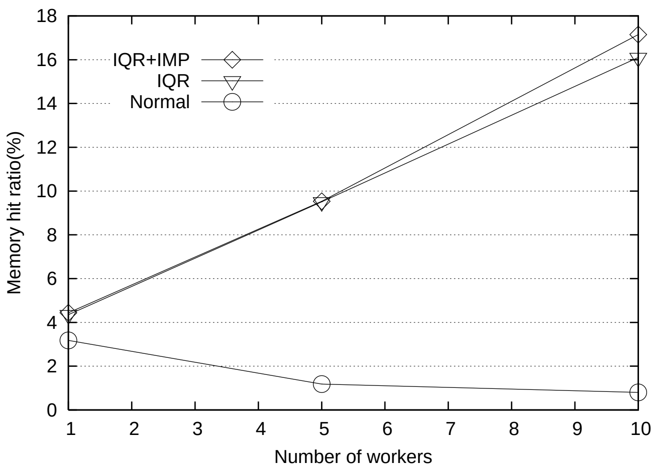

Figure 5.9: (Exp.1 LARGE+Stop) Ratio of segments in memory as the number of workers is increased.

is here somewhat lower, it does not suffice to explain this decrease. We now give a reason the decreased throughput.

One of the main benefits of the NV-tree is its constant-time retrieval, regardless of the collection size. Its performance can, however, be affected by outside factors, such as the hard drive. Since larger collections are likely to be more fragmented than smaller collections, it is likely that the hard drive has to spend more time seeking to the correct track to find the requested partition on disk. This seeking time is what slows the retrieval. Had the index be stored on a large random access device, such as a flash drive or a USB key, the performance difference between these collections would be negligible.

5.2.4 LARGE Collection + Stop Rules

In this experiment we use the same setup as in Section 5.2.3, except that we enable stopping rules. Figure 5.9 shows the memory hit ratio at the workers. From Figure 5.9 we can

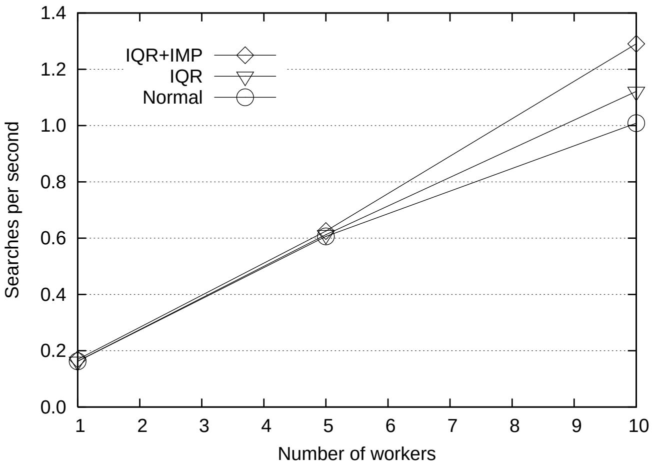

Figure 5.10: (Exp.1 LARGE+Stop) Searches per second as the number of workers is increased.

see that disabling IQR results in the hit ratio going from 3% down to less than 1%. When IQR is enabled, however, it increases linearly from 5% to 16%. Enabling IMP results in a slight improvement. We will see in Section 5.3.1 how the hit ratio is affected when scaling from 10 to 200 workers.

Looking at Figure 5.10, which shows the throughput of the system, we see that the system can evaluate about 60 searches per minute. Enabling IQR increases the throughput by about 12%. When IMP is also enabled, the total improvement is nearly 30% over the basic setup.

5.3 Experiment 2: Scalability of Central Components

5.3.1 Number of Workers

In this experiment we demonstrate how the NV-Network scales with respect to number of workers when using the LARGE collection with and without stop rules. Because we did not have access to hundreds of worker machines, we created a program that effectively behaved as a real worker. This simulated worker has the same buffer size and the same segment replacement policies as the real workers. Instead of reading from disk, however, it simulated the I/O request so many workers could share a computer.

