NV-tree: A Scalable Disk-Based High-Dimensional Index

Herwig Lejsek
Doctor of Philosophy
May 2015
School of Computer Science
Reykjavík University

Ph.D. DISSERTATION

ISSN 1670-8539

NV-tree:
A Scalable Disk-Based High-Dimensional Index

by

Herwig Lejsek

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

May 2015

Thesis Committee:

Björn Þór Jónsson, Supervisor
Associate Professor, Reykjavik University, Iceland

Laurent Amsaleg
Research Scientist, IRISA-CNRS, France

Yngvi Björnsson
Professor, Reykjavik University, Iceland

Thesis Examiner:

Shin'ichi Satoh
Professor, National Institute of Informatics, Japan

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this thesis entitled **NV-tree: A Scalable Disk-Based High-Dimensional Index** submitted by Herwig Lejsek in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

_____

Date

_____

Björn Þór Jónsson, Supervisor
Associate Professor, Reykjavik University, Iceland

_____

Laurent Amsaleg
Research Scientist, IRISA-CNRS, France

_____

Yngvi Björnsson
Professor, Reykjavik University, Iceland

_____

Shin'ichi Satoh, Examiner
Professor, National Institute of Informatics, Japan

_____

Date

_____

Herwig Lejsek
Doctor of Philosophy

NV-tree:
A Scalable Disk-Based High-Dimensional Index

Herwig Lejsek

May 2015

Abstract

This thesis presents the NV-tree (Nearest Vector tree), which addresses the specific problem of efficiently and effectively finding the approximate k-nearest neighbors within large collections of high-dimensional data points. The NV-tree is a very compact index, as only six bytes are kept in the index for each high-dimensional descriptor. It thus scales extremely well when indexing large collections of high-dimensional descriptors. The NV-tree efficiently produces results of good quality, even at such a large scale that the indices can no longer be kept entirely in main memory. We demonstrate this with extensive experiments presenting results from various collection sizes from 36 million up to nearly 30 billion SIFT (Scale Invariant Feature Transform) descriptors.

We also study the conditions under which a nearest neighbour search provides meaningful results. Following this analysis we compare the NV-tree to LSH (Locality Sensitive Hashing), the most popular method for ϵ-distance search, showing that the NV-tree outperforms LSH when it comes to the problem of nearest neighbour retrieval. Beyond this analysis we also discuss how the NV-tree index can be used in practise in industrial applications and address two frequently overlooked requirements: dynamicity—the ability to cope with on-line insertions of new high-dimensional items into the indexed collection—and durability—the ability to recover from crashes and avoid losing the indexed data if a failure occurs. As far as we know, no other nearest neighbor algorithm published so far is able to cope with all three requirements: scale, dynamicity and durability.

NV-tree:
Skalanlegur diskamiðaður vísir fyrir margvíð gögn

Herwig Lejsek

Maí 2015

Útdráttur

Í þessari ritgerð setjum við fram vísinn NV-tré (e. NV-tree) sem lausn á ákveðnu afmörkuðu vandamáli: að finna, á hraðvirkan og markvirkan hátt, nálgun á k næstu nágrönnum í stóru safni margvíðra gagnapunkta. NV-tréð er mjög fyrirferðarlítill vísir, þar sem aðeins sex bæti eru geymd fyrir hvern margvíðan lýsivektor (e. descriptor). NV-tréð skalast því mjög vel þegar því er beitt á stór söfn margvíðra lýsivektora. NV-tréð skilar góðum niðurstöðum á skömmum tíma, jafnvel þegar vísarnir komast ekki fyrir í minni. Við sýnum fram á þetta með niðurstöðum tilrauna á söfnum sem innihalda frá 36 milljónum upp í nærri 30 milljarða SIFT (e. Scale Invariant Feature Transform) lýsivektora.

Við rannsökum einnig þau skilyrði sem þurfa að vera fyrir hendi til að leit að næstu nágrönnum skili merkingarbærum niðurstöðum. Í framhaldi af þeirri greiningu berum við NV-tréð saman við LSH (e. Locality Sensitive Hashing), sem er vinsælasta aðferðin fyrir ϵ-fjarlægðarleit, og sýnum að NV-tréð er mun hraðvirkara en LSH. Til viðbótar við þessa greiningu ræðum við hagnýtingu NV-trésins í iðnaði og uppfyllum tvær þarfir sem oft er litið framhjá: breytileika (e. dynamicity)—getu til að höndla í rauntíma viðbætur við lýsingasafnið—og varanleika (e. durability)—getu til að endurheimta vísinn og forðast gagnatap ef um tölvubilun er að ræða. Að því er við best vitum, uppfyllir enginn annar þekktur vísir allar þessar þrjár þarfir: skalanleika, breytileika og varanleika.

Acknowledgements

Thanks to ...

Publications

Much of the text of this thesis has appeared in the following four publications:

1. Lejsek, Ásmundsson, Jónsson, and Amsaleg (2009);
2. Lejsek, Jónsson, and Amsaleg (2011);
3. Lejsek, Jónsson, and Amsaleg (2015) (under review); and
4. Amsaleg, Jónsson, and Lejsek (2015) (under review).

More specifically, the text of these publications is represented in the thesis as follows:

- Chapter 3, Overlapping NV-tree, contains results and text from Lejsek, Ásmundsson, Jónsson, and Amsaleg (2009), adapted to the context of the thesis.
- Chapter 4, Non-Overlapping NV-tree, contains results and text from Lejsek et al. (2011), adapted to the context of the thesis.
- Chapter 5, Industry Strength NV-tree, contains results and text from Lejsek et al. (2015), except Section 5.5 which is adapted from Amsaleg et al. (2015).

I fully acknowledge that these papers are written in collaboration with my co-authors and that this thesis therefore contains significant text that is not written by me, as well as some ideas that have been conceived in collaboration. In particular, most of the text of Amsaleg et al. (2015) originates from Amsaleg (2014), as well as some portions of the text of Lejsek et al. (2015).

Any text that I have not written is used with full permission from the co-authors. The bulk of the ideas represented in this thesis, however, are mine; large portions of the various papers were written or drafted by me; and I have performed and analysed all experiments described in this thesis, with one notable exception: the experiments and analysis reported in Section 5.5 were performed by Laurent Amsaleg, but in collaboration with me. I felt it important to include these results, however, as they represent the most recent steps of our investigation into the scalability of the NV-tree index structure.

Aside from my work on the scalability, dynamicity and durability of the NV-tree, I have also worked on many other aspects of scalability of near-duplicate detection of images and videos, both as part of my Ph.D. studies and within Videntifier Technologies. Several other related publications have been made from that work during this time, some of which are cited in this thesis. Furthermore, two successful patent application have been filed. I have chosen to focus this thesis on the NV-tree, however, as this work represents a complete timeline from the inception of the idea to a full-scale industry application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nearest neighbour search in high-dimensional space is a theoretical problem that has impact on many practical applications. These practical applications range from content-based image retrieval and copyright protection through finding correlations in stock data or geo statistics (e.g. in the field of metrology) up to analysing and aligning similar organic molecules with desired properties. Nearest neighbor search is therefore a field of interest for many different research communities, and over the last two decades significant research effort has been spent trying to improve its efficiency. In the work described in this thesis, we have focused on one particular field of usage—search for visual content in the context of near-duplicate detection for images and videos for the purpose of detecting copyright infringements. Such techniques have many applications, such as copyright detection and multimedia forensics, and our emphasis is on the properties required for large-scale industrial applications.

1.1 From Labs to the Real World

It must be mentioned at this stage that the database index described in this thesis is the foundation of the Icelandic start-up company Videntifier Technologies and has been deployed in practise within law enforcement. The main area of usage is the analysis of large amounts of illegal image and video content, typically content depicting the sexual abuse of children or extreme terrorist and hate-speech propaganda that is distributed over the world wide web. The technology is already in use with the ICSE database of Interpol, the CRIS system of the US National Center of Missing and Exploited Children (NCMEC), the UK Home Office and the UK secret service. Each of these organisations deals with

collections of tens of millions of image files and video files, with a combined total length of more than 100,000 hours.

Obviously these large collections of multimedia cannot be processed any longer by humans and must be supported by computer automation. The computer logically connects visually similar content, so that the investigator can quickly compare and analyse only relevant visual information and—in the best case—connect with some additional intelligence that thus helps to recover the victim and find the perpetrators of the crime. Another closely related application is the pre-analysis of content found on seized storage devices, presented in the two publications and one keynote in the Multimedia in Forensics workshops (Lejsek, Ásmundsson, Daðason, et al., 2009; Lejsek et al., 2010; Lejsek, 2010).

Special emphasis is put on the scalability of the proposed algorithm, a property that is heavily underestimated in practical life. Due to the proliferation of high-speed Internet, and the spread of mobile phones and digital cameras, the sheer number of image and video files distributed goes into the trillions. So even in small, well-defined niche applications the collection quickly grows to millions or even billions of files. This fact has been addressed to some extent within the research community, as scalability is central to many recent research contributions. But in real life many managers and business leaders have not yet understood that problems might become significantly more complex with scale and thus performance parameters correlated with problem size are often not considered.

The vast majority of the research contributions focusing on scalability are designed to work in main memory only and explicitly disregard disks. The reason for this design decision lies in the emphasis on performance, as working with a multimedia collection in memory helps guaranteeing small response times. Therefore, these techniques typically compress severely the representation of the multimedia features to fit more and more data into memory in order to keep the disks out of the way. However, such indexing schemes cannot handle collections larger than main memory and we believe that this is not a sustainable strategy for industry—that eventually data quantity will win over memory capacity.

In the real world, applications dealing with multimedia material must also face two very important challenges in addition to *scalability*. The first challenge is *dynamicity*—the ability to cope with on-line insertions of new high-dimensional items into the indexed collection while the system is up and running. Facebook, for example, claims that its collection grows by roughly 1.4 billion images every week; coping with such dynamic collections has not been considered in the literature of nearest neighbor retrieval. The

second challenge, which has not been studied in recent literature either, is *durability*—the ability to recover from crashes and avoid losing the indexed data if a failure occurs.

It turns out that, as far as we know, no nearest neighbor algorithm published so far is able to cope with scale, dynamicity and durability. This may have prevented companies from developing business models around multimedia similarity searches. In this thesis, we have therefore proposed the NV-tree, a disk-based high-dimensional index, which handles all three industry requirements. While the work on compressing descriptors to fit larger and larger collections into memory is important, we believe that an index that gracefully adapts to disk-based processing is an equally important building block for truly scalable multimedia applications.

1.2 Contributions

The remainder of this thesis is organised into five chapters. Chapter 2 describes the state of the art in high-dimensional indexing and then summarizes the differences between the proposed NV-tree structure and alternative approaches. The main contributions of this thesis are then presented in Chapters 3 to 5, before concluding in Chapter 6.

As each of Chapters 3 to 5 is based on one publication (or two, in the case of Chapter 5) this order represents a time-line of our investigation into the properties of the NV-tree. Note that since the results are obtained over a long period of time, the hardware used to obatin the initial results is now outdated. We believe, however, that this does not invalidate those results and the demonstrated tradeoffs.

The main contributions of Chapters 3 to 5 can be summarised as follows:

- In Chapter 3 we first propose the NV-tree, a disk-based data structure that gives good approximate answers with a *single random disk read*, even for very large collections of high-dimensional data. Furthermore, searching the NV-tree incurs negligible CPU overhead, making it suitable for main-memory based processing. We describe the fundamentals of the NV-tree, as well as different strategies for its construction.

  Second, we analyze the properties of a large-scale copy detection application using the well known SIFT descriptors (Lowe, 2004). We show that SIFT descriptors are very distinctive and have high contrast. Furthermore, we show that using contrast-based ground truth sets is necessary to obtain meaningful results for all queries.

Third, we analyze the performance of the NV-tree and show that the NV-tree works well for our workload. We show that using a single NV-tree yields high recall, but also a number of false positives. By combining the results from two or three NV-trees, however, most of those false positives can be avoided while retaining the high recall.

Fourth, we compare the NV-tree to two competing data structures. In particular, we focus on LSH which is currently a popular high-dimensional indexing method. We show that LSH can return results of similar quality, but only by using many more disk reads.

- In Chapter 4 we then look at the performance implications of redundancy in the original NV-tree, caused by overlapping partitions. We show that it is simply necessary to remove the overlap when indexing ever larger collections of high-dimensional feature vectors. We show that removing the overlap does in fact reduce result quality, but we propose three different strategies to "re-capture" that result quality; these strategies more than compensate for the losses due to lack of redundancy.

  Secondly, we present two performance studies which compare the new "overlap-free" NV-tree with previous results, showing that although more non-overlapping NV-trees are required for retrieval quality, each index is so much smaller that retrieval is actually faster and that retrieval quality and performance are not affected significantly when the collection size grows to 2.5 billion descriptors.

- In Chapter 5 we finally demonstrate how to enforce ACID properties (atomicity, consistency, isolation, and durability) within the NV-tree for a large class of important applications, and show that with our implementation dynamic inserts can be efficiently managed.

  We then show detailed performance evaluations of the scalability of the NV-tree using standard image benchmarks embedded in collections of up to 30 billion high-dimensional vectors. Our analysis of the literature shows that these are by far the largest single-server experiments reported anywhere.

Overall, we show that the NV-tree is not only an extremely scalable approximate indexing strategy, but also a unique approximate nearest neighbor search system that achieves scalability, dynamicity and durability, and thus meets industrial standards for a database management solution.

Chapter 2

Background

There are many studies in the literature that address the problem of nearest neighbour search in high-dimensional space. In this chapter we sketch the development since the early 1980s and how the community has slowly been progressing towards solving this phenomenon that we refer to today as the curse of dimensionality. This term was coined by Bellmann (1961), who was the first to point out that this problem was far from trivial and—as we know today—is only solvable (in a meaningful way) for data collections with well-defined characteristics.

As all of our experiments are related to content-based image retrieval we start by briefly introducing the developments in this particular domain in order to give the reader a better understanding of the overall problem domain and the applications this thesis targets. As our focus is squarely on methods for near-duplicate detection, other related areas are not considered, such as methods for semantic content analysis, classification or clustering. See Datta, Joshi, Li, and Wang (2008) and Chang (2011) for detailed surveys of the field.

2.1 Content-Based Retrieval

Since the early days of content-based image retrieval, computer vision researchers have tried to capture the most important visual characteristics of a given piece of content in so-called feature vectors. In these early days such vectors only extracted the most basic features, namely color and simple shape information, and encoded this information into a high dimensional vector, typically ranging from 64 to 256 dimensions. In these vectors,

each entry is typically a number computed from specific properties of the content, such as particular range of pixel colors in a color histogram.

In order to query such a content-based retrieval system, the user could either perform a query by example (e.g., choosing a displayed image and asking for images similar to the selected one) or specify a specific color distribution. As a similarity metric, most researchers chose Minkowski metrics, in particular Euclidean or Manhattan distance, because of their simple implementation and relatively low computational complexity.

In the 1980s the typical strategy for performing content-based retrieval was to extract only a single, global feature vector from every piece of visual content. This strategy made the overall description of the content very imprecise because each vector failed to capture details of the visual content and also because these feature vectors were often compressed (e.g., through dimensionality reduction) to apply good high dimensional indexing techniques that were both fast in computation and high in retrieval quality. Due to the imprecision of the description, these strategies exhibited severe lack of robustness against simple image modifications; color histograms, for example, can be foiled simply by cropping away parts of an image that are of minor interest. Also their recognition power suffers since extracting just one major shape or texture is not descriptive enough for a whole image, which usually contains many small details which are important for our perception.

In order to overcome the lack of recognition power due to using just one global descriptor, Florack, ter Haar Romeny, Koenderink, and Viergever (1994) devised the use of a fine-grained image recognition scheme for grey-level images based on local descriptors. Each of these local descriptors describes a small, local area of the image, so-called visual interest points. Such interest points are calculated using specific visual algorithms and typically reflect areas with specific visual characteristics (e.g., edges, corners, or contrast changes). The number of descriptors per image can vary significantly, depending on the size, resolution, quality and contents of the images. For typical images, several hundreds of descriptors may be created; for large, high quality images, even more than a thousand descriptors. To know which image a descriptor has been computed from, image identifiers are stored together with the descriptors.

Lowe (1999) introduced the so-called SIFT (Scale Invariant Feature Transform) local descriptor method by making efficient use of scale-space theory and the Gaussian kernel (DoG = Difference of Gaussians) for interest point detection. The scale-space theory was developed by Lindeberg (1994) and discussed in the concept of interest point detection by (Lindeberg, 1998). It finds interest points when the DoG generates a blob-like structure at a particular image scale, which happens especially in areas of the image with strong

contrast changes. Brown and Lowe (2002) improved the method in 2002 by introducing inter-pixular point locations and Lowe (2004) described in even more detail the parameter choices and the encoding of the gradient histogram around the interest points into a characteristic 128-dimensional feature vector of unified length.

Local descriptor schemes have shown to be robust to many different types of image modifications (e.g., see Amsaleg (2014) for a summary of the literature). They are insensitive to a whole variety of different transformations, such as resizing, color variation, cropping, rotation, jpeg-compression, mirroring, various illumination changes, partial occlusions, etc. In order to evaluate the robustness of different local descriptor schemes, the copyright protection benchmarking tool Stirmark, created by Petitcolas et al. (2001), is typically used. Stirmark takes a set of images and transforms them using various graphical filters, including rotation, rescaling, cropping, affine distortions, jpeg-compression, convolution filters, and many combinations of such transformations.

Local image descriptors can be extended to video retrieval, either by computing descriptors for key frames or by finding descriptors that are prominent in many frames. See Laptev (2005) for a survey of video retrieval methods.

To visually compare two images using a local descriptor scheme, the two descriptor sets are compared by calculating the Euclidean distance between each pair of descriptors; short distances indicate a match, while longer distances indicate differences. The strategy for identifying similar images (e.g., near-duplicates) within a large image collection is very similar. In this case a k-nearest neighbor query is run for each local descriptor computed on the query image. Each retrieved nearest neighbor "votes" for the image in the collection that it is computed from. Then, the most similar images are found by ranking the images according to their number of votes. In general, the number of images receiving at least one vote is very large. If the database does not contain any image that is similar to the query image, then the votes of all the images returned are roughly similar and of small values. In contrast, if one or more images are indeed similar, then they have many more votes.

Because of their high robustness and their fast extraction speed the SIFT descriptors have become a de-facto standard in the field of content-based image retrieval and we used them as reference descriptors throughout the entire thesis. It must be mentioned, however, that there exist a whole range of alternative algorithms that might even perform better for certain transformations. Performance evaluations of various local descriptor schemes were reported by, for example, Mikolajczyk and Schmid (2003, 2005) and Lejsek, Ásmundsson, Jónsson, and Amsaleg (2006).

Furthermore, as mentioned in the Introduction, most recent research contributions focusing on scalability are designed to work in main memory only and explicitly disregard disks. This is done, e.g., by aggregating the many SIFT descriptors to a single (or a few) very high-dimensional descriptor, which is then indexed. The reason for this design is that working with a multimedia collection in memory helps guaranteeing small response times. Good representatives of this approach are the VLAD descriptors by Arandjelovic and Zisserman (2013) and work by Jégou et al. (2012).

In this thesis, we use large SIFT collections because we are focusing on the scalability of retrieval; since the SIFT descriptors produce very large collections of descriptors, they are the perfect candidate for our experimental evaluations. While the work on compressing descriptors to fit larger and larger collections into memory is also important, we believe that an index that gracefully adapts to disk-based processing is an equally important building block for truly scalable multimedia applications.

2.2 Exact Nearest Neighbor Algorithms

Early proposals for handling high-dimensional vectors date back to the 1980s. First approaches, such as the R-tree and KD-tree, have shown to work well for relatively low-dimensional data with dimensionality of up to 10–15. When turning to higher dimensional spaces, however, these approaches turned out to deliver worse performance than a sequential scan through the whole collection.

In the following years several other solutions for solving neighbor search in higher dimensional space have been discussed. Most of them were derived from the above mentioned data structures (e.g., the SR-tree by Katayama and Satoh (1997) or the K-D-B-tree by Robinson (1981)), but none of them worked particularly well for large collections of high-dimensional data.

Then in 1999 and 2000 two independent papers were published that changed the research direction on this topic significantly. Beyer, Goldstein, Ramakrishnan, and Shaft (1999) and Hinneburg, Aggarwal, and Keim (2000) studied the question of under what circumstances nearest neighbor search in high-dimensional space actually creates meaningful results. Their conclusion was that nearest neighbor search for a given query vector was only meaningful when the neighbor vectors were significantly closer to the query vector than all the other vectors in the collection. In the case when vectors were randomly distributed over the whole high-dimensional space, all vectors are nearly equidistant to each other, thus rendering the retrieval of nearest neighbors totally pointless.