We took the best setup from Section 5.2.3 and Section 5.2.4 and used that in this experiment. We then ran the experiment with with 10, 50, 100, 150 and 200 simulated workers. All the simulated worker processes were distributed equally among the 10 worker machines.

From Figure 5.11 we see that, when not using stop rules, the hit ratio starts at about 5% with 10 workers and rises smoothly to about 40% with 200 workers. Recall that the hit ratio with 10 workers from Section 5.2.3 was also about 5%, so this is likely a good

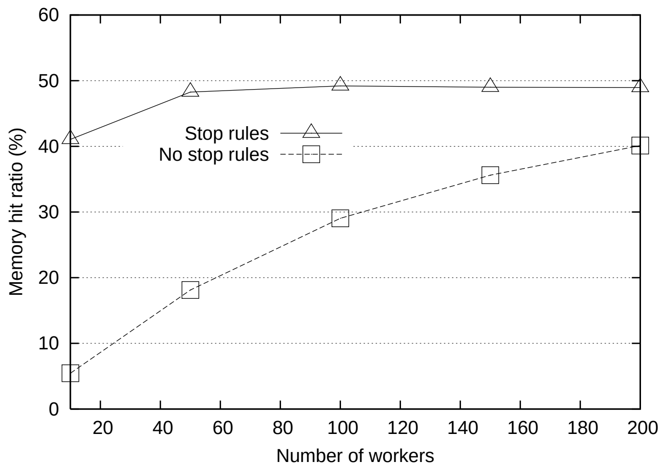

Figure 5.11: (Exp.2 Number of workers) Ratio of segments in memory as the number of workers is increased.

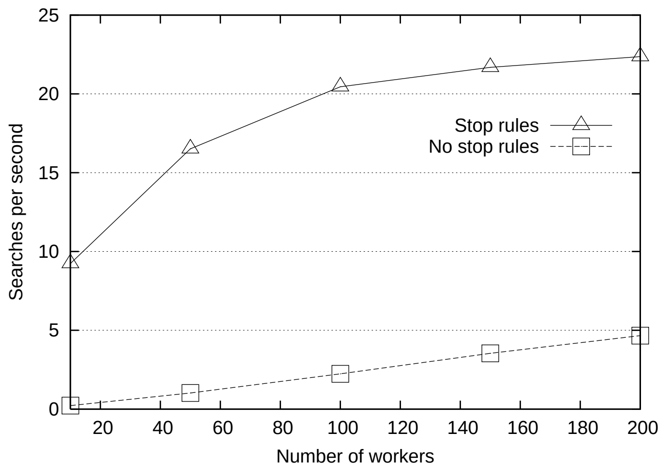

Figure 5.12: (Exp.2 Number of workers) Searches per second as the number of workers is increased.

estimate on how the hit ratio would increase in a real setting. When enabling stop rules, however, we see, that, somewhat surprisingly, we never reach more than 50% hit ratio. This tells us simply, that when some worker has a segment in memory, it rarely has both segments in the segment pair necessary to evaluate the query. This is reasonable, since the probability that $both$ segments are in memory is much lower than the possibility that only one segment is in memory.

Figure 5.12 shows, that when not using stop rules, the NV-Network scales at least linearly as more workers are added. The system reaches the “sweet” spot of one search per second when using 50 workers. The number of searches evaluated per second rises from 0.22 with 10 workers to 2.23 with 100 workers and to 4.66 with 200 workers. When enabling stop rules, we see that with around 100 workers, the system’s throughput appears to have reached a limit as it does not increase much with additional workers. The reason for this can be explained as follows.

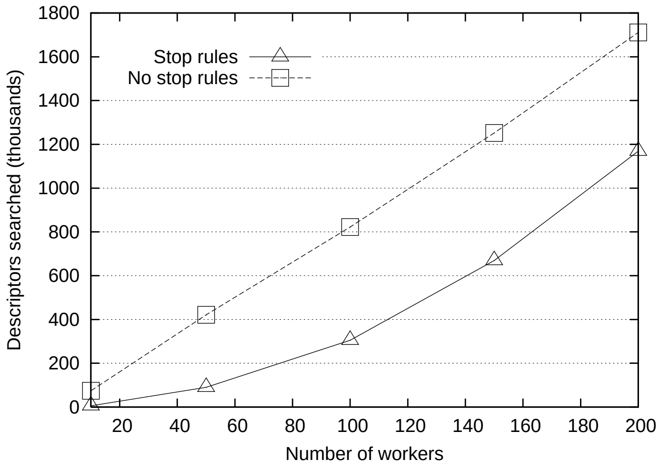

Figure 5.13: (Exp.2 Number of workers) The total number of descriptors searched.

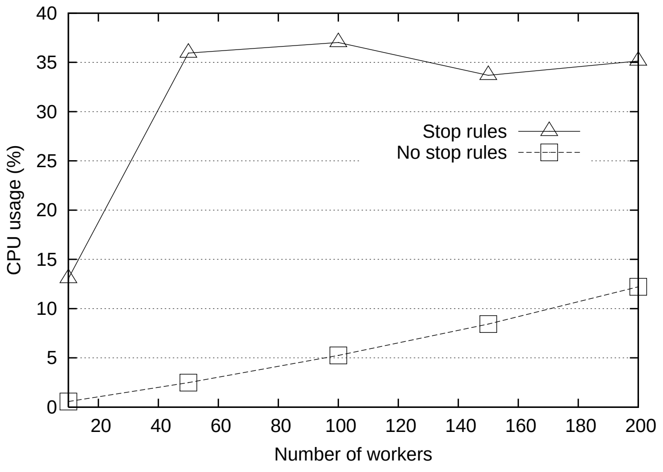

Figure 5.14: (Exp.2 Number of workers) Coordinator CPU usage as the number of workers is increased.

Figure 5.13 shows how fast the number of descriptors searched increases when using stop rules. The reason for why the system's throughput does not scale when using stop rules stems from this rapid increase. The reason for the increase (as explained in Section 5.2.2) is because the server cannot invalidate descriptors which have already been sent to the workers. This simulation, therefore, demonstrates that the NV-Network does not scale as well when using stop rules. We give insights on how this flaw might be fixed in Section 6.

Figure 5.14 shows the CPU usage at the coordinator while the experiment was running. From Figure 5.14, when not using stop rules, we see that the CPU usage with 200 workers is only about 12%. This means that the central components of the NV-Network could scale even further. When the stop rules are enabled, however, the CPU usage rises to almost 40%. Note, that if scalability ever becomes an issue, recall that the clients, server and coordinator are all run on the same machine; so by splitting them to seperate machines we can devote more CPU time to the coordinator. Additionaly, by replacing the

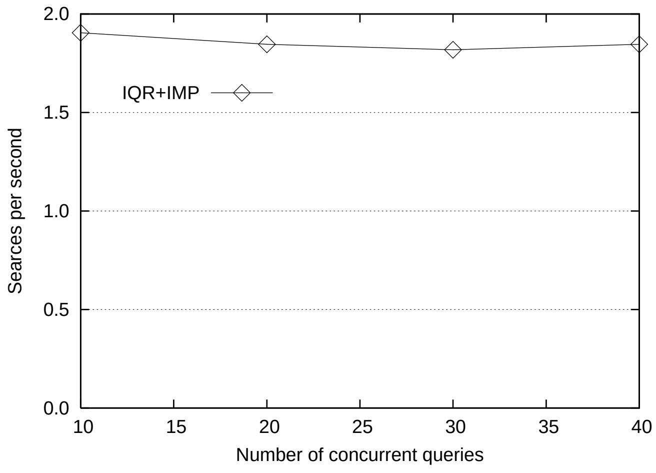

Figure 5.15: (Exp.2 Number of clients) Searches per second while varying the number of concurrent queries.

1.6GHz processor with a more modern one will allow the NV-Network to scale much further.