2.3 Approximate Nearest Neighbor Algorithms

Due to this fact, researchers began looking into approximate indexing methods. It started with approximating an exhaustive search through the whole descriptor collection (Weber, Schek, & Blott, 1998) and then diverted into three different directions. One line of work is based on indexing data clusters. One of the most representative technique is the hierarchical k-means decomposition of the data collection: Voronoi cells are created to partition and store the high-dimensional vectors, and the cells are organized as a multi-level tree to facilitate traversal and improve response time (Fukunaga & Narendra, 1975). Many variants of this basic idea have been proposed (e.g., see (Li, Chang, Garcia-Molina, & Wiederhold, 2002; Nistér & Stewénius, 2006; Fraundorfer, Stewénius, & Nistér, 2007; Philbin, Chum, Isard, Sivic, & Zisserman, 2007; Chierichetti et al., 2007; Philbin, Chum, Isard, Sivic, & Zisserman, 2008; Guðmundsson, Jónsson, & Amsaleg, 2010)). This work has been extended to cope with collections of up to 30B descriptors, using distributed "big data" techniques such as Hadoop, resulting in the DeCP algorithm (Guðmundsson, Amsaleg, & Jónsson, 2012; Shestakov, Moise, Guðmundsson, & Amsaleg, 2013; Moise, Shestakov, Guðmundsson, & Amsaleg, 2013; Guðmundsson, 2013).

A second line of work developed around the idea of approximate (fuzzy) hashing. Fuzzy hashing approaches typically only look for neighbors within a certain distance radius of the actual query vector. The earliest notable approximate method was Locality Sensitive Hashing (LSH), which is an approximate hashing scheme initially proposed by Gionis, Indyk, and Motwani (1999) for Hamming distance and later extended to handle Euclidean distance by Datar, Indyk, Immorlica, and Mirrokni (2006). Basically, LSH uses a large number of hashing functions to project high-dimensional vectors onto segmented random lines. At query time, the hash tables are probed with the query vector, and candidates from all these hash tables are then aggregated to find the true neighbors. The performance of such hashing schemes is highly dependent on the quality of the hashing functions. Hence, many approaches have been proposed to improve hashing (e.g., see (Weiss, Torralba, & Fergus, 2008; Jain, Kulis, & Grauman, 2008; Tao, Yi, Sheng, & Kalnis, 2009; Wang, Kumar, & Chang, 2010; Paulevé, Jégou, & Amsaleg, 2010; Zhang, Agrawal, Chen, & Tung, 2011; Jin et al., 2013)) as well as to reduce the number of hash tables, which in turn diminishes the high storage costs of these tables (Lv, Josephson, Wang, Charikar, & Li, 2007; Joly & Buisson, 2008). Tuning hash functions is reported to be a complicated task and some schemes try to automatically adapt to the data distribution (Bawa, Condie, & Ganesan, 2005).

A third approach is based on the idea of a search tree structure. The NV-tree is one proponent of this group. Another piece of work which has influenced the idea of the NV-tree significantly is the concept of median rank aggregation. This concept was first applied by Fagin, Kumar, and Sivakumar (2003) and builds upon the idea that not the actual feature vectors are stored in the indexing structure but rather references that are aggregated over a whole set or indexing structure, returning the feature with the highest references as the nearest neighbour, the one with second highest as the second highest and so on. Major drawback of that algorithm, whose basic search structure is a set of B^+-trees, is the excessive search across the individual trees making it necessary to aggregate about 5% of the total data collection. Yet another approach in the category of search trees are the Metric tree by Uhlmann (1991) and a variant named Spill-tree proposed by Liu, Moore, Gray, and Yang (2004), which is a tree-structure based on splitting dimensions in a round-robin manner, and introducing (sometimes very significant) overlap in the split dimension to improve retrieval quality.

There are a couple of very recent approaches. Muja and Lowe (2014) proposed, through the FLANN library, a series of high-dimensional indexing techniques based on randomized KD-trees, k-means indexing and random projections. Jégou, Douze, and Schmid (2011) proposed a slightly different high-dimensional indexing scheme, called product quantization. It has been demonstrated that indexing data clusters works best when the representatives truly capture the location of points in the high-dimensional space, that is, when using a very large value for k. This is indeed observed by Nistér and Stewénius (2006). As it is computationally difficult to run k-means with a very large k (in the order to several hundred millions to billions), however, product quantization decomposes the high-dimensional space into low-dimensional subspaces which are indexed independently. This produces very compact code words representing the vectors that, when used together with an asymmetric approximate distance function, exhibit very good performance for a moderate memory footprint. Several variants of product quantization have been published (e.g., see (Xioufis, Papadopoulos, Kompatsiaris, Tsoumakas, & Vlahavas, 2014; Ge, He, Ke, & Sun, 2014; Kalantidis & Avrithis, 2014; Heo, Lin, & Yoon, 2014)); in particular Sun, Wang, Xu, and Zhang (2013) propose an indexing scheme based on product quantization, using ten computers to fit the 1.5 billion images collection they index in memory.

2.4 Discussion from a Database Perspective

Overall, most of the high-dimensional indexing schemes for nearest neighbor search disregard disks. They typically tackle the scalability problem by relying on clever and effective compression mechanisms for the feature vectors—the best example may be the schemes based on product quantization. However, there is a limit to the number of images that can be indexed. According to Jégou et al. (2011), an extremely well optimized indexing scheme based on product quantization needs 32 to 128 bytes per image for near duplicate detection, and several kilobytes for object recognition. Even with the most compact representation, it is thus difficult to go beyond 10 billion images as a computer with more than 320 GB of main memory becomes quite expensive. Going to distributed settings is possible, but this adds significant complexity and increases the likelihood of failures. Please note also that the ability of most indexing techniques to cope with dynamic inserts remains a question, in particular at such a large scale.

Many variants of the original R-trees and KD-trees do take disks into account and support dynamic inserts. Concurrency control algorithms have been developed for these two indexing schemes and they can be made fault tolerant by implementing the write-ahead-logging protocol. These two approaches, however, are known to perform poorly when indexing very large collections of high-dimensional data. Multiple randomized KD-trees, as presented by Muja and Lowe (2014), cope much better with scale. The datasets they used were large, both when a single server was used and when distributed search across multiple machines was used to cope with the 80 million tiny images of Torralba, Fergus, and Freeman (2008). However, it has not yet been demonstrated that randomized KD-trees can handle collections containing a billion vectors or more. The API in the FLANN library for randomized KD-trees only allows for bulk-loading the index, with no suggestion that dynamic inserts are supported. Note, however, that the index can be pushed to disk and later read back, but no comment on recovery is provided by Muja and Lowe (2014).

The Spill-tree has shown to work for a collection of up to 1.5 billion high-dimensional feature vectors representing the same number of images (Liu, 2006; Liu, Rosenberg, & Rowley, 2007). In that study two thousand workstations were used, presumably having at least a terabyte or two of total main memory. Due to the significant added complexity using such massive amounts of resources cannot be considered economical. Already Gray and Putzolu (1987) and Gray and Graefe (1997) showed that using very large main memory is not economical; that data which is accessed less frequently than every five minutes should not be kept in main memory.

Sun et al. (2013) needed 10 common servers to support real-time search on 1.5 billion images. Each server indexed between 100 and 200 million images with about 60 GB of memory. While large-scale collections are addressed well by this work, it is completely main-memory oriented which explains why the aggregated memory of ten servers is needed to fit the collection. Disks are not used, so the index is not persistent and does not resist failures, and no information on dynamic inserts is given. If one server fails, the entire system is down and re-indexing the images might be needed. Similar comments can be made about the work described by Jégou et al. (2011) where 2 billion vectors are indexed (on a single machine, however).

Zäschke, Zimmerli, and Norrie (2014) combine binary patricia-tries (Nickerson & Shi, 2008) with a multi-dimensional approach similar to quadtrees while being navigable through hypercubes. The resulting PH-tree is able to store its data pages on disk to provide persistent storage for the indexed data and Zäschke et al. (2014) claim the PH-tree can handle updates, but neither consistency issues nor support for the ACID properties of transactions are discussed.

The only method which addressed scale similar to that reported in this thesis, is the DeCP algorithm (Moise, Shestakov, Guðmundsson, & Amsaleg, 2013; Guðmundsson, 2013). This work, which stems from the same research group as the work reported in this thesis, focuses on a very simple disk-based clustering method—essentially the first step of a k-means clustering algorithm—but uses a large number of workstations to index the collection and answer queries. While the scale of the experiments is indeed impressive, the philosophy of the system is quite different: it is dedicated to processing large batches of queries and cannot run interactively. Furthermore, that work has not addressed the dynamicity and durability requirements.

The NV-tree is a disk-based data structure, which builds upon a combination of projections of data points to lines and partitioning of the projected space. By repeating the process of projecting and partitioning, data is eventually separated into small partitions which can easily be fetched from disk with a single disk read, and which are highly likely to contain all the close neighbors in the collection. In this thesis we develop the NV-tree as a full-fledged database solution, addressing all three requirements of scalability, dynamicity and durability.

Chapter 3

Overlapping NV-tree

This chapter addresses approximate disk-based search in very large high-dimensional collections. It makes several major contributions:

- First, we propose the NV-tree, a disk-based data structure that gives good approximate answers with a *single random disk read*, even for very large collections of high-dimensional data. Furthermore, searching the NV-tree incurs negligible CPU overhead, making it suitable for main-memory based processing. We describe the fundamentals of the NV-tree, as well as different strategies for its construction.
- Second, we analyze the properties of a large-scale copy detection application using the well known SIFT descriptors (Lowe, 2004). We show that the SIFT descriptors are very distinctive and have high contrast, even in large collections. Furthermore, we show that using contrast-based ground truth sets is necessary to obtain meaningful results for all queries.
- Third, we analyze the performance of the NV-tree and show that the NV-tree works well for our workload. We show that using a single NV-tree yields high recall, but also a number of false positives. By combining the results from two or three NV-trees, however, most of those false positives can be avoided while retaining the high recall.
- Finally, we compare the NV-tree to two competing data structures. In particular, we focus on LSH which is currently a very popular high-dimensional indexing method. We show that LSH can return results of similar quality, but only by using many more disk reads.

The remainder of this chapter is organized as follows. First, we present the NV-tree in Section 3.1, and its implementation details in Section 3.2. Then we present the collec-

tion and workload used in our experiments in Section 3.3. In Section 3.4, we analyze the ground-truth result quality of our workload, and in Section 3.5 we describe the performance of the NV-tree. In Section 3.6 we compare the performance of the NV-tree to that of LSH. We conclude with a summary in Section 3.7.

3.1 The NV-tree

The NV-tree (Nearest-Vector-tree) is a disk-based data structure designed to answer k-nearest neighbor search in very large collections of high-dimensional feature vectors in an approximate, yet very efficient way. In essence, it transforms costly nearest neighbor searches in the high-dimensional space into efficient uni-dimensional accesses using a combination of projections of data points to lines and partitioning of the projected space. By repeating the process of projecting and partitioning, data is eventually separated into small partitions which can be easily fetched from disk with a single disk read, and which are highly likely to contain all the close neighbors in the collection.

The curse of dimensionality suggests that close descriptors might get separated by a partition boundary when partitioning the space. Therefore, the NV-tree also adds redundancy by allowing the partitions to overlap. Due to the redundancy, good approximate results are obtained by processing a single partition. The drawback, of course, is higher storage requirements.

In this section we first outline the algorithms for NV-tree creation (Section 3.1.1) and search (Section 3.1.2). Then we consider strategies for projections (Section 3.1.3) and partitioning (Section 3.1.4). Finally, we highlight key properties of the NV-tree (Section 3.1.5). The implementation of the NV-tree is described in Section 3.2.

3.1.1 NV-tree Creation

Overall, an NV-tree is a tree index consisting of: a) a hierarchy of small *inner nodes*, which are kept in memory during query processing and guide the descriptor search to the appropriate leaf node; and b) larger *leaf nodes*, which are stored on disk and contain references to actual descriptors.

When the construction of an NV-tree starts, all descriptors are considered to be part of a single temporary partition. Descriptors belonging to the partition are first projected onto a single *projection line* through the high-dimensional space. Strategies for selecting the projection lines are discussed in Section 3.1.3.

The projected values are then partitioned into disjunct sub-partitions based on their position on the projection line. For each pair of adjacent partitions, an overlapping sub-partition, covering 50% of both partitions, is created for redundant coverage of the partition borders. Information about all these sub-partitions, such as the partition borders on the projection line, form the inner node of the first level of the NV-tree. Strategies for partitioning are described in Section 3.1.4.

To build subsequent levels of the NV-tree, this process of projecting and partitioning is repeated for all the new sub-partitions using a new projection line at each level, creating the hierarchy of inner nodes. The process stops when the number of descriptors in a sub-partition falls below a specified limit designed to be disk I/O friendly (this limit includes extra space for subsequent insertions). A new projection line is then used to order the descriptor identifiers of the sub-partition, and the ordered identifiers are written to a leaf node on disk.

3.1.2 NV-tree Nearest Neighbor Retrieval Process

During query processing, the query descriptor first traverses the hierarchy of inner nodes of the NV-tree. At each level of the tree, the query descriptor is projected onto the projection line associated with the current node. The search is then directed to the sub-partition with center-point closest to the projection of the query descriptor. This process of projection and choosing the right sub-partition is repeated until the search reaches a leaf node.

The leaf node is fetched into memory and the query descriptor is projected onto its projection line. The search then starts at the position of the query descriptor projection. The two descriptor identifiers on either side of the projected query descriptor are returned as the nearest neighbors, then the second two descriptor identifiers, etc. Thus, the $k/2$ descriptor identifiers found on either side of the query descriptor projection are alternated to form the ranked k approximate neighbors of the query descriptor. Note that if the leaf does not contain k identifiers, then only the identifiers in the leaf are returned.

Note that since leaf partitions have a fixed size, the NV-tree guarantees query processing time of a single disk read regardless of the size of the descriptor collection. Larger collections need deeper NV-trees but the intermediate nodes fit easily in memory and tree traversal cost is negligible.

As the results are based on a projection to a single line, however, false positives do arise when processing a leaf node. Since distances can not be calculated, other means of re-

moving false positives are required. The method we use to eliminate false positives is based on aggregation of the ranked result sets from multiple NV-trees, which are built independently over the same collection. Since each NV-tree is based on random projections, the contents of the ranked results are very likely to differ, except for descriptors that are actual near neighbors. Therefore false positives can largely be eliminated by applying any rank aggregation method to combine results from more than one NV-tree index. The effectiveness of this method is studied in Section 3.5.3.

3.1.3 Projection Strategies

Projecting high-dimensional data points to random lines was introduced by Kleinberg (1997) and subsequently used in some other high-dimensional indexing techniques (Fagin et al., 2003; Datar et al., 2006; Liu et al., 2004). Such projections have two main benefits. First, in some cases, they can alleviate data distribution problems. Second, they allow for a clever dimensionality reduction, by projecting to fewer lines than there are dimensions in the data. Random lines are best generated isotropically in a quasi-orthogonal manner (requiring a minimal angle between pairs of lines).

In the NV-tree, projection lines are used at each level of the tree, and hence a strategy is needed for selecting those lines. The default strategy is a *Random* strategy, which picks random lines as described above; this strategy is simple and data independent. The retrieval quality, however, can be improved with data-dependent generation of lines, for example using the well known Principal Component Analysis (PCA). Instead of picking a random line for a partition, PCA can be run to determine its best projection line; the line with the largest projection variance. Running PCA for each partition, however, is computationally expensive because there are many partitions and each partition holds many points. We have therefore devised a faster *Approximate PCA* strategy for selecting projection lines, which we describe in the remainder of this section.

Before starting the NV-tree creation, a large set of isotropic, quasi-orthogonal random lines is generated and kept in a *line pool* in main memory. During line selection, the partition about to be projected is first sampled. The data points in this small sample are projected onto all the pre-computed lines, and a fraction of the lines with the highest variance is selected. A larger sample of the same partition is then extracted and projected onto only the selected lines. Fewer lines are in turn selected, again the ones with the highest variance. By repeating this process a few times, a single line is finally elected as the projection line of the partition.

Instead of choosing the single best possible line for the partition, determined by costly PCA calculations, this efficient process picks a "reasonably good" line from the large line pool by using many cheap projection calculations over small samples.

3.1.4 Partitioning Strategies

A partitioning strategy is likewise needed for the NV-tree. In the following, we describe three strategies, *Balanced*, *Unbalanced* and *Hybrid*. We end with a discussion of their implications.

The *Balanced* strategy partitions data based on cardinality. Therefore, each sub-partition gets the same number of descriptors, and all leaf partitions are of the same size. Although node fanout may vary from one level to the other, depending on the desired tree height and leaf size, the NV-tree becomes balanced as each leaf node is at the same height in the tree.

It has been observed in the literature that the density of projections of high-dimensional data sets onto a random line generally follows a normal distribution. As a result, the absolute distance between partition boundaries varies significantly along the line with the *Balanced* strategy. Dense areas in the data space have very close boundaries, while sparse areas have more distant boundaries. This strategy may therefore separate close data points from dense areas while storing together distant data points from sparse areas, which can reduce the accuracy of the search.

The *Unbalanced* partitioning strategy avoids this problem, by using distances instead of cardinalities. In this case, sub-partitions are created such that the absolute distance between their boundaries is equal. All the data points in each interval belong to the associated sub-partition. With this strategy, however, the normal distribution of the projections leads to a significant variation in the cardinalities of sub-partitions. Due to the repeated application of the partitioning strategy, the NV-tree becomes unbalanced as dense areas are partitioned more often than sparse areas.

To implement the *Unbalanced* strategy, we calculate the standard deviation σ and mean m of the projections along the projection line. Then a parameter α is used to determine the partition borders as $\ldots, m - 2\alpha\sigma, m - \alpha\sigma, m, m + \alpha\sigma, m + 2\alpha\sigma, \ldots$. Small adjacent sub-partitions are merged until the resulting cardinality hits the leaf node size limit and then written to disk. Sub-partitions containing many data points, on the other hand, are subsequently re-partitioned. Overlapping partitions are created similarly, using σ, m and

α, by shifting the borders by 0.5. For example, the central overlapping partition borders are $m - 0.5\alpha\sigma$ and $m + 0.5\alpha\sigma$.

The sub-partitions farthest away from the mean are likely not to be partitioned again, as their cardinality is such that they fit into a leaf node. Conversely, partitions close to the mean are likely to require further partitioning. Thus, the "center" of an *Unbalanced* NV-tree is typically partitioned deeper than its "sides".

The *Unbalanced* strategy tends to produce significantly larger trees, due to two reasons. First, it frequently creates trees that are deeper on average than the *Balanced* strategy. Due to the overlapping partitions, each additional level in the tree roughly doubles its size. Second, as partitions no longer contain precisely the same number of descriptors, leaf partitions tend to be less filled, resulting in higher space requirement. To give an example, consider a sub-partition which has one more descriptor than would fit in to a leaf partition. In this case, at least three partitions would be created (including the overlapping partition) in place of the one, giving rise to both problems described above.

In order to alleviate this data explosion problem, we propose the *Hybrid* strategy. This strategy follows the *Unbalanced* strategy until a sub-partition is of a size that could fit in l leaf partitions (including extra space for insertions; we have found $l = 6$ to be a good number). The *Balanced* strategy is then used to construct the leaf partitions. As a result, leaf partitions are better utilized and the tree is shallower, resulting in smaller space requirements.

Overall, the *Unbalanced* strategy requires twice as much space as the *Balanced* strategy, while the *Hybrid* strategy is much closer to *Balanced* in size. We have observed that *Unbalanced* and *Hybrid* NV-trees yield equivalent results, but significantly better than *Balanced* NV-trees.

Note that all strategies can be partitioned aggressively, by specifying many sub-partitions in the *Balanced* strategy or a small α in the *Unbalanced* strategy. Aggressive partitioning tends to produce shallow and wide NV-trees, while a "gentle" partitioning scheme tends to produce deep and narrow trees. Aggressively built NV-trees occupy less disk space but may yield lower recall.

3.1.5 Summary

Overall, an NV-tree consists of a hierarchy of small inner nodes, which fits in memory, and larger leaf nodes, which are stored on disk and contain descriptor identifiers. In this

section, we have described the processes for index creation, index search, and insertions and deletions, as well as alternative strategies for the index creation.

One fundamental property of the NV-tree is that it requires a single disk read per query descriptor. This property holds even with very large descriptor collections, making query processing cost largely independent of the collection size.

Another fundamental property of the NV-tree is that this single disk read is used to return approximate results in a ranked order, rather than distance order. Having a ranked result list has three major consequences. First, since no distance calculations are required, little CPU cost is incurred, even for large collections. Second, the descriptors themselves need not be stored within the leaf nodes, making it possible to store many descriptor identifiers in a single leaf node, which increases the likelihood of having actual neighbors in that leaf. The redundancy introduced with overlapping partitions further increases that likelihood. Third, as the results are based on a projection to a single line, false positives do arise when processing a leaf node. Since distances can not be calculated, other means of removing false positives are required.