5.3.2 Number of Clients

One of our main goals when designing the NV-Network was to be able to handle many concurrent queries without affecting throughput. In this experiment we want to test whether this has been achieved. Here, we demonstrate how the NV-Network scales with respect to number of clients when using the SMALL collection with stop rules, enabling both IQR and IMP. We fixed the number of workers to ten and then varied the number of clients. We measured the system's throughput with up to 40 clients, which simulates a very high load.

Figure 5.15 shows that the system's throughput is unaffected by the number of concurrent queries running, and in fact, the throughput is higher than in Figure 5.10 in Section 5.2.4, which used only four clients.

5.4 Experiment 3: Effect of Larger Main Memory

In this experiment we want to study how the system behaves if the main memory of the workers is large in comparison to the index size. Since memory is expensive, we created three rather small image collections, instead of buying more memory.

Table 5.2 shows the details of these collections. Note, that the 30K collection is the whole SMALL collection used in Section 5.2, while the 10K and 20K contain first 10,000 and 20,000 images from the SMALL collection.

We chose, for this experiment, to fix the number of workers at 10 and use the setup where both IQR and IMP were enabled, with and without the stop rules.

Collection	Images	Descriptors	NV-tree setup	Index Size
10K	10,000	6,813,636	[13, 13, 13]	2 x 275MB
20K	20,000	13,989,984	[17, 17, 15]	2 x 500MB
30K	29,277	20,445,898	[11, 11, 11, 9]	2 x 1,5GB

Table 5.2: (Exp.3) Three Small Image Collections.

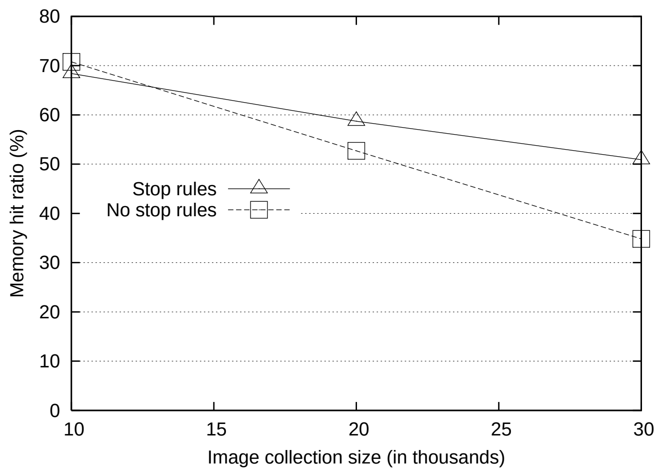

Figure 5.16: (Exp.3) Ratio of segments in memory as the collection size increases.

Figure 5.16 shows that the hit ratio decreases from about 70% with the 10k collection to about 50% where stop rules are enabled and 35% where they are disabled. The 35% hit ratio at the 30k mark is identical to that in Figure 5.1, and similarly, the 50% hit ratio is identical to that in Figure 5.4.

Figure 5.17 shows that the system's throughput is almost unaffected by the collection size when stop rules are not used, reflecting the benefit of the NV-tree as discussed in Section 5.2.3. When the stop rules are enabled, however, we see a significant improvement in the system's throughput when the collection size is decreased.

Although the hit ratio is also higher, it cannot explain the difference seen in Figure 5.17. The throughput using the 10K collection is more than two times more than with the 30k collection. This can be explained by the disk controller (and the operating system) buffering the data as the experiment is running.

Since the total memory of the workers is about 2,5GB, it would seem natural to think that the hit ratio should be close to 100%, especially with the 10K collection which is only 550MB in size. This is, however, not the case since the memory needed to store all the segment pair permutations grows quadratic with the index size.

5.5 Experiment 4: Effect of Matching Queries

So far, we have searched using queries which had no matches in the collection. In this experiment we want to vary the ratio of the queries that have matches in the database

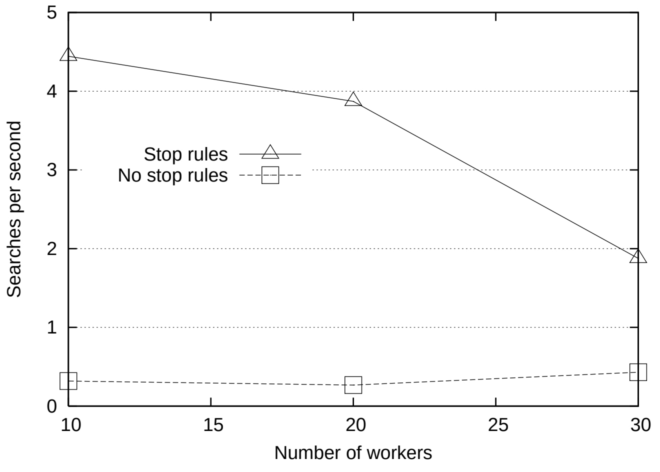

Figure 5.17: (Exp.3) Searches per second as the collection size increases.

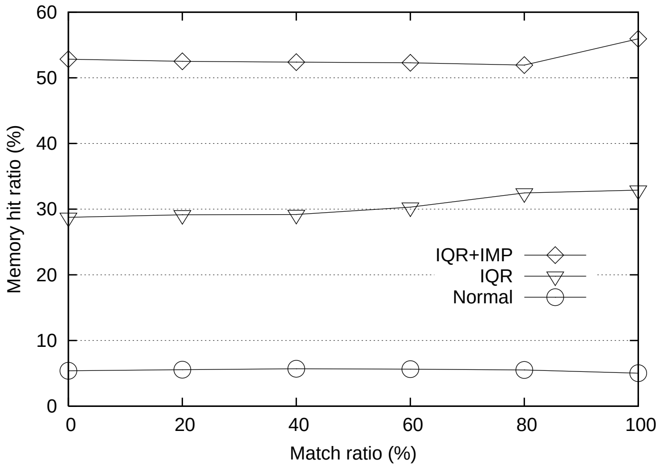

Figure 5.18: (Exp.4) Ratio of segments in memory as the ratio of matches increases.

when enabling stop rules. The ratio is denoted as the percentage of the queries that have matches in the collection. For example, if the ratio is 80%, then 80% of the queries result in a match (recall that the stop rules typically stop after 10 descriptors if the image is located in the collection) while the other 20% are queries that have no matches. We measured the system with the following ratios: 0%, 20%, 40%, 60%, 80% and finally 100%. We fixed the number of workers to 10.

Figure 5.18 shows that increasing the match ratio does not affect memory hit ratio much. Figure 5.19, which shows the throughput, indicates that the system's throughput increases substantially when increasing the match ratio.

In Figure 5.19, we see a steady increase in performance while increasing the match ratio. When the match ratio is 100%, the performance almost doubles when IQR is enabled! This increase cannot be explained by the memory hit ratio since it stays more or less the same. The reason for this can be explained as following.

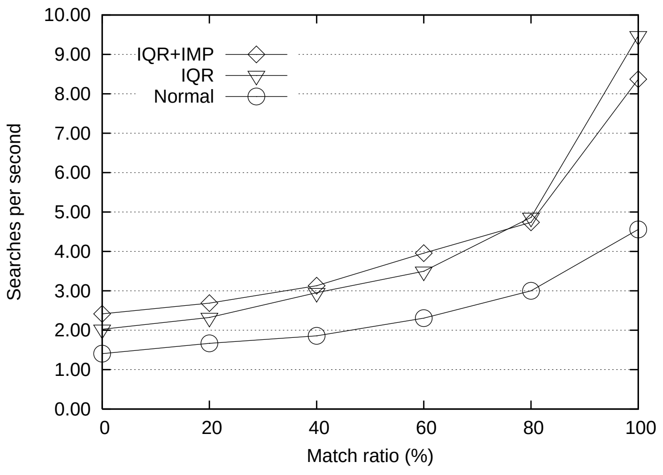

Figure 5.19: (Exp.4) Searches per second as the ratio of matches increases.