3.2 NV-tree Implementation Overview

One NV-tree is stored in three different files: 1) the *line pool file*, which stores the details of each random line created for the tree; 2) the *in-memory file*, which stores the hierarchy of inner nodes that is kept in memory during query processing; and 3) the *leaf file*, which stores all the leaves of the NV-tree.

The NV-tree is written in C++. Upon invocation, the NV-tree server first reads the line pool file and the in-memory file, and opens the leaf node file. At that point, it can receive requests for searches and insertions. During insertions, the server also takes care of the maintenance of the files; insertions are discussed in Chapter 5.

In the remainder of this section, we give a high-level description of the implementation of the NV-tree. The description focuses on the *Unbalanced* partitioning strategy, which requires the more complicated implementation. We first outline the index creation process. Then we describe the data structures used for storing intermediate nodes and leaf nodes, respectively.

3.2.1 NV-tree Creation

As described in Section 3.1.1, the NV-tree is constructed via repeated applications of projection and partitioning. During the NV-tree creation process, the descriptor collection is first sampled to create the initial projection line, as described in Section 3.1.3. The collection is then sampled yet again, using a larger sample, to determine approximate values for the α, m, and σ parameters, described in 3.1.4 (this is done to avoid sorting the entire collection). Finally, the entire collection is scanned, and each descriptor is projected to the initial random line. The descriptor is then assigned to the appropriate (one or two) sub-partitions and written to temporary files for those sub-partitions. This whole process is then repeated for each of the temporary files in a depth-first manner. When a leaf partition is formed, the $(\textit{projected value}, \textit{descriptor identifier})$ pairs of the leaf partition are sorted in memory by their projected values, and written to the leaf node.

3.2.2 Intermediate Nodes

As mentioned above, the NV-tree is composed of a hierarchy of small intermediate nodes that eventually point to much larger leaf nodes. Each intermediate node contains four arrays:

- **Child:** This array points to child nodes of the intermediate node, including those child nodes created for overlapping sub-partitions. The child nodes may in turn be intermediate nodes or leaf nodes.
- **Partition Border:** This array keeps track of the upper and lower borders of each child node along the projection line. This array is used during insertions to guarantee that each descriptor is inserted into all relevant sub-partitions.
- **Search Border:** This array is used to direct the query descriptor search to the appropriate child node. This is done by using projection values that are half-way between the upper and lower partition borders of adjacent child nodes.
- **Projection Line:** As described in Section 3.1.3, the potential projection lines are kept in a line pool in memory. This array stores pointers into the line pool, which point to the projection lines of the child nodes.

Each intermediate node typically has a fan-out of 2–32, including the overlapping partitions. These intermediate nodes therefore require little space and can easily be kept in main memory.

3.2.3 Leaf Nodes

All leaf nodes are kept in a single large file on disk. Each leaf node is the size of a suitable I/O granule and contains $(\textit{projected value}, \textit{descriptor identifier})$ pairs. For efficiently finding the pair of the leaf, which has its projected value closest to the projection of the query descriptor, leaves are organized by the projected values in a sorted look-up table.

The leaf nodes can also be organized in a *sparse* manner, where fewer projected values are stored, and interpolation is used to find the "correct" location in the leaf. With the sparse organization, almost twice as many descriptor identifiers can fit into the leaf partition. This typically results in half the number of leaf nodes, and a correspondingly smaller index. The reduced space requirement comes at the potential cost of more inaccurate query results, as the exact position of descriptors along the projection line is not available. When evaluating this optimization, however, we observed next to no influence on the result quality. Our implementation therefore typically only stores every 16^{th} projected value; this setting is used throughout the whole thesis.

3.3 Experimental Setup

In this section, we describe the experimental environment used in our performance studies. First, we describe the descriptor collection and query workload used in all the experiments. Then we describe the result quality metrics studied in our analysis.

All experiments in this chapter were run on DELL PowerEdge 1850 machines, each equipped with two 3GHz Intel Pentium 4 processors, 2GB of DDR2-memory, 1MB CPU cache, and two (or more) 140GB 10Krpm SCSI disks. The machines run Gentoo Linux (2.6.7 kernel) and the ReiserFS file system.

3.3.1 Descriptors and Queries

In this study we use the well-known SIFT (Scale Invariant Feature Transform) method (Lowe, 1999, 2004), which is a standard method in the image processing community for extracting local features from images. The SIFT extraction process is performed over several scales of the image and finds interest points where the contrast changes significantly. Once the interest points have been identified, the signal around them is encoded into a 128 dimensional vector, which is normalized to a length of 512.

The descriptor collection was obtained by extracting local features from an archive of about 150 thousand images obtained with permission from the local newspaper Morgunblaðið (www.mbl.is). The images are largely high-quality press photos, which are highly varied in content. In order to reduce the number of descriptors extracted from each photo, the images were first resized such that their larger edge was 512 pixels. The resulting descriptor collection contained a total of 179,443,881 SIFT descriptors.

We have created query descriptors by modifying copies of images from the collection, following the approach of Lejsek et al. (2006), using the Stirmark benchmarking tool by Petitcolas et al. (2001). The image transformations include rotation, rescaling, cropping, affine distortions and convolution filters. SIFT descriptors cope rather well with most of these distortions at the image level (Lowe, 2004; Lejsek et al., 2006), meaning that a significant percentage of interest points are found in the same location as in the original image and that the corresponding descriptors are relatively close in the Euclidean space. But the transformations also include distortions which the SIFT descriptors have been shown not to handle well (Lejsek et al., 2006).

3.3.2 Result Quality Metrics

We place a strong emphasis on recall, for two main reasons. First, we expect only a few answers to each query, unlike more interactive applications. Second, large scale applications typically arise with local descriptors, where many descriptors provide evidence of matches. Such local descriptors can typically tolerate some false positives, as they are distributed randomly among all the data items. A small number of false positives is thus acceptable, but strong recall is imperative.

Computing result quality requires the definition of a *ground truth set* against which the results are compared. In the literature, one of two different approaches is typically used to define the ground truth set. The first approach is to run an exact k-nearest neighbor search for every query descriptor, leading to a result set of fixed cardinality, but with arbitrary distances. The second approach is to run an exact ϵ-range search for each query descriptor, which returns all neighbors within distance of ϵ from the query point, leading to a result set with a bounded distance, but of arbitrary cardinality. In both cases, an exhaustive sequential scan is typically used to ensure that the result lists defining the ground truth truly reflect the contents of the descriptor collection. The result quality of the indexing scheme in question can then be computed by comparing its results to these ground truth sets. Of course, both methods are highly sensitive to the choice of k or ϵ, respectively.

Results by Beyer et al. (1999) and Shaft and Ramakrishnan (2006), however, have shown that high dimensional data sets must exhibit some *contrast* to be indexable and to draw any meaningful conclusion from search results. In this work, contrast means that a nearest neighbor must be significantly closer to the query point than most other points in the dataset in order to be considered meaningful. In the absence of contrast, collections suffer from vanishing variance and instability of near neighbors, which preclude the construction of meaningful result sets.

A direct consequence of the theoretical analysis of Shaft and Ramakrishnan (2006) is that it is possible to construct a contrast-based ground truth set, against which indexing schemes can be compared. In order to construct such a set, a sequential scan may be used to determine, for each query descriptor, all the descriptors in the dataset that fulfill a given contrast criteria.

Using a contrast-based ground truth set has several theoretical benefits. First, the size of the ground truth set tends to be small compared to the k-nn approach, which collects (irrelevant) neighbors regardless of their distance from the query descriptor. Second, using the contrast-based ground truth alleviates the two typical problems that ϵ-range search faces. On one hand, when query points fall in very dense areas, very many vectors are returned using an ϵ-range query, although the results are hardly distinguishable from each other. On the other hand, when query points fall in sparse areas, no results may be returned using an ϵ-range query, while there may be many useful answers in the collection. Overall, therefore, building a ground truth based on contrast will allow more reliable result quality measurements.

According to Lowe (2004), computing SIFT over an image collection produces a contrasted set of descriptors. In his work, Lowe considered the nearest neighbor n_1 of a query descriptor q meaningful if and only if $d(n_2, q)/d(n_1, q) > 1.8$, where n_2 is the second nearest neighbor (Lowe, 2004). When the nearest neighbor passed the criteria threshold, then further checks were run to see whether n_1 was indeed a modified copy of the query descriptor. If the nearest neighbor did not pass the criteria threshold, then n_1 was rejected and no answer returned. Since, for many applications, a query may have more than one meaningful result, we adapt Lowe's criterion, by saying that returned neighbor n_i is meaningful with respect to contrast c (default value of c is 1.8) when $d(n_{100}, q)/d(n_i, q) > c$.

Note that in a result list ranked by distance, the constraint $d(n_{100}, q)/d(n_i, q) > c$ need only be checked for at most $i \in 1, \ldots, 99$; when the first neighbor is found that does not satisfy the criterion no more neighbors need checking. Also note that we can in fact generalize Lowe's criterion by saying that returned neighbor n_i is meaningful with respect

to contrast c up to rank j when $d(n_j, q)/d(n_i, q) > c$, where $j \geq 2$ and $i < j$. In our work we have found, however, that with j between 2 and 100, the number of descriptors passing the contrast criterion grows fast, while for $j > 100$ it grows slowly. We have therefore used $j = 100$ in the remainder of this analysis.

Using this contrast criterion, it is possible to build a ground truth set for an application. In the next section, we analyze the quality of these three approaches to generating the ground truth sets for our application from the results of a sequential scan.

3.4 Analysis of Ground Truth

The goal of this section is to analyze the properties of the query workload and descriptor collection and establish a meaningful ground truth set for our experimental studies. To that end, we have chosen 500,000 query descriptors at random from the workload described in Section 3.3.1. We have then run a sequential scan to calculate the 1,000 nearest neighbors for each query descriptor, yielding 500 million neighbors in all.

Note that the semantics of the copyright protection application, from which the workload is drawn, is such that for each query descriptor precisely one descriptor in the collection is a *correct match*, while the remainder should be considered *false matches*. In our collection, a total of 248,852 query descriptors found a correct match among the top 1,000 neighbors, or slightly less than 50%. While this may at first seem a low percentage, it is still a good recognition performance considering that some query descriptors originated from severely modified images (Lejsek et al., 2006). What we seek in this section is a general method for building a ground truth set, which includes a large number of the 248,852 correct matches and only a small number of false matches.

3.4.1 Ground Truth Based on k-NN

When taking a close look at the individual results we observed that the correct matches that appeared among the 1,000 nearest neighbors were in most cases ranked first in the result set. This indicates that by far the best choice for building a ground truth based on k nearest neighbors, would be by choosing $k = 1$. But even with $k = 1$, however, more than half of the neighbors in the ground truth set would be false matches. Furthermore, for many other applications, choosing a ground truth set of $k = 1$ would be too restrictive.

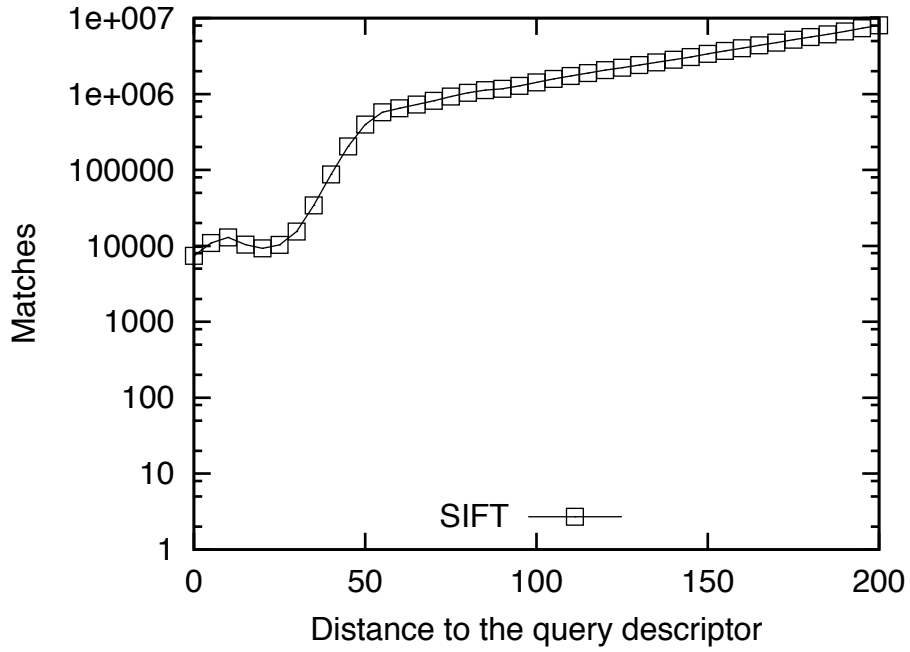

Figure 3.1: The distribution of all neighbors based on distance to the query descriptor.

3.4.2 Ground Truth Based on ϵ-Distance

We now analyze how the absolute distance between the query descriptor and returned neighbors affects the result quality. Figure 3.1 shows the distribution of all 500 million neighbors depending on the distance of each neighbor to the query descriptor. The x-axis shows the absolute distance (corresponding to varying ϵ), while the y-axis shows the number of neighbors with approximately that distance (the point at 0 corresponds to a distance of 0, while the point at 5 corresponds to the distance range $(0, 5]$, and so on). We observe that the number of descriptors stays rather uniform and small for short distances. Once the distance surpasses 25, however, we can see an exponential increase in the number of neighbors at each distance range (note the logarithmic scale). Recall that in our application almost all of these descriptors are false matches.

Figure 3.2, on the other hand, shows the cumulative distance distribution of the correct matches. In the figure, the x-axis is the distance from the correct match to the query descriptor, while the y-axis shows the cumulative fraction of correct matches found below that distance. From the figure we see that about two thirds of the correct matches can be found within an ϵ-distance of 100, and that within this distance they are rather uniformly distributed. The final third lies beyond a distance of 100, where the likelihood of finding further neighbors slowly becomes smaller; the last correct matches can actually be found at a distance of 370.

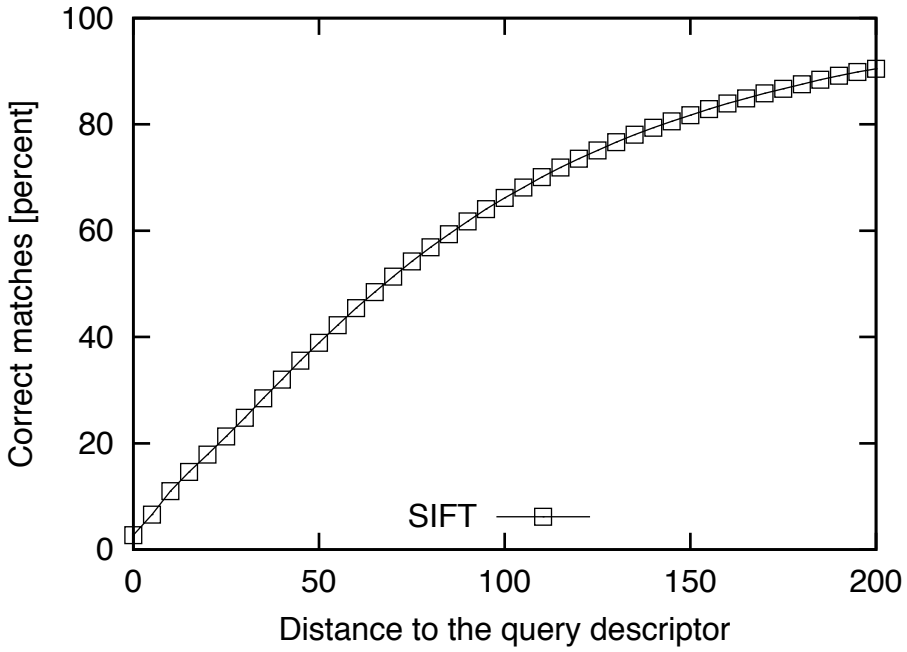

Figure 3.2: The cumulative distribution of correct matches based on distance to the query descriptor.

More importantly, however, Figure 3.2 shows that fewer than 20% of the correct matches are found at a distance smaller than 25, which is where the number of false matches started increasing exponentially. Thus, it is impossible to select a global ϵ for building a ground truth set which includes a large number of correct matches and only a small number of false matches.

3.4.3 Ground Truth Based on Contrast

Finally, we consider the effect of contrast on the quality of the ground truth set. Figure 3.3 shows an analysis of the correct matches based on different thresholds of the contrast criterion. The x-axis shows the contrast c, while the y-axis shows the percentage of correct matches with contrast higher than c, defined in Section 3.3.1 as $d(n_{100}, q)/d(n_i, q) > c$. The figure shows that 36% of the correct matches are more than five times closer in distance than the 100$^{\text{th}}$ nearest neighbor in the result list. For $c = 1.8$, which is the value that Lowe recommended, 186,290 out of 248,852 correct matches, or about 74.9%, pass the contrast threshold. About 20% of the correct matches have a contrast threshold lower than 1.5, and are therefore rather hard to detect from the false matches.[1]

[1] A small portion of the correct matches has contrast smaller than 1, which means that they were found at a rank higher than 100.

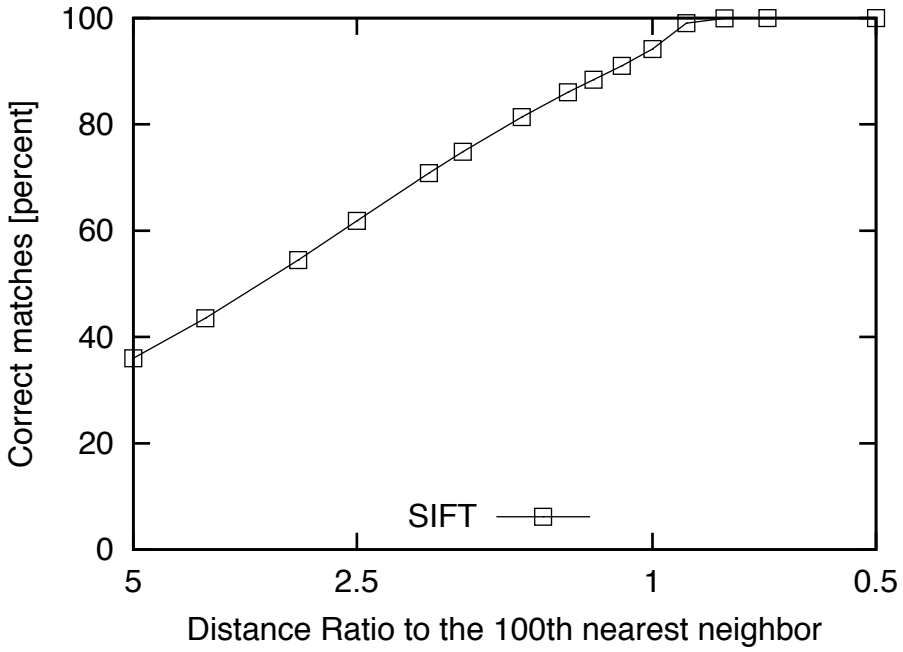

Figure 3.3: The cumulative distribution of correct matches based on contrast.

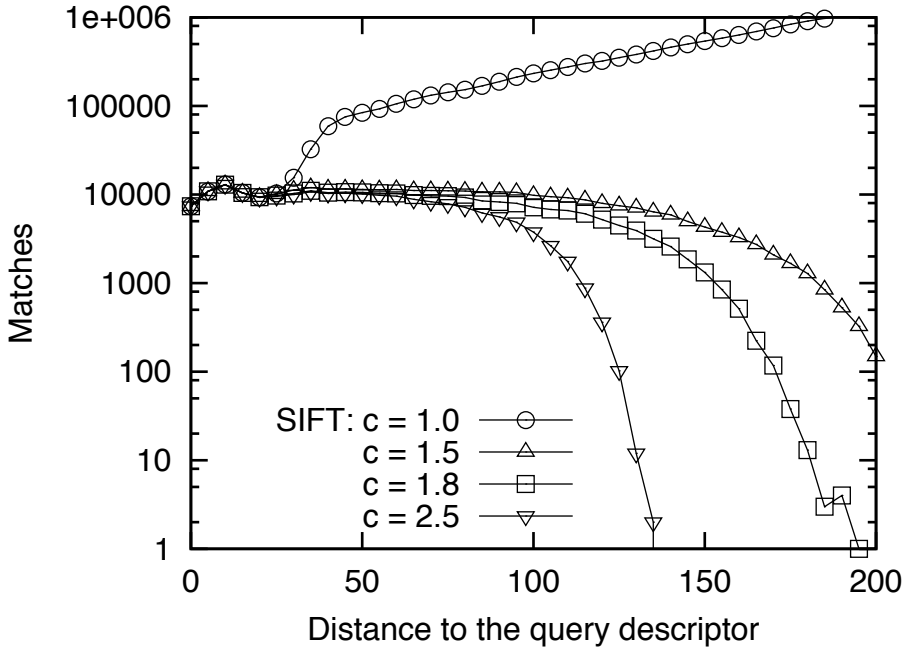

Figure 3.4: Distribution of neighbors passing the contrast criterion by distance to query descriptor, for various contrast thresholds.

Figure 3.4, on the other hand, shows the effects of the contrast criterion on the number of descriptors that pass the threshold filter (these include the correct matches). This time, the x-axis shows the absolute distance from the result descriptor to the query descriptor, while the y-axis shows the number of descriptors found at each distance. Overall, we

observe that a contrast threshold of $c = 1$ shows an exponential increase in the number of descriptors (similar to Figure 3.1, but at a smaller scale since at most 100 neighbors are considered), while all values of $c \geq 1.5$ avoid this behavior and show a well controlled number of descriptors; the higher the threshold, the fewer descriptors are returned.

Comparing Figures 3.3 and 3.4, we see that choosing a higher contrast threshold results in lower recall but fewer false matches, and vice versa. Comparing these to Figures 3.1 and 3.2, however, we see that any choice from 1.5 to 2.5 performs very well compared to the ϵ-based criterion. So the threshold of 1.8, proposed by Lowe, seems reasonable.

With the threshold $c = 1.8$, a total of 248,212 descriptors pass the contrast filter.[2] As described above, the number of descriptors that are both correct matches and pass the $c = 1.8$ contrast criterion is 186,290. Thus, about 75.1% of the descriptors in the contrast-based ground truth set are correct matches and about 24.9% are false matches.

3.4.4 Discussion

The analysis above shows that using a contrast-based criterion to construct the ground truth set is clearly preferable to using either k nearest neighbors or ϵ-distance, as using the contrast-based criterion yields the best ratio between correct matches and false matches (about 3:1 for $c = 1.8$). Furthermore, as described in Section 3.3.2, it is the only approach with solid theoretical underpinnings. As a result, we use contrast-based ground truth sets in the remainder of this thesis. We typically use $c = 1.8$ to build the ground truth set, but we also illustrate some results using c values ranging from 1.0 to 2.5.

Furthermore, the quality of the ground truth set is strong evidence that the distinctiveness of the SIFT descriptors holds even at large scale, which shows that we can expect small and meaningful result sets for nearly all query descriptors.

3.5 NV-tree Performance

In this section, we start by describing the NV-tree configurations used in the experiments. Then we present two experiments which analyze key properties of the NV-tree. In Section 3.5.2 we discuss an experiment with a single NV-tree index, which shows that the NV-tree yields high recall, especially with neighbors having high contrast. In Section 3.5.3 we then discuss an experiment with up to three NV-trees, which demonstrates that with

[2] The fact that this number is similar to the number of correct matches in the sequential scan results is purely by coincidence.

such configurations false positives can be largely eliminated, while keeping most of the high recall.

3.5.1 NV-tree Configuration

For all experiments reported in this section, we used the following NV-tree configuration:

- We have used leaf partitions consisting of six disk pages (24 KB). As leaf nodes are sparse (keeping a projected value for every 16^{th} descriptor identifier), a maximum of 5,579 descriptors identifiers can be stored per leaf. Leaf nodes are typically filled to 67% of capacity, leaving room for insertions. Recall that using sparse leaf nodes generally reduces the index size by half without affecting result quality. For all of the experiments, the in-memory hierarchy fits in less than 60MB of main memory.
- We used the *Approximate PCA* strategy to select the random lines. We generated an initial line pool of 1000 lines,[3] where each pair of lines has a minimum angle of 72 degrees. Starting with a very small sample from the partition (typically 0.01%), we select a set of 128 potential random lines. In each subsequent round, the sample size increases exponentially, while the set of potential lines decreases exponentially, until a single line is selected after three rounds. Approximate PCA generally increases recall by 10% over random lines.
- We used the *Hybrid* partitioning strategy with α set to 0.55. Before partitioning, a sample of about 5% of the points in a partition are used to determine m and σ (see Section 3.1.4). The *Hybrid* strategy generally yields about 5% higher recall than a *Balanced* strategy, but equivalent to the *Unbalanced*, while only requiring about 20% more space than the *Balanced* partitioning strategy.
- We retrieved 1,000 descriptors from each NV-tree (in one experiment we vary this number).

With this configuration, the index creation took less than 16 hours per NV-tree and one NV-tree requires about 50 GB of disk space (about twice the size of the collection). We created three NV-trees in total, as some experiments use two or three NV-trees.

The NV-tree search is almost exclusively I/O bound, as CPU time is typically 1–3% of the total query processing time. Furthermore, the NV-tree is designed such that a single

[3] We have experimented with line pools ranging from 64 lines to 4,000. Generally, retrieval quality increases slightly with line pool size, but so does index construction time. We have found 1,000 lines to be a good trade-off.

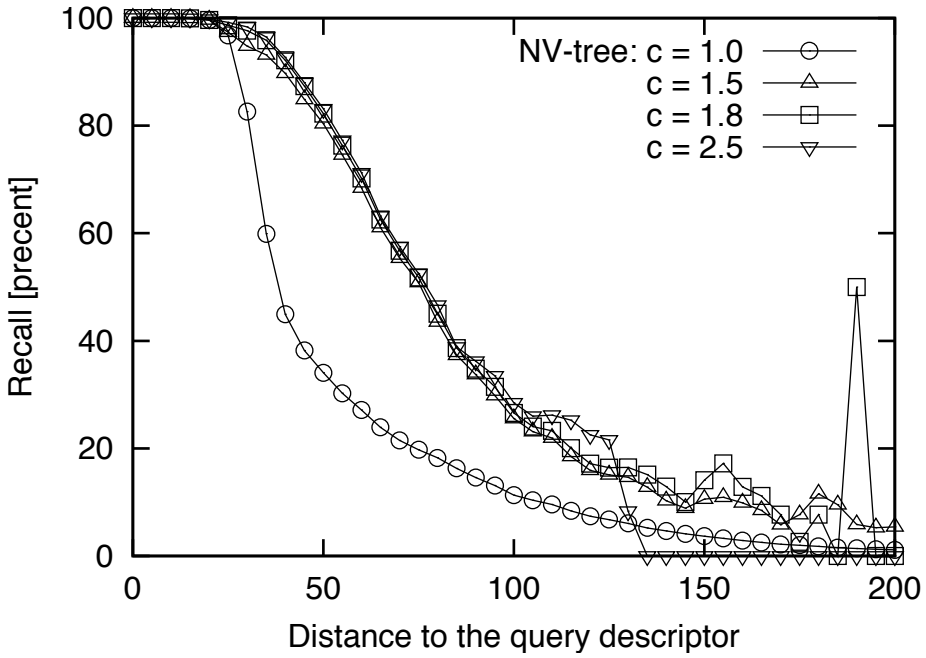

Figure 3.5: Recall for a single NV-tree retrieving 1000 Nearest Neighbors per query.

disk read is required for each tree. Therefore, the performance analysis focuses on index size, index creation time, and running time of the search. Note, however, that disk times are highly hardware dependent and may vary significantly based on the size and location of the index on disk, as well as how full the disk is, as we are using an off-the-shelf file system.

Nevertheless, one NV-tree needs about 12.5 milliseconds to return the 1,000 neighbors of a query descriptor, which is essentially the time required for a single random disk read. This can be contrasted with our highly optimized sequential scan, which takes 14 seconds per descriptor in a batch query process. When 3 NV-trees are used, the response time is about 38.5 ms.

3.5.2 Experiment 1: A Single NV-tree

In this experiment we ran the 500,000 queries and retrieved each time 1,000 nearest neighbors from a single NV-tree. We then used several contrast-based ground truth sets having different c values to compute recall and the number of false positives.

Recall

Figure 3.5 shows the recall of the search computed against four contrast-based ground truth sets for c ranging from 1 to 2.5. The x-axis shows the distance from the retrieved neighbors to the query descriptors, and the y-axis shows the fraction of meaningful neighbors returned for each distance category.

Consider first the ground truth set where $c = 1.0$. In this case, the 100 closest neighbors to the query descriptor form the answer. Overall, with this setting, descriptors which are closer to the query descriptor than 25 in distance are always found. For larger distances, the recall drops significantly. Recalling, however, the corresponding line from Figure 3.4 where the number of neighbors for $c = 1.0$ is increasing exponentially, then the reason for such a strong decline for distance larger than 25 is rather obvious; as very many neighbors are returned, the meaningful ones become only a small fraction.

Turning to the other recall lines when using ground truth sets having $c \in \{1.5, 1.8, 2.5\}$, we observe that the recall is much higher. While recall is still near-perfect only for distances smaller than 25, the recall is significantly higher in the range 25–100. Turning back to Figure 3.2 which showed that about two thirds of the meaningful neighbors are found at a distance closer than 100, this tells us that the single NV-tree is finding most of the meaningful neighbors and, in fact, the NV-tree is able to find 65.8% of all meaningful neighbors.

Interestingly, varying the contrast threshold between 1.5 and 2.5 does not affect quality in the range from 0 to 100, because the NV-tree copes very well with contrasted data and finds most of the meaningful neighbors. Two interesting effects are worth noting when the distance goes beyond 100, however. First, the fluctuations in that range are due to the small number of neighbors. Second, we observe that using $c = 2.5$, no neighbors are found beyond a distance of 130; at that point the other descriptors are not far enough to allow any descriptors to pass this threshold. A similar effect occurs with $c = 1.8$ at a distance of about 180. In the remainder of our experiments we use the ground truth set defined by $c = 1.8$, as proposed by Lowe.

False Positives

The NV-tree index performs approximate searches. Given that the ground truth set of descriptors that passes the contrast criterion is quite small as we have observed, most of the returned neighbors are indeed false positives. Since the NV-tree does not store the actual descriptor (it stores only its identifier) and retrieving the descriptor from disk to

compute distances is infeasible in practice, there are no means to filter out these false positives using a single NV-tree.

In general, some applications may tolerate false positives while others, such as applications with strong precision constraints, may not. Requesting only a handful of nearest neighbors from a single NV-tree tends to reduce the number of false positives, but it affects recall quite significantly (not shown). On the other hand, it is possible to reduce the number of false positives by using more than one NV-tree; this is the topic of the next experiment.

3.5.3 Experiment 2: Additional NV-trees

In this section we study the result quality obtained by using two or three NV-trees together to yield nearest neighbors. Take first the case of two indices. A technique called median rank aggregation (Fagin et al., 2003) can be used to combine the two ranked lists from the two indices. Median rank aggregation essentially traverses both ranked lists and outputs as the nearest neighbor the first descriptor seen in both lists, as the second neighbor the second descriptor seen in both lists, and so on. When three indices are used, we can either return as the nearest neighbor the first descriptor seen in any two indices, or in all three. These three strategies are called 2/2, 2/3 and 3/3, respectively, where a/b means that b indices are used and the first descriptor to be seen in a of those is returned as the nearest neighbor; in all cases we discard descriptors seen in fewer than a indices. In this terminology a single index is 1/1. We first study briefly retrieval performance and recall, and then focus on false positives.

Performance

The query response time (not shown) is proportional to the number of indices used; using a single index took 12.5 ms while using three indices took about 38.5 ms.

Recall

Figure 3.6 shows the recall of the four strategies considered (1/1, 2/2, 2/3, and 3/3). For this experiment the partition fetched by the search for each NV-tree was entirely processed, yielding as many descriptors as possible for each configuration. As the figure shows, the overall shape of the recall curves is similar when using more indices, The 2/2

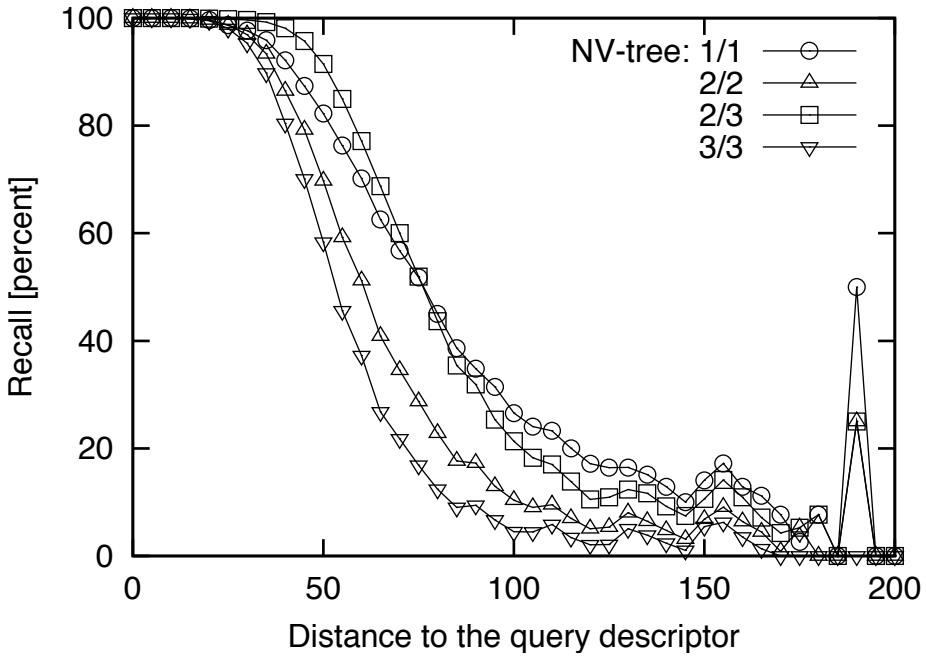

Figure 3.6: Recall by aggregating the result lists of two or three independent NV-trees. $c = 1.8$.

and 3/3 strategies always perform worse than 1/1. This is because some relevant descriptors may, by chance, miss one of the two or three necessary partitions, and thus not be considered part of the answer.

Turning to the 2/3 strategy, we see that for descriptors with short distances, it performs better than 1/1. This is due to the fact that these relatively close descriptors are more likely to be found in two corresponding partitions of three possible, than in the single correct partition of a single index. For descriptors that are farther away the tables turn, however, as then it is difficult for those descriptors to land in two corresponding partitions. Overall, however, the 2/3 strategy has slightly higher recall than the 1/1 strategy; in the following we therefore focus on the 2/3 strategy.

False Positives

The major motivation for searching more than one NV-tree, however, was not to obtain higher recall but to reduce the number of false positives. The overall strategy used for this purpose is as follows: Each of the three NV-trees is probed to yield a (ranked) result set of a specific size. Then these results sets are traversed to yield nearest neighbors to the query descriptor as described above. This time, however, only a few such "aggregated" neighbors are retrieved; we even consider retrieving a single such neighbor. The expec-

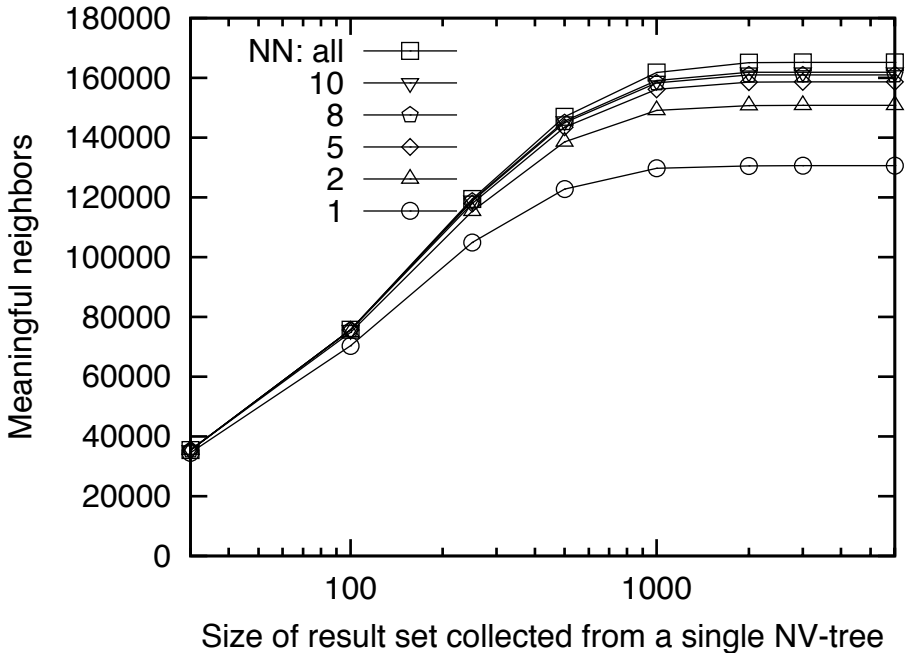

Figure 3.7: Meaningful Neighbors of 2/3 NV-trees based on number of neighbors retrieved.

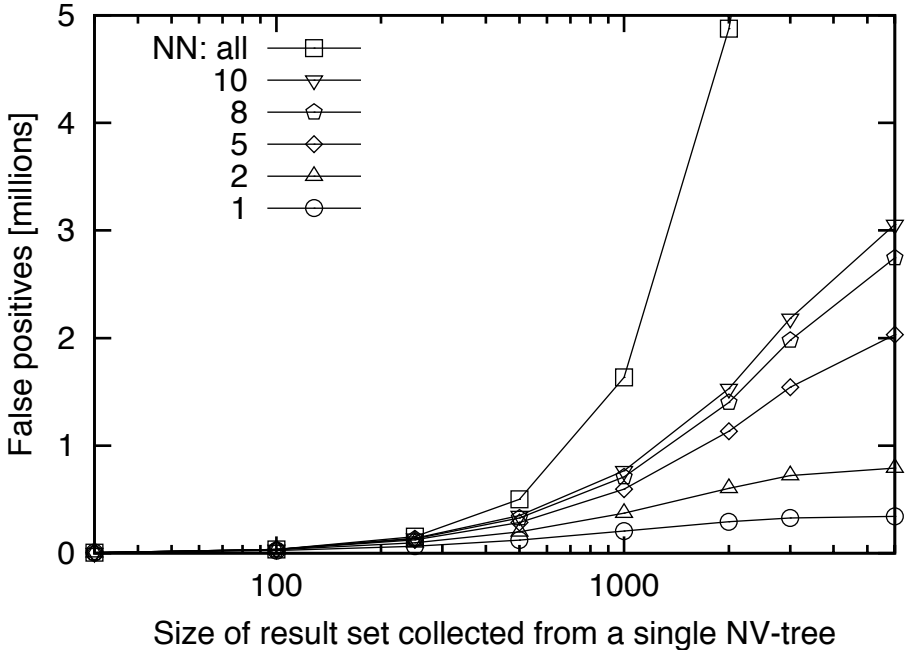

Figure 3.8: False Positives of 2/3 NV-trees based on number of neighbors retrieved.

tation is that these aggregated nearest neighbors will be very meaningful, as they appear close to the query point in at least two NV-trees; thus we expect to retain the high recall, while removing most false positives.

Figure 3.7 shows the recall of this approach. The x-axis shows the size of the result set obtained from each index. Each line of the figure shows the recall for a given number of aggregated nearest neighbors; recall that the sequential scan returns 248,212 neighbors. Considering first the overall shape of the lines, we see that, as expected, small result sets give low recall. When larger result sets are collected as input to the rank aggregation, however, recall improves. Beyond retrieving 1,000 descriptor identifiers from each index, the recall curve becomes flat and result sets over 2,000 descriptor identifiers show next to no recall gains.

Turning to the effects of retrieving additional aggregated nearest neighbors in Figure 3.7, we see that recall is improved significantly when going from one to two aggregated neighbors. By returning just one aggregated neighbor, we obtain a very reasonable recall of more than 130,000 meaningful neighbors (out of the 248,212). By returning two neighbors, recall improves to over 155,000 and with 10 aggregated neighbors we reach over 160,000 meaningful neighbors. Larger results sets achieve only minor improvements, but as we will see in a moment they increase the number of false positives significantly.

Figure 3.8 shows the number of false positives for the same experiment. As before, the x-axis shows the size of the result set obtained from each index, and each line of the figure shows the false positives returned for a given number of aggregated nearest neighbors. Overall the figure shows that as the result set size grows and as more aggregated nearest neighbors are returned, the number of false positives returned grows very sharply. About 15% of all queries return more than 10 nearest neighbors, and 2.5% even more than 100 neighbors; these query descriptor are clearly landing in very dense areas. Note that, in comparison, the number of false positives returned by a sequential scan ranges from about 15 million when 30 nearest neighbors are returned to about 3 billion when 6,000 nearest neighbors are returned.

Combining the results shown in Figures 3.7 and 3.8, we see that returning a result set of 1,000 descriptor identifiers from each index is necessary for recall, but we should limit the number of aggregated nearest neighbors returned very significantly, in order to limit the number of false positives.

Finally we briefly discuss the 2/2 and 3/3 configurations. As already shown they yield lower recall, about 136,000 and 119,500 descriptors, respectively. On the other hand, with these configurations, the false positives drop by another order of magnitude. For the 3/3 setup collecting a maximum of five neighbors at a result set size of 1000 gives only 16,000 false positives with a recall of 119,500 meaningful neighbors. Therefore, if false positives must be reduced at all costs, then this setup is the right choice.

3.5.4 Discussion

Overall, these experiments show that the NV-tree is a very good data structure for approximate nearest neighbor search in high dimensional space. This is because the construction of the NV-tree essentially respects the local contrast of the descriptor collection and encodes it into the partitions of the indexes.