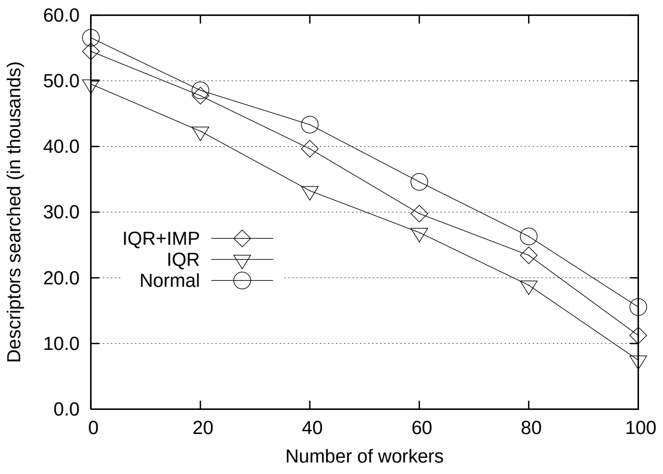

Figure 5.20: (Exp.4) The total number of descriptors searched as the ratio of matches increases.

Figure 5.20 shows the number of descriptors evaluated while the experiment was running. As the match ratio is increased, the number of descriptors evaluated decreases accordingly because of the stopping rules. When comparing the number of descriptors searched at the 80% and 100% mark, we witness a 50% decrease when IQR is enabled. This is reasonable, since when the match ratio is 80%, every fifth query needs to evaluate at least 100 descriptors. When the match ratio is 100%, however, all queries have matches, so only 10 descriptors need to be evaluated. Due to this, the system's performance increases accordingly.

Throughout these experiments, we have shown, using two image collections, that the NV-Network scales at least linearly with respect to the number of worker machines, and that enabling IQR and IMP gives the best performance. We demonstrated the throughput of the system using up to 200 worker machines and 40 concurrent queries, and showed that the system achieves very high-throughput without sacrificing result quality.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In previous work, we have proposed a new indexing structure, the NV-tree, which repeatedly segments the descriptor collection based on projections to random lines. Although using the NV-tree yields performance which is orders of magnitude faster than previous approaches, it is still unsuitable for high-throughput environments.

In this thesis, we have presented a new distributed CBIR system, called the NV-Network, which mirrors the NV-tree across many worker machines and uses a coordinator machine to load balance and manage the system. The NV-Network was designed to be scalable, effective and reliable to hardware failures while making only moderate hardware requirements.

In our extensive experiments, we have shown that the NV-Network scales at least linearly with respect to number of worker machines, is scalable to hundreds of worker machines, achieves very high throughput, even when under very high load, and does not sacrifice search quality for speed, unlike other CBIR systems.

We believe, in fact, that the NV-Network is the most scalable, most efficient and most effective CBIR system that currently exists.

6.2 Future Work

In this section, we discuss several interesting ideas for improving the NV-Network, which, due to time constraints, could not be addressed in this thesis.

Address Scalability Issues When Using Stopping Rules: Because of the dynamic behavior of the stop rules, it is always difficult to predict how many descriptors need to be evaluated. The NV-Network tries to minimize unnecessary descriptor evaluation by keeping the local queues at the workers as small as possible and by forwarding the stop signal as soon as possible to the coordinator. However, as clearly witnessed in Section 5.3.1, the NV-Network does not scale linearly when stop rules are used.

Currently, IQR assigns all the query descriptors at once. When all the descriptors have been assigned, however, more and more descriptors are unnecessary evaluated. Therefore, the coordinator should be given a smarter assignment policy when the stop rules are enabled. It could for instance, only send a $subset$ of the query descriptors to the IQR at each time slot instead of sending them all as is done in the current system. In this way, the stop signal from the server will most likely invalidate most of the remaining query descriptors before they are assigned and sent to the workers.

Global Memory Management: As we saw in Section 5.3.1, it is rarely the case that a worker has both segments in memory. A very interesting approach for enhancing the system's throughput is therefore to look into global memory management (see [10] for details) by borrowing segments from other workers.