In general, the NV-tree returns more than 99% of the meaningful neighbors that are found below a distance of 25. For neighbors beyond this distance the detection rate drops significantly, but overall about two thirds of the meaningful neighbors are found. Using a single NV-tree, the returned results have high recall, but contain a high number of false positives. By combining two or three NV-trees, those false positives can largely be eliminated while retaining the high recall.

3.6 Comparison to Related Work

The two major data structures most related to the NV-tree are the Spill-tree (Liu et al., 2004; Liu, 2006; Liu et al., 2007) and Locality Sensitive Hashing (LSH) (Gionis et al., 1999; Datar et al., 2006). In this section, we first briefly compare the NV-tree to the Spill-tree, before focusing on LSH in the remainder of the section.

3.6.1 The Spill-tree

The Spill-tree is, like the NV-tree, based on repeated partitioning of the descriptor collection into overlapping partitions, and then using a similar search algorithm to process a single leaf node. It has significant differences, however.

Most importantly, the Spill-tree only partitions the data into two partially overlapping partitions at each level, resulting in a much taller tree which in turn leads to significantly larger disk space requirements. Additionally, the overlapping factor is globally defined and does not consider the distribution of the data points along the projection line. Since the projected high-dimensional data tends to produce a normal distribution on the line, intermediate splits are very likely to have large portions of the data in common, resulting in a very limited usefulness of those splits. In the worst case, when most of the data falls in both partitions, the authors recommend re-partitioning without any overlap and subsequently directing the search to both partitions (the guaranteed query processing time is sacrificed in this case). This approach is called a hybrid Spill-tree. While the higher

query processing time of the hybrid Spill-tree may be acceptable for small collections in a main-memory setting, the performance impact for large collections and/or disk-based settings can be significant. Finally, each Spill-tree leaf node contains a set of descriptors rather than a ranked list, leading to high storage consumption and expensive distance calculations.

In order to understand the space requirements of the Spill-tree, we have considered how it would deal with our collection of 180 million descriptors. Given the nature of high-dimensional projections, we expect average overlap to be about $66,7\%$. We also assume a node size of 6,000 descriptor identifiers; note that this is larger than the NV-tree nodes and leaves no space for insertions. In order to determine the depth of the Spill-tree, we must then solve the equation $180,000,000 \times 0.667^x = 6,000$, which yields $x = 25.5$. The Spill-tree would thus require 26 hierarchies, resulting in $180,000,000 \times (2 \times 0.667)^{26} = 300$ billion pointers to descriptors, requiring more than a terabyte of data just to store descriptor identifiers. If, as proposed, the actual descriptors are stored, the space requirements become larger by two orders of magnitude.

Since the Spill-tree clearly has orders of magnitude higher storage requirements and gives weaker query performance than the NV-tree in a disk-based setting, we do not consider it further.

3.6.2 Locality Sensitive Hashing

In the remainder of this section we focus on Locality Sensitive Hashing which we believe to be the most competitive method to our proposed NV-tree for very large collections. In the following, we first describe the algorithm behind Locality Sensitive Hashing. Then, we present our adaptation of LSH to a disk-based setting and explain how the various parameters affect performance and quality of the search. Subsequently, we compare LSH to the NV-tree, before concluding with a discussion.

Unlike most other nearest neighbor search methods, the algorithmic idea behind Locality Sensitive Hashing is not based on a tree structure, but on hashing the data points into buckets. The chosen hash functions are constructed so as to guarantee that very close points coincide in the same bucket with much higher likelihood than those far apart. LSH was first published for the binary Hamming space by Gionis et al. (1999) and then later

extended to l_p norm by Datar et al. (2006). Most of its applications, however, have used rather small collections which could easily fit in memory.[4]

One major benefit of LSH is the simplicity of its algorithmic idea. Each descriptor is projected onto a set of k random lines through the search space. The lines are partitioned into fixed sized intervals (determined by a radius r) and each of the intervals is named by a symbol. Projecting to k lines gives k symbols, which are then concatenated to a word of length k. These words are built over an alphabet, whose cardinality is defined by the number of partition intervals, and form a kind of locality sensitive fingerprint. The smaller the radius r is chosen, the more intervals are created and hence the more symbols the alphabet contains. Note, however, that the probability of individual symbols is very different, because the projected points are normally distributed along the projected line. Increasing the number of partitions on the projected lines increases the variety of words at a fixed size k, but also increases the chance that close descriptors generate a different fingerprint.

In order to efficiently search for descriptors, they are hashed via a standard hash function into a hash table. Since LSH does not apply overlapping and the likelihood of separating two close neighbors also increases with fingerprint length k, it needs several such hash tables (parameter L in LSH notation) to guarantee a certain probability in recall. With very large databases, however, each additional hash table causes one additional I/O, making these additional tables very costly.

During query processing with LSH, the query descriptor q needs to look up the appropriate buckets for all L hash tables. q is therefore projected to all k lines for each individual table and the result is concatenated to a k length fingerprint which then references the bucket in the hash table that must be read from disk. For all candidate descriptors referenced in this bucket, the LSH algorithm computes the precise distance between the descriptor and the query point q. When the given descriptor falls within the selected ϵ-distance (the radius r) it is included in the result set; otherwise it is dismissed. After all L hash tables have been looked-up this way, all descriptors in the result set are sorted according to their distances to q and returned.

[4] A disk-based strategy was developed by Ke, Sukthankar, and Huston (2004). Since it was only tested on a small collection which was easily buffered in memory, it cannot be taken as a conclusive disk-based evaluation of LSH.

3.6.3 Adapting LSH to Disk

In order to run LSH in our context, it would have been necessary to keep not only the indices of the hash tables in main memory but also the whole descriptor collection as actual distances need to be computed. As our descriptor collection consumes about 22 GB, this approach is impossible. Furthermore, keeping the collection on disk and performing a random disk read to fetch each descriptor in the result is also unacceptable.

For our experimental evaluation we adapted the original LSH implementation (Andoni & Indyk, 2005) to disk, using a standard sorting library. In the interest of a fair comparison between the NV-tree and LSH we do not compare the running times of the search, since the NV-tree executable is very well tuned and we did not wish to spend the same time on optimizing the LSH algorithm. We can, however, make a fair comparison by simply counting disk reads.

The settings recommended for memory-based LSH create a very large number of hash tables. In order to make LSH more competitive to the NV-tree, we have studied the result quality of LSH with relatively few hash tables. In the remainder of this section, we therefore take a closer look at how to tune the quality of LSH in the context of very few hash tables. Since the parameters k, L, and r, as well as the cardinality of the result set, are strongly dependent on each other, we split our evaluation into two experiments. First we set the number of hashtables to $L = 3$ and vary the word-size parameter k from 6 to 12 (adjusting the radius r accordingly). In the second experiment we take the most suitable configuration of the former experiment and evaluate the quality when varying the number of hash tables (effectively varying the number of disk reads required for the search).

Figure 3.9 show the distribution of the result set size for the 500,000 queries, using LSH with three hash tables. LSH does not give any guarantee on how many neighbors are returned, so when increasing the fingerprint size k we need to shrink the radius r correspondingly in order to keep the average number of nearest neighbors at several hundreds. The x-axis shows the quantiles of the distribution, while the y-axis shows the result size set at each quantile. The figure shows that by reducing fingerprint size k and radius r the cardinality of the result sets grows slightly but becomes more stable. Longer fingerprints and larger radius generally yield fewer neighbors, but have the drawback that for 5–10% of the results the answer set grows extremely large. The LSH setup with $k = 6$ and $r = 25$ returns on average 1,305 neighbors, but in the worst case 10,572 nearest neighbors. The setup with $k = 12$ and $r = 80$, on the other hand, returns on average on 445 neighbors, but can return as many as 83,041.

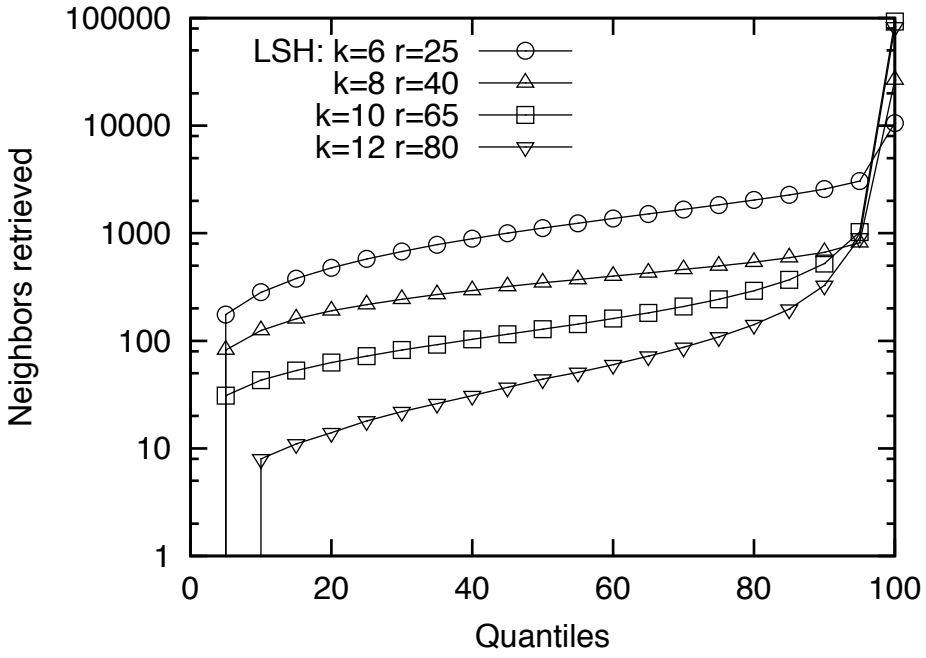

Figure 3.9: Distribution of result set size for LSH with three hash tables ($L = 3$).

The setup with $k = 10$ and $r = 65$ was chosen in the continuation, and the number of hash tables L was varied. Figure 3.10 shows that the increase in nearest neighbors is roughly linear for most of the quantiles. The largest result sets are proportionally smaller, because of the existence of duplicates and because it is unlikely that many hash tables yield very large buckets.

The major benefit of LSH over the NV-tree is the size of the index, which is due primarily to the overlapping partitions of the NV-tree. LSH needs three integers per hash table entry: one for numbering the hash bucket; one as a control hash; and finally the descriptor identifier. Since the hash bucket number is only used for sorting the table on disk, it can be removed afterwards, resulting in 8 bytes per descriptor on disk. With sparse leaf nodes, on the other hand, the NV-tree only stores a little over 4 bytes per descriptor. Due to the non-overlapping nature of LSH, however, each hash table requires only about 2.1 GB of disk space, which is significantly lower than the storage needed for a single NV-tree.

3.6.4 Recall of LSH

We now turn to a comparison of the LSH and NV-tree data structures. Figure 3.11 shows a comparison of the recall of three LSH hash tables to a single NV-tree. As the figure shows, with this setting, LSH yields significantly lower recall than that provided by the NV-tree. LSH has, on the other hand, the desirable property that it retrieves in most

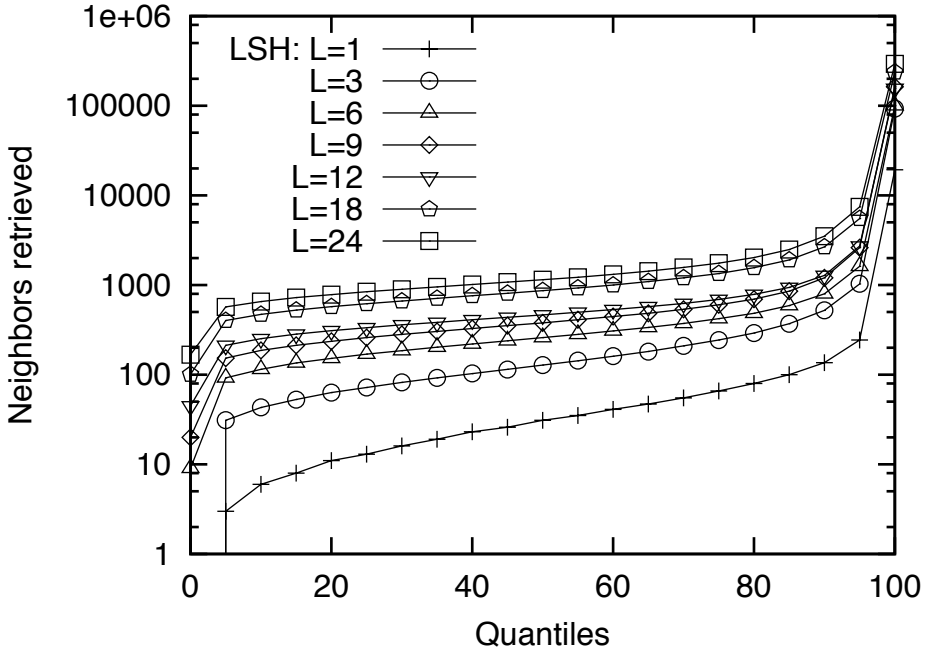

Figure 3.10: Distribution of result set size for LSH with varying number of hash tables ($k = 10, r = 65$).

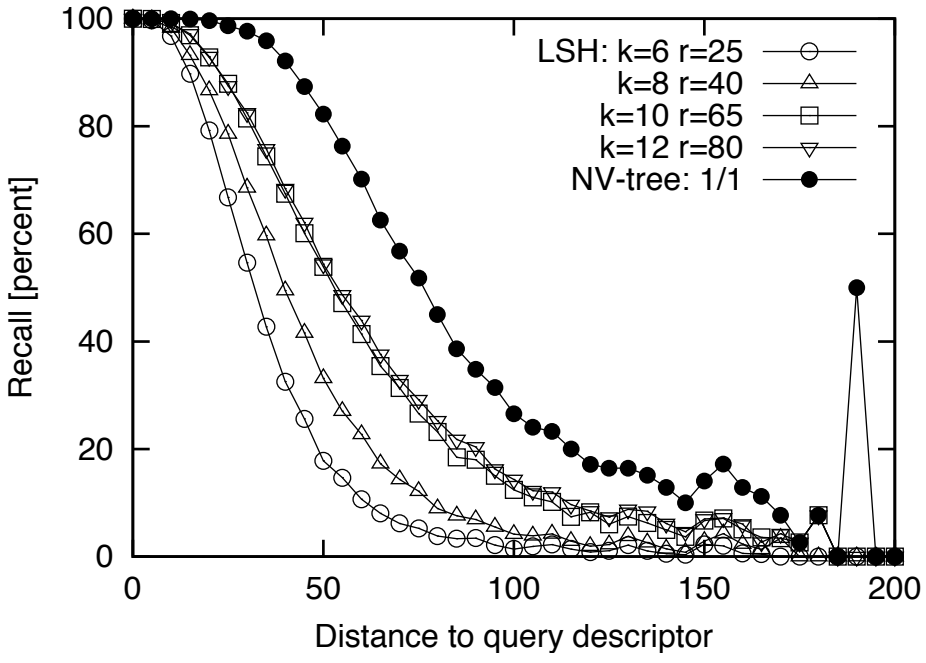

Figure 3.11: Recall for different LSH setups (varying word size and radius) with three hash tables ($L = 3$).

cases a significantly lower number of false positives (not shown). Finally, we point out that LSH makes no distinction between low and high contrast, as it is an ϵ-approximate

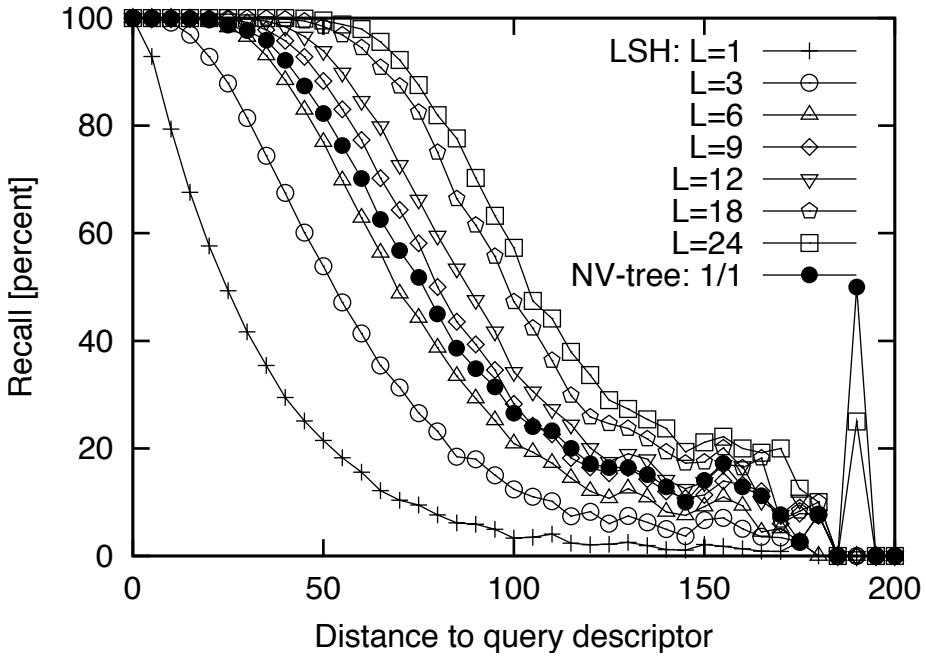

Figure 3.12: Recall for LSH with varying number of tables ($k = 10$, $r = 65$).

search. This was already known from the design of LSH, but we have observed this fact in our evaluation.

Furthermore, Figure 3.11 shows that a large radius r combined with larger k returns better results. This effect levels off, however, once the radius gets too large, because normal distribution and large symbol buckets along the lines make certain symbols appear much more frequently than others. Therefore, the LSH configuration with $k = 12$ and $r = 80$ gives only minor improvements over $k = 10$ and $r = 65$.

Figure 3.12 compares the recall of LSH with varying number of hash tables ($k = 10, r = 65$), to that of a single NV-tree index. The figure shows that by increasing the number of LSH hash tables, the recall quality improves steadily. Note, however, that this improved quality comes at the cost of extra disk reads, and that those disk reads are not of a fixed size and might in some cases go beyond the I/O granularity of today's hard drives, which is typically 128KB. Furthermore, it is well known that both small and large disk reads are more costly than reads of an optimal size. Combining the cost of each read with the number of disk reads, we see that LSH has a much higher response time.

Figure 3.12 shows that the point where LSH outperforms the NV-tree lies roughly at $L = 8$ hash tables, so we can say that the NV-tree can deliver the equivalent recall quality with single disk read that LSH can with eight disk reads. The average number of false

positives for $L = 9$ hash tables is 1,201, so we can also say that here the NV-tree and LSH yield the same "performance".

3.6.5 Filtering False Positives

As we did for the NV-tree, it is also possible to filter false positives from the LSH results. In this case we need to aggregate the result sets of the individual LSH hash tables. As explained in Section 3.6.3, adapting LSH to disk precludes any actual distance calculations, and therefore filtering false positives based on distances is impossible. Furthermore, a rank based approach cannot be used, as the buckets are essentially sets which have no internal ranking. Instead, we have taken the approach used by Baluja and Covell (2006) and filter false positives by simply counting the number of occurrences of each descriptor in the result sets from all the hash tables and ranking the result accordingly. Close neighbors are likely to be found by many hash functions, and their occurrence count will therefore be high. Then, we take a fixed number of neighbors from this ranked list and declare these as the aggregated nearest neighbors.

Figure 3.13 shows the recall of this method. As the figure shows, LSH gives high recall with this method when we have a large enough number of tables to provide a distinguishable ranking among the aggregated result sets. As the figure furthermore shows, however, LSH only manages to catch up with a three-index NV-tree setup once we collect neighbor sets from 24 different LSH hash tables. Again, this is a ratio of 1:8 in favor of the NV-tree.

Looking further at the false positives shown in Figure 3.14, we see no significant differences when using more LSH hash tables. In contrast to the NV-tree, it is completely dependent on the number of nearest neighbors, as LSH practically guarantees with very high probability very large results sets for all queries. The generation of a small and meaningful answer set is then just a matter of ranking the neighbors.

3.6.6 Discussion

As we have seen in the experiments, LSH and NV-tree can give similar quality for nearest neighbor search in high dimensional space. In order to provide a fair comparison of both methods we have put emphasis on choosing a sound selection of the parameters for both techniques. The results show that the NV-tree trades off disk space for the benefit of better query performance while LSH trades off search time for a smaller index on disk.

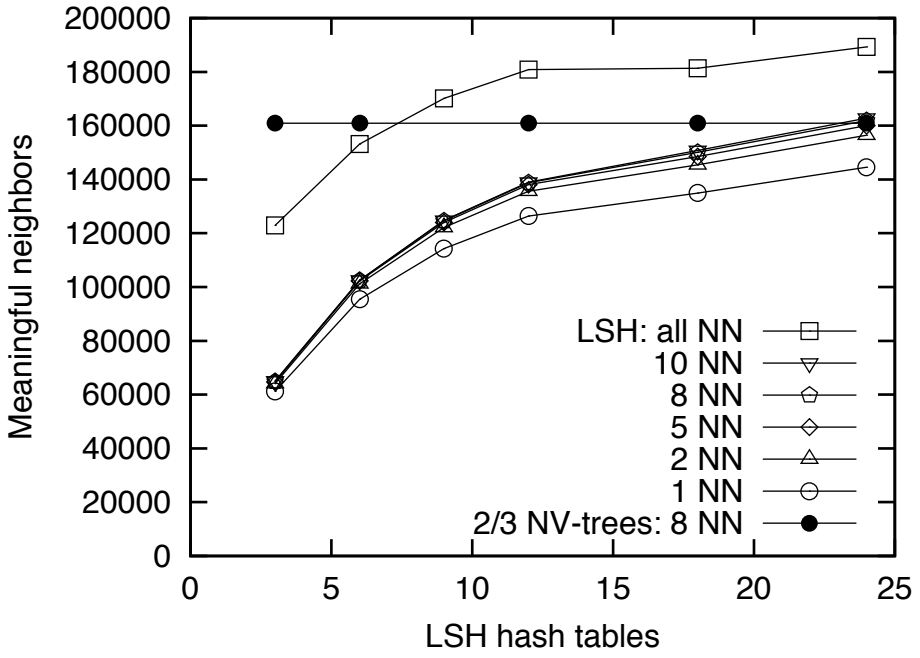

Figure 3.13: Meaningful neighbors for the NV-tree and LSH ($k = 10$, $r = 65$).

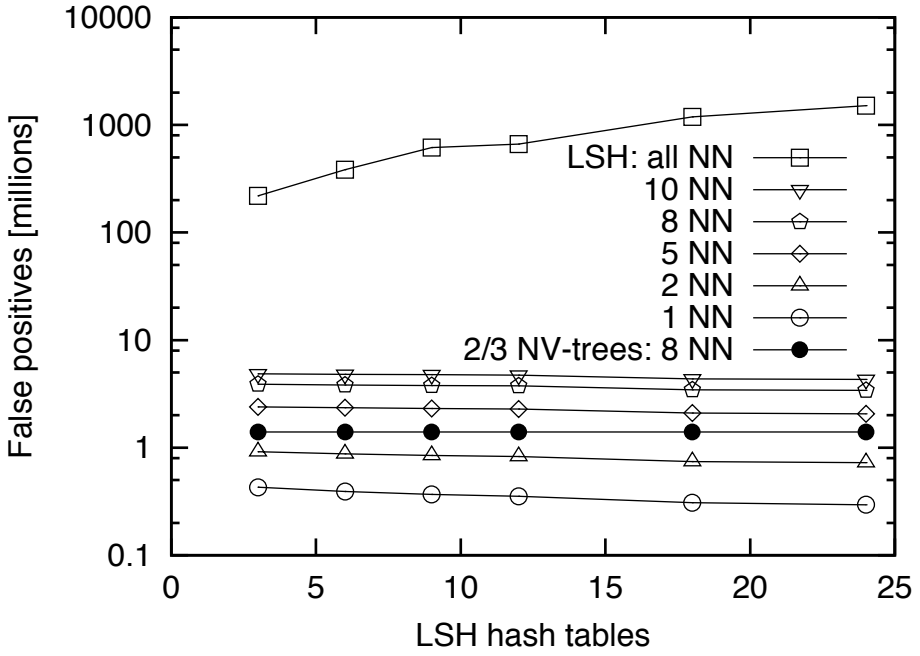

Figure 3.14: False positives for the NV-tree and LSH ($k = 10$, $r = 65$).

When false positives are tolerated, the NV-tree is about 8 times faster, but uses 50 GB of disk space vs. 8×2.1 GB $= 16.8$ GB for LSH, or 3 times more disk space. The same trade off can be seen when we filter as many false positives as possible, as then the NV-

tree needs three disk reads from 150 GB of disk space while LSH needs about 24 disk reads from 50.4 GB of disk space.

One of the clear benefits of the NV-tree is that it always loads fixed sized partitions from disk, while the number of descriptor identifiers in a single LSH hashtable bucket can be very large. This behavior may lead to unpredictably large result sets of almost 100,000 neighbors for our setup or unpredictably small result sets, which in turn leads to unpredictable I/O sizes.

The main drawback of the NV-tree is the space requirements due to the redundancy created by overlapping partitions. It is unclear whether the advantage of the overlapping NV-tree over LSH holds for very large collections, as the growth of the NV-tree could negate any advantages of main memory buffering. The space requirements for LSH at large scale, on the other hand, are less predictable: more hash tables will likely be needed for result quality, but how many more is unclear. As we will see in the next chapter, however, the redundancy within the NV-tree can be successfully eliminated without losing result quality; with that modification, each NV-tree is smaller than an LSH hash table and there are fewer NV-trees than LSH hash tables, making the NV-tree clearly the superior approach.

3.7 Summary

In this chapter we have proposed theNV-tree, which is a disk-based data structure that gives good approximate answers with a *single random disk read*, even for very large collections of high-dimensional data. Furthermore, searching theNV-tree incurs negligible CPU overhead, making it suitable also for main-memory based processing. We have described the fundamentals of the NV-tree, as well as different strategies for its construction.

We have then analyzed the properties of a large-scale copy detection application using the well known SIFT descriptors. We show that the SIFT descriptors are very distinctive and have high contrast, which is necessary for large-scale applications. Furthermore, we show that using contrast-based ground truth sets is necessary to obtain meaningful results for all queries. We have shown that the NV-tree returns very good approximate results for this workload, and we believe that the NV-tree can be used for any large-scale application, where the data set can be shown to have contrast and yield meaningful results.

Finally, we have shown that the NV-tree as well as LSH are two good indexing schemes for nearest neighbor search in high dimensional space. While both methods are built on

the concepts of projection to lines and partitioning, however, they have very different properties. The NV-tree is a tree-structure which guarantees fixed size I/O operations and a maximum size on the result set. LSH is hashing based and might in extreme cases return very large result sets. The overlapping NV-tree trades off disk space for the benefit of fewer disk reads during the search, while LSH focuses on rather small index sizes, but needs more accesses to disk during the search process. In the next chapter, we address the space requirements of the overlapping NV-tree, thus eliminating its main weakness.

Chapter 4

Non-Overlapping NV-tree

The original proposal of the NV-tree, presented in the previous chapter, included significant overlap in the index to compensate for partitioning problems in the high-dimensional space. The main contribution of this chapter is an enhanced version of the NV-tree, which is entirely free of overlap. Simulation results by Ólafsson, Jónsson, Amsaleg, and Lejsek (2011), focusing on the balanced NV-tree, strongly indicated that the overlap could be removed by using more than one index and merging the results from the individual indices. As each non-overlapping index is much smaller, the overall performance is improved. This chapter extends that work to the unbalanced NV-tree and makes the following major contributions:

- First, we analyze the performance implications of the redundancy caused by overlapping partitions and show that it is simply necessary to remove the overlap.
- Second, while removing the overlap does reduce result quality, we propose to use three different strategies to "re-capture" the result quality, which more than compensate for the losses due to lack of redundancy.
- Third, we present a performance study which compares the new "overlap-free" NV-tree with previous results, showing that although more non-overlapping NV-trees are required for retrieval quality, each index is so much smaller that retrieval is actually faster.
- Fourth, we present a second performance study, which shows that retrieval quality and performance are not affected significantly when the collection size grows to 2.5 billion descriptors.

Overall, our results show that the NV-tree is a scalable approximate indexing strategy, which yields results of acceptable quality. When the indexed data can be entirely kept in

main memory, then the NV-tree is extremely fast as very little computation is performed to find the k-nn. When the data collection to index is large, then the NV-tree gracefully adapts to efficient disk-based processing as a single disk access is required per query point.

This chapter is organized as follows. First, Section 4.1 details the negative impact of the redundancy on the index size and creation time, as well as on the search process. Section 4.2 then presents three strategies to improve the quality of the results returned by the NV-tree, which were degraded by the lack of redundancy. Section 4.3 compares both versions of the NV-tree and shows the overlap-free one outperforms the version with overlap. Then, Section 4.4 gives indications on the performance of the NV-tree when indexing a collection made of 2.5 billion SIFT descriptors. Section 4.5 then summarizes the chapter.

4.1 The Case Against Redundancy

The operlapping NV-tree uses overlapping partitions to achieve result quality with only a single disk read, thus adressing problems related to the curse of dimensionality. Redundant storage, however, impacts negatively the size of the index and its construction time, as well as the ability to use several trees to consolidate the results returned to the users. This section discusses these two negatives impacts and motivates the need to remove all redundancy from the NV-tree as the only viable way to hit the billion scale.

4.1.1 Index Size and Construction Time

It is possible to roughly estimate the size of an NV-tree index with redundancy using the following model. The depth of the tree can be computed by $d = log_f(\frac{n}{l})$, where n is the number of descriptors in the collection, l is the number of descriptors per leaf node (recall that leaf nodes only store descriptor identifiers) and f is the fan-out at each level. Due to the overlap between partitions, each descriptor is represented in multiple leaf nodes in the tree, yielding a *redundancy factor* of $r = (\frac{2f-1}{f})^d$.

For the experiments in the previous chapter, a collection of $n = 180$ million descriptors (22.2GB) was indexed with a fixed fan-out of $f = 5$ and an approximate node filling rate of 70% leading to approximately $l = 4036$ descriptors on average per leaf node. Then $d = log_5(\frac{180\times10^6}{4036}) = 6.65$ and the redundancy factor is therefore $r = (\frac{2\times5-1}{5})^{6.65} = 49.9$,

which leads to a total storage requirement of 180 million $\times$ 6 Bytes $\times$ 49.9 = 50.2GB, which is consistent with the reported storage requirements of about 50GB.

For a collection of 2.5 billion descriptors, however, the depth would increase to $d = 8.28$ and the redundancy factor to $r = 130.4$, which yields a final index size of 2.5 billion $\times$ 6 Bytes $\times$ 130.4 = 1.8TB. While such space requirements are still feasible with today's hard drive capacities (despite the fact that the index has grown 6 times larger than the actual descriptor data), the factor that makes such a setup intolerable is the index creation time. Given that index creation takes 15 hours for 180 million descriptors, an estimate of 24 days can be given for a 2.5 billion descriptor collection.

Completely removing the redundancy of overlapping partitions from the NV-tree creates an index of 13GB for a 2.5 billion descriptor collection in about 15 hours, which is 38 times less than the estimated construction time for a single overlapping NV-tree.

4.1.2 Searching in Multiple Trees

While the NV-tree is designed to adapt well to disk-based processing, its performance is best when the entire index is cached in main memory. Since results with good recall and a reduced number of false positives are returned when several NV-trees are used simultaneously, it is interesting to determine how many NV-trees can be cached in a given amount of RAM.

It is quite clear that the redundancy in the NV-tree is in this case a serious problem as each index is very large. For example, each index used in the previous chapter was about 50GB; no index could therefore be entirely cached, and a disk read was invariably needed per index to answer each query. In contrast, without redundancy, that same index would be about 1.1GB. It would easily fit in main memory, as well as a few additional NV-trees, all together returning high quality results very efficiently since no disk reads would be involved.

Figure 4.1 illustrates these tradeoffs between the overlapping NV-tree and the version without redundancy. The x-axis varies the collection size, while the y-axis shows the number of overlapping and non-overlapping indices that fit within 64GB of main memory. The figure clearly shows the exorbitant difference in index size between the two NV-tree types, due to the exponential growth of the overlapping NV-tree. While an overlapping NV-tree for a ten million descriptor collection is only 15 times as large the non-overlapping version, the ratio rises up to 53:1 for a 215 million descriptor collection (the largest overlapping NV-tree to fit in memory).

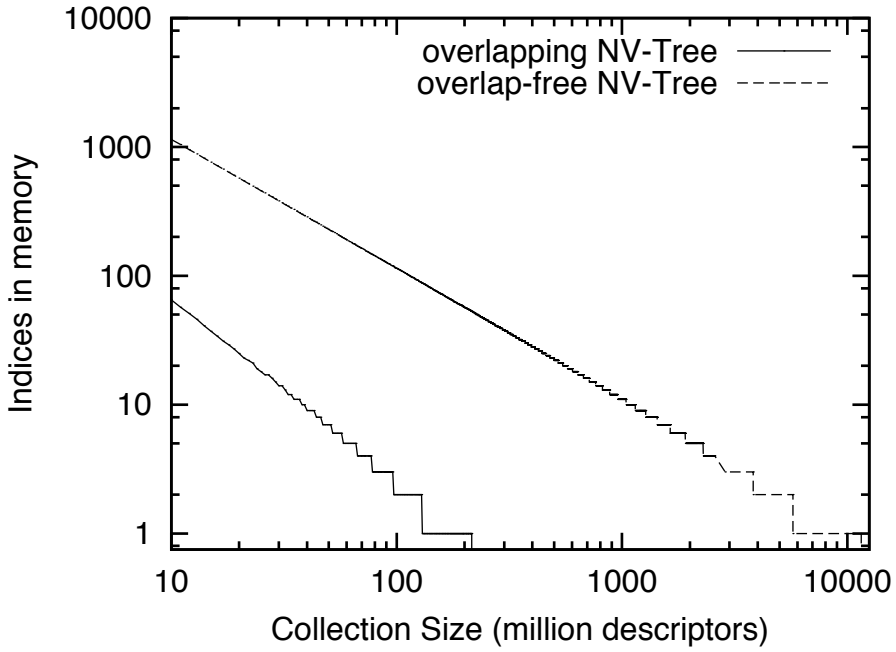

Figure 4.1: Comparison of overlapping and non-overlapping NV-trees that fit within 64GB.

Removing redundancy from the NV-tree is therefore also useful at search time, as it allows to use more indices simultaneously.

4.2 Overlap-Free NV-Tree

Removing overlapping from the NV-tree is the key to scaling up the indexed collections. Not surprisingly, the quality of the results drops with removal of the overlapping; it decreases from about 66% (see the previous chapter) to about 48% (reported in the experiments of Section 4.3).

This loss in quality does not affect neighbors that are close to the query point in the high-dimensional space (within a small ϵ threshold) and *well contrasted*, i.e., significantly closer than other data points. It does, however, affect neighbors that are well contrasted, but not close in terms of distance. As we wish to preserve the key advantage of the NV-tree over index structures based on ϵ thresholds, of being much less affected by the actual distance, we therefore propose to use the following three strategies to address this reduction in the result quality: (i) Creating additional NV-trees; (ii) Creating deeper NV-trees with smaller leaf nodes; and (iii) Reading additional leaf nodes.

Note that none of these strategies is particularly novel. It is their combined effect that is the novelty of this chapter, however, as they make the NV-tree truly scalable to very large collections. In fact, we show in Section 4.3 that the combined strategies improve recall beyond the original results.

4.2.1 Creating Additional Indices

Creating many indices built over the same data collection, querying them in parallel, and aggregating their results improves quality. Due to their reduced sizes, several overlap-free NV-tree are also likely to fit in main memory. When the collections are really big, however, then disk accesses are mandatory and the cost of retrievals will be linear with the number of indices. An appropriate operating point can thus be determined when trading quality against I/Os for applications with specific performance requirements.

4.2.2 Deeper Trees with Smaller Leaf Nodes

We ran extensive experiments to determine the best size for the leaf nodes. We found that one page leaf nodes (i.e., 4KB) provided the best recall and the least false positives. This can be explained by the additional projections and partitioning steps required to reduce the number of descriptors in such small leaf nodes, compared to leaf nodes of a larger size. These additional steps help to better capture the true neighborhood on the points in space. Creating trees with such small leaf nodes was not an option with overlapping NV-trees, however, since each additional level of the tree almost doubled its size.

4.2.3 Reading Additional Leaf Nodes

For many index structures, the approach taken to increase (or even guarantee) result quality is to descend to multiple leaf nodes and merge the results. A natural extension of that approach for the NV-tree would be to choose the two adjacent sub-partitions at each level in the tree, and to retrieve neighbors from all leaf nodes found in this manner. This approach, however, is not feasible for two reasons. First, it would violate the design criterion of having at most a single I/O per index; each additional leaf node would require accessing disks. Second, since there would be very many leaves and each leaf only contains ranking information, merging the results into a meaningful order would be difficult.

It is therefore infeasible to descend into the tree along multiple paths starting from the root of the tree. Instead, it is possible to consider multiple leaf nodes once the second lowest

level of the tree is reached. At that level, 6 parent nodes define 36 leaf nodes of 4KB each, filling the Linux 128KB I/O granule. When the search reaches that penultimate level, then it reads adjacent partitions in the two most relevant parent nodes to eventually fetch from disk up to four leaf nodes (among those 36) that can be read (with high likelihood) within a single I/O. In order to merge descriptors from different leaf nodes, we propose to assign a priority to each of the four leaf nodes. The priority is based on the distance from the projected value of the query descriptor to the center point of the partition.

4.3 Comparative Experiments

We have implemented and evaluated all three adaptations to the NV-tree data structure proposed above, and compared them to the performance results of the overlapping NV-tree. First, Section 4.3.1 gives an overview of the experimental setup. Then, Section 4.3.2 presents performance measurements of the index construction and query performance, while Section 4.3.3 analyses the result quality.

Note that since a) the overlapping NV-tree has already been shown in Chapter 3 to significantly outperform LSH in terms of the number of indices (NV-trees vs. LSH hash tables) and b) with our modifications each NV-tree is smaller than an LSH hash table, this comparison is sufficient.

4.3.1 Experimental Setup

We used the same workload as in Chapter 3, but repeat the key information here for completeness. The set of 179,443,881 128-dimensional SIFT descriptors was obtained by extracting local features from an archive of about 150,000 high-quality press photos. In order to evaluate the query performance of the NV-tree, we extracted 500,000 query descriptors from transformed versions of images from our collection. We then used the contrast-based ground truth defined in Chapter 3.

The experiments in this section were run on DELL PowerEdge 1850 machines running Gentoo Linux (2.6.7 kernel), each equipped with two 3GHz Intel Pentium 4 processors, 2GB of DDR2-memory, 1MB CPU cache, and two (or more) 140GB 10Krpm SCSI disks with the ReiserFS file system.

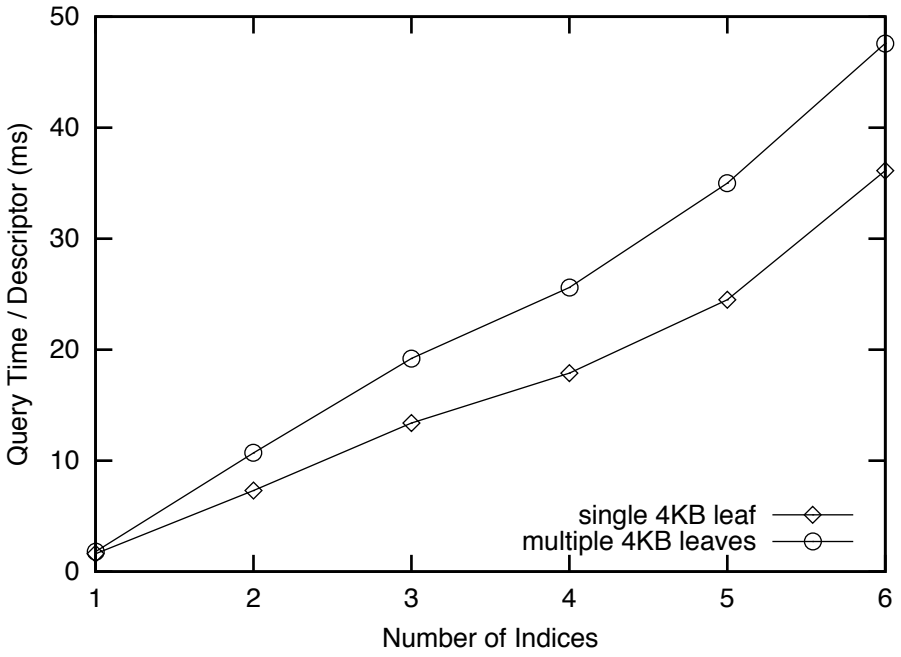

Figure 4.2: Retrieval time of NV-tree configurations for a collection of 180 million descriptors.

4.3.2 Indexing and Retrieval Performance

The major goal of the non-overlapping NV-tree is to reduce the index creation time. The creation time for a single non-overlapping NV-tree is about 2 hours, compared to more than 15 hours for the overlapping NV-tree (Lejsek, Ásmundsson, Jónsson, & Amsaleg, 2009). Furthermore, each non-overlapping tree consumes about 1 GB of disk space, compared to 50 GB for the overlapping NV-tree.

The retrieval time is also reduced significantly due to the smaller index size. Figure 4.2 shows the retrieval time for different configurations of the overlap-free NV-tree. The x-axis shows the number of indices used, while the y-axis shows the average time to retrieve the k nearest neighbors for each query descriptor. Since the main memory is 2 GB, some of which is used for the operating system, a single overlap-free NV-tree can be kept in memory, and the retrieval time is only 1.6 ms per query descriptor. The search time increases, however, when aggregating from more indices; it is 7.3 ms per descriptor for two indices, up to 37 ms for 6 indices.