This could be implemented as follows: Worker $w1$ receives a segment pair from the coordinator and discovers that it has one segment in memory, but not both. Now, consider that some other worker $w2$ has this missing segment already in memory. In this case, the coordinator could have piggybacked the location of $w2$ into the message. Then, $w1$ could send a message to $w2$ requesting the missing segment. The missing segment would then be sent over the network to $w1$, likely increasing the performance.

Scalability of the Descriptor Extraction: We have already implemented a web service on top of the NV-Network [12]. When using this system, the user uploads the query image, the web application extracts the query descriptors and forwards them to the coordinator. Here, the bottleneck is not the query evaluation but the descriptor extraction. In this thesis we have focused on the scalability and efficiency of the search, but we have not spent time on the scalability of the descriptor extraction. We showed that a single search could be evaluated in a fraction of a second; the descriptor extraction, however, can take seconds. This is left for future work.

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(2/3):108–124, 2001.

[2] H. Balakrishnan and et al. Looking up Data in P2P Systems.

[3] S. Berchtold, C. Böhm, and H. P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of ACM SIGMOD*, pages 142–153, Seattle, WA, 1998.

[4] S. Berrani, L. Amsaleg, and P. Gros. Approximate searches: k-neighbors + precision. In *Proceedings of ACM CIKM*, pages 24–31, New Orleans, LA, 2003.

[5] J. L. Bosque, O. D. Robles, A. Rodriguez, and L. Pastor. Study of a Parallel CBIR Implementation using MPI. In *Proceedings of IEEE CAMP*, pages 195–204, Washington, DC, USA, 2000.

[6] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using gossiping to build content addressable P2P information sharing communities. In *Proceedings of HPDC*. IEEE Press, June 2003.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of PODS*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[8] C. Dwork, S. Ravi Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of WWW*, pages 613–622, 2001.

[9] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of ACM SIGMOD*, pages 301–312, New York, NY, USA, 2003. ACM Press.

[10] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In Li-Yan Yuan, editor, *Proceedings of VLDB*, pages 596–609. Morgan Kaufmann, 1992.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of VLDB*, pages 518–529, Edinburgh, Scotland, 1999.

[12] B. Þ. Jónsson H. Lejsek, F. H. Ásmundsson and L. Amsaleg. Blazingly fast image copyright enforcement, demonstration paper. In *Proceedings of ACM Multimedia*, Santa Barbara, CA, USA, 2006. To appear.

[13] B. Þ. Jónsson H. Lejsek, F. H. Ásmundsson and L. Amsaleg. Scalability of local image descriptors: A comparative study. In *Proceedings of ACM Multimedia*, Santa Barbara, CA, USA, 2006. To appear.

[14] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of Alvey Vision Conf.*, pages 147–151, Manchester, England, 1988.

[15] A. Joly, C. Frélicot, and O. Buisson. Robust content-based video copy identification in a large reference database. In *Proceedings of CIVR*, pages 414–424, Urbana-Champaign, IL, USA, 2003.

[16] Y. Ke and R. Sukthankar. PCA-SIFT: A more distinctive representation for local image descriptors. In *Proceedings of CVPR*, pages 506–513, Washington, DC, USA, 2004.

[17] Y. Ke, R. Sukthankar, and L. Huston. Efficient near-duplicate detection and sub-image retrieval. In *Proceedings of ACM Multimedia*, pages 869–876, New York, NY, USA, 2004.

[18] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. Efficient and effective image copyright enforcement. In *Proceedings of BDA*, Saint Malo, France, 2005.

[19] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. In *Proceedings of IEEE Transactions on Knowledge and Data Engineering*, pages 792–808, 2002.

[20] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[21] W. Muller and A. Henrich. Fast retrieval of high-dimensional feature vectors in P2P networks using compact peer data summaries. In *Proceedings of MIR*, pages 79–86, New York, NY, USA, 2003.

[22] C. H. Ng and K. C. Sia. Peer clustering and firework query model. In *Proceedings of WWW, poster*, 2002.

[23] C. H. Ng, K. C. Sia, and C. H. Chan. Advanced peer clustering and firework query model in the peer-to-peer network. In *Proceedings of WWW, poster*, 2003.

[24] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB*, New York, NY, USA, 1998.