For our disks, a random disk read takes about 12.5 ms, yielding an expected retrieval time of 75 ms for 6 indices. The retrieval time is reduced, however, since the small size of the indices facilitates both buffering and disk locality. In contrast, the retrieval time of the

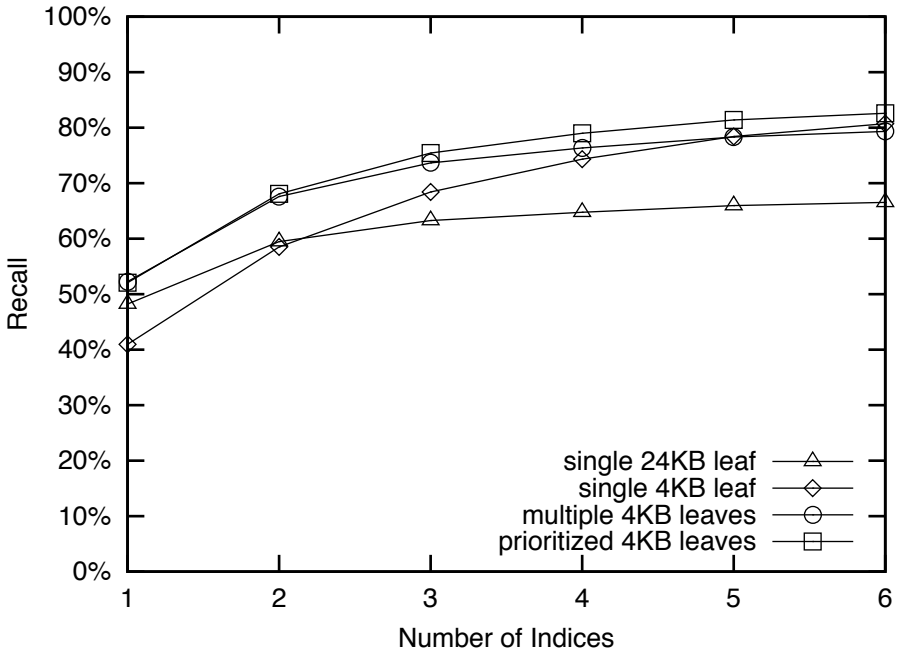

Figure 4.3: Recall of NV-tree configurations for a collection of 180 million descriptors.

much larger overlapping NV-tree does indeed grow by about 12.5 ms per index, resulting in 37.5 ms for three indices.

Finally, Figure 4.2 shows that the overhead of reading additional leaf partitions is about 40% (with or without priority assignment), despite the fact that all leaf partitions are designed to fall within a single disk read. The first reason is that one logical I/O may end up as two physical I/Os. The second reason is due to disk buffering, as the likelihood of finding a single leaf in buffers is significantly higher than that of finding a whole range of subsequent pages.

4.3.3 Result Quality

We now study the impact of the proposed techniques on result quality. Figure 4.3 shows the recall of the different NV-tree configurations. As before, the x-axis shows the number of indices used to answer queries, but the y-axis now shows the recall relative to the sequential scan.

Figure 4.3 shows that recall is relatively low for a single non-overlapping NV-tree in any configuration. With additional indices, however, recall is improved significantly to the point where it exceeds the 65.8% recall reported in Chapter 3. Furthermore, the figure shows that while the major quality improvements are caused by adding indices and a

partitioning level, further improvements are seen by reading additional partitions, in particular with the priority-based scheme. Overall, we observe the best trade-off between performance and quality with 3 indices (75.4% total recall). Adding further indices yields up to 82.6% recall, but at a cost of higher retrieval time ($\times 3$) and larger database size ($\times 2$).

Detailed analysis shows that with a single non-overlapping index, many close neighbors are lost, which the overlapping NV-tree can find easily due to the redundancy in the leaf partitions. By aggregating results from more than one non-overlapping NV-tree, however, results are improved across all distance ranges; close neighbors are always found, and more distant neighbors are more likely to be found than with the overlapping NV-tree. Note that, in contrast, the strategy of adding additional trees did not show any additional benefits in terms of recall for the overlapping NV-tree.

4.4 Large-Scale Experiments

In this section we present our detailed experiments on a collection of 2.5 billion SIFT descriptors. First, Section 4.4.1 gives an overview of the experimental setup. Then, Section 4.4.2 presents query performance measurements, while Section 4.4.3 analyses the result quality.

4.4.1 Experimental Setup

This descriptor collection has been obtained from 2.5 million images downloaded from the Flickr image sharing website, plus the images used in Section 4.3. The images were processed as before, resulting in a total of 2,485,568,191—nearly 2.5 billion—128-dimensional SIFT descriptors.

We used the same query workload as in the previous experiment, consisting of 500,000 query descriptors. Although the descriptor collection is an order of magnitude larger, we assume the same ground truth set of 248,852 descriptors, as running a sequential scan to determine a new ground truth is much too time-consuming. This may artificially lower the result quality, but as we shall see the effect is relatively small.

As the experimental setup used in the previous section was quite dated, we obtained a moderate server computer containing two Intel Xeon E5420 CPUs running at 2.50GHz, 12MB L2 Cache, and 32 GB of DDR 2 main memory, running at a clock speed of 667 Mhz. Due to the high storage requirements for such a large descriptor collection, as well

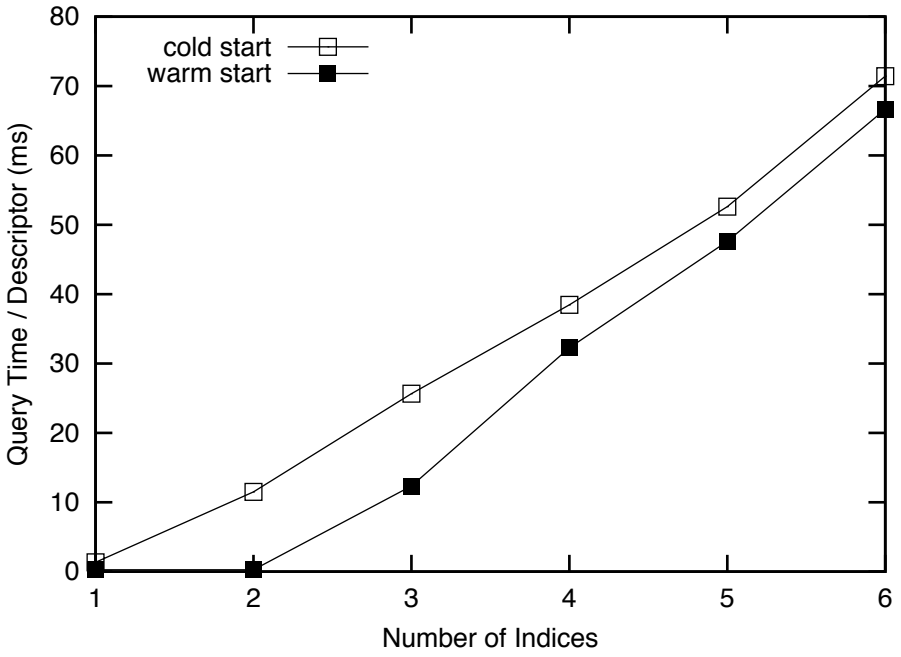

Figure 4.4: Retrieval time of NV-tree configurations for a collection of 2.5 billion descriptors.

as for performance, the server was equipped with 6 hard drives. Three 1.5 TB hard drives are used for storing the NV-tree indices, one for the operating system and related tasks, and two to store the descriptor collections and result files.

4.4.2 Indexing and Retrieval Performance

Each NV-tree consumes about 13 GB of disk space, and takes about 15 hours to construct. As the server has 32 GB of main memory, at most two indices can fit in memory, while configurations with three or more NV-trees can only be partially loaded into memory. In order to get a better understanding of the impact of buffering on NV-tree performance, we measure both a "cold start" where the buffers are empty, and a "warm start" where leaf partitions from the measured NV-trees are loaded into memory in a round-robin fashion until the memory is full. Filling the buffers can take several minutes, depending on the number of indices, but full buffers are clearly more representative of the long-term performance of the system.

Figures 4.4 and 4.5 show the retrieval time and throughput, respectively, for the two buffering approaches. Note that only a single CPU core was used in each case. As the figures show, query processing is very efficient using one or two indices when buffers are full; each query descriptor requires only 0.3 ms of processing time, yielding a remarkable

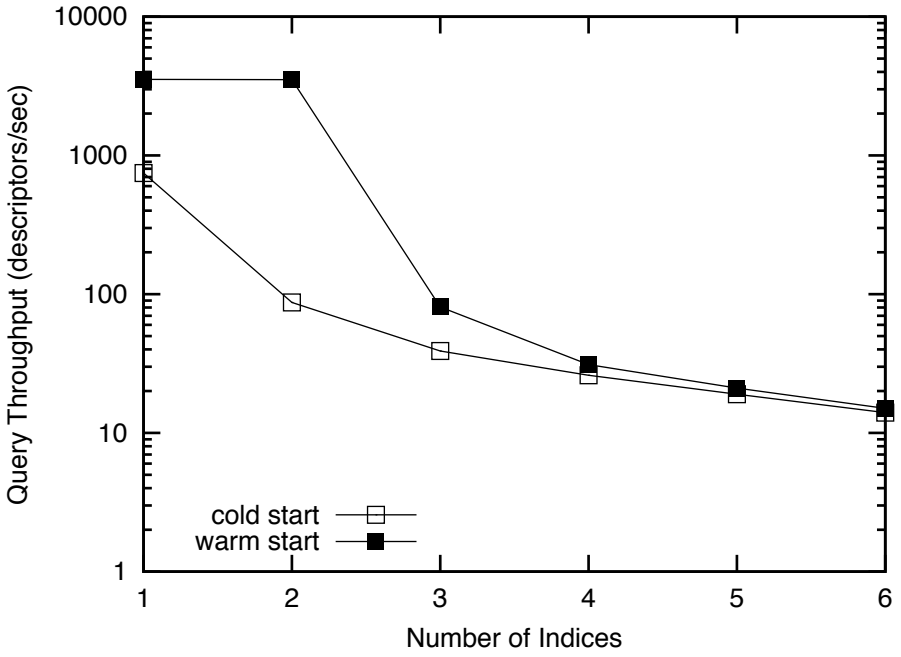

Figure 4.5: Throughput of NV-tree configurations for a 2.5 billion descriptor collection.

throughput of 3,500 queries per second. Once the indices do not fit into memory, however, retrieval time increases significantly, to 12 ms for three indices and up to 50 ms for six indices, with a corresponding reduction in throughput.

Comparing to the "cold start" strategy, the preload time clearly pays off when the indices fit as a whole into main memory. When they do not, the heavy load of additional I/O accesses soon dilutes the performance gains achieved by preloading so the performance gains are only marginal. Nevertheless, the throughput, using three indices in continuous operation, is about 81 descriptors per second.

4.4.3 Result Quality

Figure 4.6 shows the recall for the large collection, compared to the collection used in the previous section. As the figure shows, recall is about 8% lower across the range of indices. Nevertheless, the recall is still quite acceptable, and nearly equivalent with the recall of a single overlapping NV-tree for the 180 million descriptor collection.

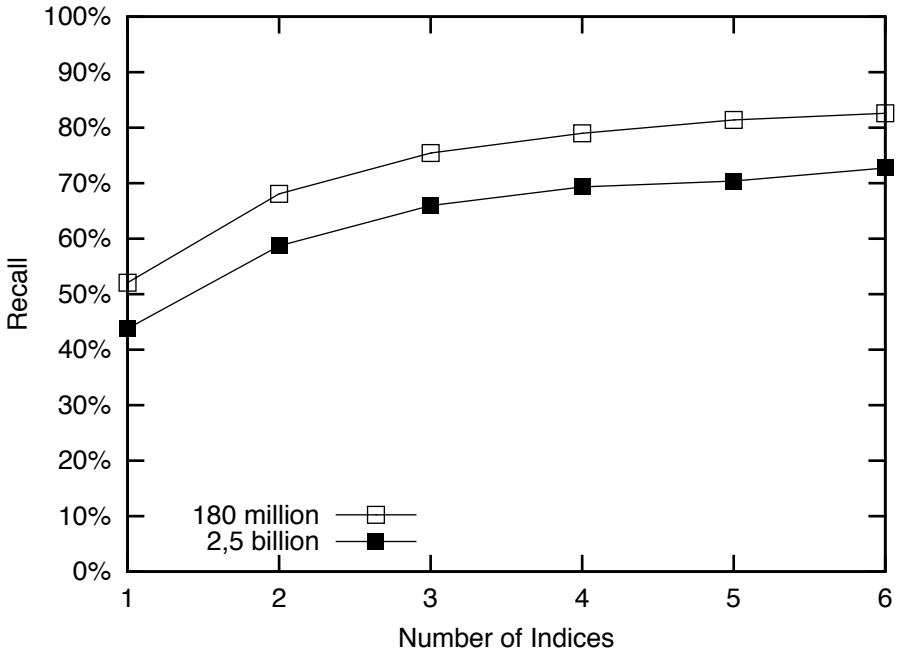

Figure 4.6: Recall of NV-tree configurations for a collection of 2.5 billion descriptors.

4.5 Summary

In this chapter, we have proposed the non-overlapping NV-tree and demonstrated that it works extremely well for providing approximate nearest neighbor search in very large collections of high-dimensional descriptors. Using our preferred configuration of three NV-trees, we achieve 66.0% recall for a collection of 2.5 billion SIFT descriptors. Query processing is also efficient, in particular when the indices can be kept in main memory, as each CPU core can answer up to 3,500 queries per second.

Chapter 5

Industry Strength NV-tree

Decades of research in databases have proven that very efficient, scalable, dynamic and durable systems can be built. Efficiency is such that while the database fits in memory, processing its contents is extremely fast, and when the database exceeds the memory size, then disks are used as efficiently as possible—scalability beyond main memory limitations is not a problem with database technology. The data structures that are fundamental to databases all allow dynamic inserts and protocols for correctly handling simultaneous inserts have been defined—dynamicity is thus not a problem with database technology. Efficient methods to enforce the ACID properties of transactions[1] which protect the database from all sorts of failures, have been very clearly specified—durability is therefore not a problem either with database technology.

So why is it that high-dimensional nearest neighbor algorithms from the literature rarely scale beyond main memory limitations, cannot cope with dynamic inserts, and do not address durability issues? We believe that this is because a *database perspective* has been lacking from the world of high-dimensional indexing schemes. We argue that such schemes: must provide ACID transactional properties; must be disk-based for coping with dynamic insertions, failures, and scale; must carefully balance resources to avoid bottlenecks; and must consider the trade-offs between response time and throughput.

In this chapter we present our implementation of the two desirable properties of dynamicity and durability within the NV-tree. We make the following main contributions:

[1] ACID is an acronym which stands for Atomicity—either the transaction is completed or any impact of its partial execution is removed; Consistency—correct transactions take a database from one consistent state to another; Isolation—each transaction is isolated from concurrent execution of other transactions; and Durability—the effects of a completed transaction are not lost in the event of crashes.

- We demonstrate how to enforce ACID properties within the NV-tree for a large class of important applications, and show that with our implementation dynamic inserts can be efficiently managed.
- We then show detailed performance evaluations of the scalability of the NV-tree using standard image benchmarks embedded in collections of up to 30 billion high-dimensional vectors. Our analysis of the literature shows that these are by far the largest single-server experiments reported anywhere.

Overall, we show that the NV-tree is not only an extremely scalable approximate indexing strategy, but also a unique nearest neighbor search system that meets industrial standards for a database management solution.

The remainder of this chapter is organized as follows. In Section 5.1 we describe briefly the characteristics of the applications we have in mind, and discuss their impact on the implementation of the ACID properties within a high-dimensional indexing scheme. We then describe how the NV-tree index can be extended to support dynamic inserts. We first describe the overall insertion process for a single NV-tree in Section 5.2, then describe the enforcement of the necessary ACID properties for a set of NV-trees in Section 5.3, and subsequently evalute the performance of dynamic updates in Section 5.4. Finally, in Section 5.5 we evaluate the performance and result quality of the NV-tree using the largest single-server experiments reported in the literature, before summarizing our findings in Section 5.6.

5.1 Application Characteristics and ACID Properties

Large collections of multimedia objects, and thus high-dimensional vectors, are typically dynamic and require efficient insertions. Furthermore, these insertions must be concurrent with searches as web-scale collections cannot be taken offline for maintenance. It is therefore important to implement them as transactions.

While this seems completely obvious from a database perspective, it is a rare paper about high-dimensional indexing that ever discusses mechanisms to recover from failures, let alone implement the ACID properties. Instead, it is implicitly assumed that the entire index can be recreated from scratch if a serious crash happens. We believe, however, that this is not acceptable from an operational point of view, as it may take weeks to index a sufficiently large data collection. Worse, the raw data may even not be available anymore, as it may have been provided by the data owner or a third party.

While traditional techniques for enforcing ACID properties of transactions can be used, both the data structure and the applications have particular characteristics that in some cases require special attention but in other cases allow for efficient implementation. The typical application we are concerned with is a web-scale similarity-based multimedia service, such as a copyright protection service or a forensic analysis service, where queries are multimedia items that are transformed into high-dimensional vectors. Such services typically have strong performance requirements for query processing, but also a steady stream of new material to include in the collection.

Vectors corresponding to individual multimedia objects are inserted together. Typically, this is a large number of vectors; e.g., around one thousand vectors in an image database and a much larger number for more complex media objects. Furthermore, insertions can be batched, allowing a focus on throughput. We observe, for example, that in many of today's large scale media services, such as YouTube, media does not become immediately available, as it is first processed by the services in potentially multiple ways before making it available. For such media sharing sites, media files to be protected from copyright violations are most likely added by administrators. And in the case of forensic applications, police forces insert data only rarely, for example as a result of new offensive media collections being discovered.

Thus there exists a class of applications where it is safe to assume that (a) updates are made centrally, and (b) that throughput is more important for this update thread than response time. For such applications, it is feasible to serialize insertions such that only one insertion transaction is running at each time, which greatly simplifies the implementation of the insertion process. Due to serialization, two insertion transactions will never conflict, which means that a simple locking mechanism based on tree-traversals is sufficient to enforce *isolation*. Because insertions are never aborted and they never deadlock, ensuring *atomicity* is only needed when the system crashes. Furthermore, since at most one insert transaction is running concurrently, enforcing *durability* is greatly simplified. Finally, since there are no constraints on the vectors, as such, the traditional notion of *consistency* does not apply here (or alternatively, with these applications, atomicity implies consistency).

Note that deletions may occur, although they will be very rare in practice. Deletions can be implemented using very similar techniques to those implementing insertions described below. By adding the deleted media item to a list of deleted media, we can avoid returning partially deleted items; when deletion is finished the item can be removed permanently from this list. Note that the vector collection is never updated, however, as a modified me-

dia item will typically yield a very different set of vectors; updates must thus be modelled as deletions followed by insertions.

5.2 Insertions

Insertion to NV-tree leaves proceeds as follows. When the correct leaf node is found, using the exact same process as during search, the position within the leaf node is calculated and the identifier is inserted in the designated position. During index creation, leaf nodes are not filled completely (typically they are between 50% and 85% full, and about 70% full on average) in order to leave space for such insertions. Once a leaf node is filled, however, it must be split in order to provide more storage capacity within the tree. This split operation is a complex and expensive operation which must be carefully implemented; the remainder of this sub-section describes the details of the split operation and its implementation.

5.2.1 The Split Operation

The basic method to split a node is very similar to the index construction process. A new internal node is created with two new leaf nodes as children. A projection line is then assigned to the internal node using the same method as during index construction. The contents of the full leaf node are projected along the projection line and inserted to the appropriate leaf. Each leaf is then also assigned a projection line and the contents of the leaf ordered based on the projection to that line.

Using this basic method, however, the index would quickly become very deep, as each leaf would always be split into two leaves and all the new internal nodes would have only two children. A better method is to consider a group of l leaves together, and redistribute their contents to $l + 1$ leaves. We chose to go one step further, and consider the group of leaves described in Chapter 4 as the unit of splits; when that group exceeds $6 \times 6 = 36$ leaves, the group is split into multiple new groups, typically between 4 and 8 new groups, depending on the distribution of the high-dimensional vectors in the original group undergoing a split.

5.2.2 The Leaf-Group Database

In order to project the contents of the filled leaf and its neighbors, it is necessary to use the original vectors, as a) a new projection line may be chosen for the internal nodes, and b) each new leaf has a new projection line. As the actual vectors are not stored in the leaf nodes, they must be retrieved from disk. Since they are randomly distributed over the whole vector collection, however, a naive implementation would result in a large number of very costly random disk accesses for each split.

A more efficient implementation is to maintain an independent vector database for each NV-tree. Each tuple in each database stores the identifier of the vector, all of its components (the values for all its dimensions), as well as the leaf-group identifier within which it is stored in the corresponding NV-tree. Each database is organized according to a clustering index defined over the leaf-group identifiers. Therefore, all the vectors that are together within one leaf-group of one NV-tree are together in the database, hence the "leaf-group database" name we use here. It is thus efficient to load all the vectors belonging to the leaf-group that is being split, which minimizes the number of random I/Os that must be performed. The multiple NV-trees indexing the same collection have very different vectors in their respectives leaves because the construction of each index uses the pseudo-random projections discussed in Chapter 3. Therefore, the grouping of vectors per leaf-groups differs across NV-trees, explaining why there is one such independent leaf-group database per NV-tree indexing the collection of vectors.

To further improve efficiency, inserts and updates in the leaf-group databases are buffered in memory. A thread pushes to the leaf-group databases the contents of the buffer, independently from the thread pushing to disks the updates to the leaves within the leaf groups of the NV-trees that are splitting. The buffer is also clustered according to the identifiers of the leaf-groups, which facilitates propagating the updates to the databases. While vectors in the leaf-group databases are clustered by the identifier of their leaf group, no other order is imposed on the vectors within each group, simplifying implementation.

Storage technology is progressing, however, and the ubiquitous SSDs (Solid State Disks) are quickly becoming viable for large-scale industrial data stores. With SSDs, issuing many small random reads is much more efficient, which means that it may not be (performance-wise) necessary to cluster the vectors per leaf-group and to maintain one database per NV-tree. Instead, maintaining a unique database where the full vectors are stored sorted on their identifiers, and that is randomly accessed by the split operations is likely to be as efficient, while consuming less storage space. Fully understanding the impact of SSD technology is part of our future research agenda.

5.3 Enforcement of ACID Properties

In this section, we consider the enforcement of ACID properties of transactions during NV-tree insertions. As mentioned above, traditional consistency concerns do not apply here. We therefore start by considering isolation for a single NV-tree in Section 5.3.1, then consider atomicity and durability in Section 5.3.2, before addressing some practical issues relating to concurrent insertions to multiple NV-trees in Section 5.3.3.

5.3.1 Isolation

Isolation is implemented using a standard locking algorithm adapted from the B^+-tree. A search thread starts by obtaining a read lock on the root. Before accessing a child node, the thread must then obtain a read lock on that node. At that point, the lock on the parent can be released. Finally, the leaf group selected for retrieval is locked and only released after all necessary identifiers have been retrieved from the designated leaves. Note that locks are implemented using `pthread` mutexes; each internal node contains the mutexes for all its children and the leaf groups are locked as a unit since they are treated as a unit during both retrieval and node splits. Note also that the overhead of obtaining locks is low and hence locking is always activated.

The insertion process uses the same locking mechanism, except that finally an exclusive lock is acquired for the leaf group, preventing concurrent insertions into and concurrent retrieval from that leaf group. In the case of a leaf group split, a new internal node is created pointing to all the newly created leaf groups; the lock on the original leaf group is sufficient to protect the modification of the parent node.

Since each query or insertion transaction needs to access multiple trees multiple times, it is necessary, however, to consider the overall interaction between search and insertion transactions. Recall that insertion transactions are serialized; they are therefore assigned with ever-increasing transaction identifiers (TIDs) which are stored with each inserted vector. Isolation is enforced by omitting from the query result vectors with transaction identifiers larger than that of the last transaction that committed before the search started.

5.3.2 Atomicity and Durability

For atomicity and durability, we adapt the standard write-ahead logging protocol (e.g., see (Gray et al., 1981; Mohan, Haderle, Lindsay, Pirahesh, & Schwarz, 1992)). Note that while it is difficult to prove the correctness of the implementation of durability, the

implementation has been tested methodically by pausing operations in certain places and crashing the computer; in all cases has recovery been successful.

The write-ahead log is distributed to a number of files. The *vector collection log file* contains all inserted vectors. A *global log file* is used for logging information necessary for the overall recovery process. Finally, each NV-tree has its own *index log file*, which stores the information necessary to undo and redo split operations. Special care is taken not to overwrite the disk space belonging to leaf groups that have been split until the splitting transaction has committed. The fact that a single insertion transaction is writing to each index helps to simplify the process, as only committed vectors exist immediately after commit.

The recovery manager uses regular *checkpoints* to facilitate efficient recovery. During checkpoints, the contents of the entire NV-tree are flushed to disk and information about the status of the internal nodes is stored in a *checkpoint file*. The leaf-group database itself is already synchronized on disk, aside from the contents of the leaf-group buffer; to avoid writing to the entire leaf-group DB, the contents of the buffer are also stored in the checkpoint file. Two different checkpoint files are used, so that the previous checkpoint is valid until the completion of the current checkpoint. Furthermore, checkpoints are only taken immediately after an insertion transaction commits, which means that after the checkpoint everything is consistent on disk and contains only committed vectors.

During recovery, the latest checkpoint file is first read and the status at the time of the checkpoint is adopted for the internal nodes, the leaf nodes and the leaf-group DB. Then the split operations are retrieved from the index log file, and those split operations that were performed due to committed transactions are re-played on the internal structure, while other split operations are ignored. At this point, the internal structure is correct, as of the time of the crash, but vectors may be incorrectly included and/or missing. Next, therefore, vectors that belonged to uncommitted transactions, but made it to the leaf nodes of the NV-tree are removed; note that no such vectors are ever found in the leaf-group DB, because they are only added to the leaf-group buffer when the transaction is ready to commit and the checkpoint is only written after commit. Finally, the vector collection log file is used to re-insert the committed vectors that did not make it to disk, both to the NV-tree and the leaf-group DB, taking care to avoid re-insertion to the split leaves.

Note that since the insertion operations are serialized and do not conflict, the undo and redo phases can be performed in any order. Since vector removal requires moving other vector identifiers in the leaves, however, it makes sense to do that before inserting new identifiers that would subsequently need to be moved.

The recovery process also supports crashes within the recovery process itself, or repeated crashes. The process starts by recovering the memory based structure into the last consistent stage before the crash, and no disk writes are necessary during this stage that would interfere with or complicate subsequent recovery efforts in the case of repeated crashes during the recovery process. Only after successful recovery of the indices in memory, and after the system has successfully generated a checkpoint for each index, will the system invalidate the log files and accept insert or query requests. As there are always two separate checkpoint files, even a crash during a checkpoint does not invalidate the correctness of the structure, nor complicate the recovery progress. The only performance related parameter during recovery is the time passed since the last successful checkpoint. When the crash happens immediately after the checkpoint, the recovery process is short as hardly any undo and redo operations must be carried out. If the crash happens long after the last checkpoint, on the other hand, or during the generation of a new checkpoint, then the recovery effort is significant as there will be many undo and redo operations, in addition to a large number of split operations, that must be taken care of.

5.3.3 Practical Issues with Multiple NV-trees

When inserting to multiple NV-trees, each tree is preferably located on a separate hard drive (as are the log files) so that the full write-back capacity of the disks can be used for the leaf-group DB thread. In the early stages of our implementation, we inserted synchronously to all NV-trees in the system, as we expected the performance of insertion to be rather uniform. We immediately observed a major performance bottleneck, however, as only about 40% of the available disk capacity of the three available hard drives was used, thus significantly reducing the total insertion throughput of the system.

The reason for this reduced insertion throughput was twofold. First, the number of disk operations caused by an insert transaction can differ significantly between NV-trees depending on the number of split operations required (this is by far the most expensive operation). A significant amount of time was thus spent waiting for the slowest NV-tree (which would be different from one transaction to the other) to finish the current insertion transaction before the next one could start.

The second, and more serious, cause for the reduction in disk throughput was caused by a problem in the Linux disk scheduler we were using. To avoid idle disk time, the buffer manager decouples disk write operations as seen by the applications from the actual disk operations by delaying the write-back of dirty pages as long as possible. The performance of the write-back scheduler degrades, however, in the case of intensive access to many

hard drives at the same time. When the buffer cache size fills above a certain threshold of main memory, the internal scheduler stops all further disk operations and bursts a full write to a single disk, rather than triggering writes to several disks at once. During these bursts all other interaction with the disks is blocked, up to several minutes in the worst case, meaning that the other disks are spinning idle as they had long finished their part of the current insertion transaction.

In order to compensate for this problem, we decoupled the insertion process (as well as logging and checkpointing) for each NV-tree. Each NV-tree can thus be inserting from a different transaction, but they must all process the transactions in the same order. Since transactions may progress differently across different trees, more than one uncommitted transaction may have inserted vectors to some trees before a crash. Due to the ordering of transactions, however, the last NV-tree to finish a transaction decides the commit time and transactions will therefore commit in the same order, and all the techniques described above are unaffected by this change. Using this optimization, disk utilization was improved from about 40% up to 75% to 80%, without violating the previously described ACID properties.

5.4 Performance of Index Maintenance

In this section we investigate the performance of dynamic inserts, while guaranteeing ACID properties, as described above. As the index experiences splits upon inserts, it is also important to verify that the evolution of the data structure does not dramatically impact the ability of the NV-tree to correctly identify nearest neighbors. We first discuss insertion throughput and then result quality.

5.4.1 Experimental Setup

This experiment was designed to show the two interesting cases that govern the performance of inserts. First, when the index fits in main memory, inserts are done in memory and later asynchronously pushed to disks, resulting in excellent performance. The second case arises when the index is larger than memory. In this case, it might be required to load from disks the data pages into which inserts take place, which is not only slow, but also interferes with writing back updated pages. We therefore expect this second case to show much worse performance.

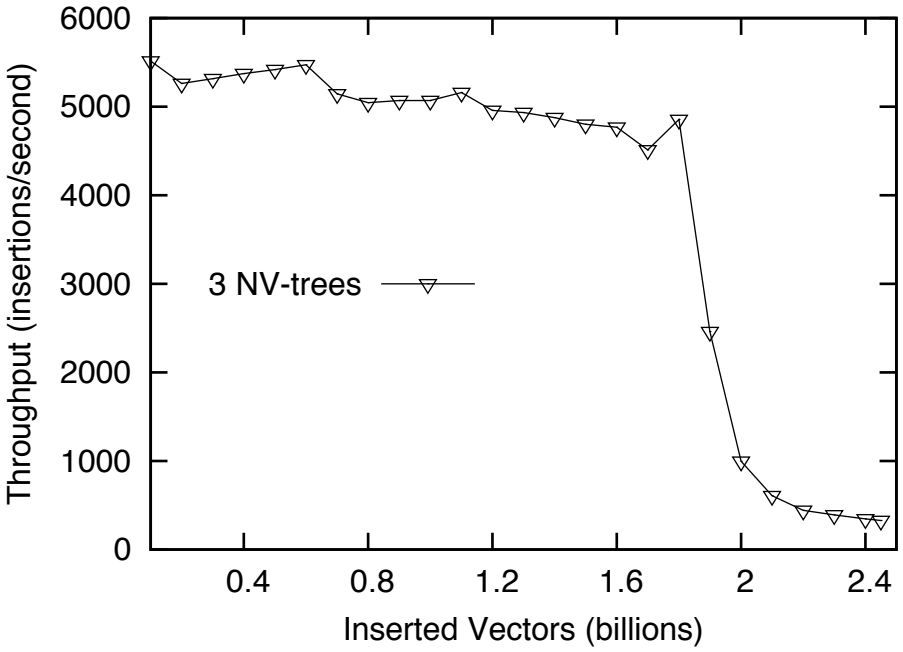

Figure 5.1: Insertion throughput (three NV-trees; six hard disks; 32 GB of main memory).

To illustrate these two cases, we used a machine with only 32GB of main memory. We started by indexing a collection of 36 million vectors with three NV-trees. This is a tiny collection which can be indexed very quickly, and the resulting NV-trees together occupy slightly more than 500MB. We then ran sequences of 1,000 insertion transactions. Each transaction is inserting 100,000 new vectors into the three NV-trees, which means that each sequence of insertion inserts 100 million new vectors. We then observed the time it takes for each sequence to complete. We repeated this process and ran multiple sequences until each of the NV-trees contained nearly 2.5 billion vectors, occupying about 328 GB each.

5.4.2 Insertion Throughput

Figure 5.1 shows the evolution of the insertion throughput (measured by vectors inserted per second) for the duration of this workload. In the beginning of this workload, all three NV-trees fit into main memory and the throughput is excellent, around five thousand vectors per second. After running 16 such transactions, thus inserting 1.6 billion vectors, the 3 NV-trees no longer fit in main memory. After that point, Figure 5.1 clearly shows the insert behavior corresponding to the second case discussed above, where the rate of inserts slows down significantly due to conflicting disk operations. It should be noted,

however, that with throughput of 500 vectors per second, each insertion only takes 2 ms., which is significantly less than one disk operation per insertion.

The most important aspect of this experiment is not the reduced performance of inserts after 1.6 billion vectors have been inserted and the index no longer fits in memory. (With the machine used in the experiments described in the next section, memory would have been exhausted after having inserted close to 25 billion vectors.) Rather it is the fact that even when the collection no longer fits in memory, dynamic maintenance of the index is still possible as the insertion throughput degrades gracefully.

While we have not formally measured the overhead of enforcing the ACID properties of the NV-tree, we still believe that the overhead is insignificant. The logging mechanism for transactions does not impact the overall performance as it is operated independently from the insertion threads for each individual NV-tree. As the log file is stored on a separate disk, it also does not compete with the I/O demand of the insertion process. Node splits are logged on the same drive as the index is residing, however, but the amount of data that needs to be logged is tiny in comparison to the I/O operations required for performing the actual split. Thus, also here the overhead can be considered negligible.

The only major effect on performance is due to the generation of checkpoints, where all leaf nodes residing in main memory are written back to disk. Checkpoints are thus becoming increasingly expensive with the size of the index; this is most likely one of the factors explaining why the insertion throughput of insert transactions is slightly decreasing over time even when the whole index fits in memory. Considering, however, that one NV-tree index with 2.5 billion vectors is only about 14 GB in size, and that it only takes about 45-90 seconds to write back that amount of data on today's hard drives, these delays affect overall performance only marginally.

5.4.3 Retrieval Quality

To evaluate the query performance of the NV-tree, we borrow the worklload and ground truth defined in Chapter 3. We included these 248,212 vectors in the database of 36 million distracting vectors, computed from a set of random images downloaded from Flickr. Once this database was created, we ran the same 500,000 queries as before and computed their recall, i.e., we count how many of these 248,212 ground truth vectors are found. We repeated that same workload after every insertion transaction (of 100 million vectors), to observe how the quality of the answers evolves as the database grows.

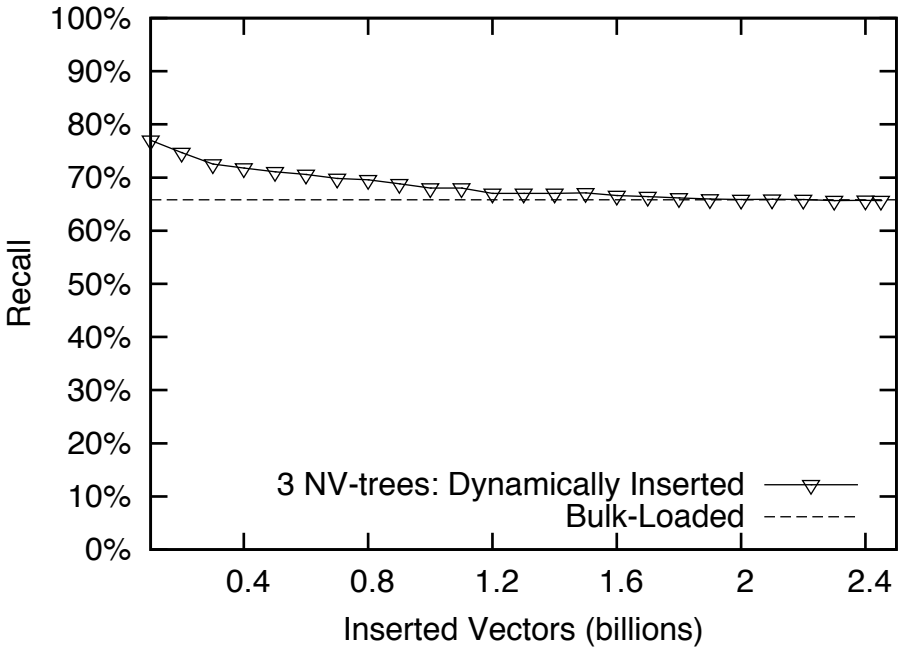

Figure 5.2: Recall, increasing the insert load, two NV-tree settings

Figure 5.2 plots the recall percentage from the 500,000 queries described above, as the collection grows in size. The figure shows a configuration where the results are aggregated from three NV-trees. As the figure shows, recall drops slowly as the collection grows, which was expected. For comparison, the figure also contains a dashed line indicating the result quality when the NV-trees for the 2.5 billion vectors is constructed from scratch via bulk loading. As the figure shows, the results for the dynamically created NV-trees and the bulk-loaded NV-trees are identical, meaning that dynamicity has no impact on result quality.

5.5 Large-Scale Experiments

In the previous sections we have shown how to transform the NV-tree from a disk-based high-dimensional index to a full-fledged database solution for scalable nearest-neighbor retrieval, by allowing dynamic updates and supporting the ACID properties of transactions, thus ensuring dynamicity and durability. We now report on scalability experiments, where we have embedded standard image benchmarks into a collection of nearly 30 billion vectors, demonstrating the extreme scalability of our database solution and showing that it is a complete industry-strenght solution for large-scale approximate high-dimensional indexing.

5.5.1 Experimental Setup

To evaluate the query performance of the NV-tree, we yet again use the ground truth defined in Chapter 3. We briefly recall the definition of this ground truth for completeness. We used a sequential scan to determine the 1,000 nearest neighbors of 500,000 query vectors, all coming from a collection of 180 million SIFT vectors. Analyzing the resulting 500M neighbors, we identified 248,212 vectors as being contrasted enough to be considered the true nearest neighbors of the query vectors. Contrast here is directly derived from criterion of (Lowe, 2004); a neighbor is considered a true neighbor if it is significantly closer than neighbor number one hundred.

We then embed these 248,212 vectors within vector sets of varying cardinalities to distract the search. These sets of *distracting vectors* have been created by extracting SIFT features from nearly 30 million images randomly downloaded from Flickr. These images are ignored here, however, as we solely focus on nearest neighbor retrieval of individual vectors. The resulting distractor sets contain about 30 million vectors, 180 million, 300 million, 2.5 billion, 3 billion and nearly 30 billion vectors, respectively. For the sake of completeness, Table 5.1 gives the exact figures for the collections used in this experiment.

These experiments focus primarily on recall, i.e., how many of these 248,212 ground truth vectors are found using the original set of 500,000 queries when varying the number of distractors, but we also report on the retrieval performance. We are not aware of any other single-server experiments ever published where recall measurements are obtained from searching the nearest neighbors of individual query vectors lost within nearly 30 billion distracting vectors.

We ran experiments using a Dell r710 machine that has two Intel X5650 2.67Ghz CPUs. Each CPU has 12MB of L3 cache that is shared by the 6 actual and 6 virtual cores. There are therefore 24 cores, 12 being real processing units. The RAM consists of 18x8GB 800Mhz RDIMMs chips for a total of 144GB. That machine is connected to a NAS 3070 from NetApp offering about 100TB of disk space, in a RAID configuration. We ran the experiments using a single core. Some experimental settings involve multiple NV-trees; they are probed one after the other—no parallelism is enforced here while this could be trivially done. We focus mostly on one, two or three NV-trees, but report some results with up to six NV-trees.

Table 5.1: Distracting vector collections

Collection	Collection Size (SIFT vectors)	(on disk)	NV-tree size (one tree, on disk)
30M	28,799,690	4 GB	180 MB
180M	179,443,881	24 GB	1 GB
300M	305,443,749	40 GB	2 GB
2.5B	2,485,568,191	328 GB	14 GB
3B	3,040,856,472	401 GB	17 GB
30B	28,484,904,924	3.7 TB	162 GB

5.5.2 Quality of Nearest Neighbor Retrieval

Figure 5.3 shows the recall for various collections. The x-axis shows the size of the collection used in this study, while the y-axis shows recall obtained by using a varying number of NV-trees. Up to three NV-trees were used against all the datasets. We also considered using up to six NV-trees to improve recall; as such experiments are complicated and time consuming, however, we used only two moderate size datasets for this purpose.

When using a single NV-tree, recall is relatively low. Close to 54% of the 248,212 ground truth vectors are found when they are lost in the 30M collection. This percentage then slowly decreases as the distracting collection grows, to about 38% when challenged by the 30B collection.

Using additional indices dramatically improves performance, however. With the 30M collection, recall jumps to 72% with two NV-trees and 79% using three NV-trees. At the other end of the figure, with the 30B collection, recall is lower as before but remains remarkably good given the size of the distracting collection: 52% with two NV-trees and 58% with three NV-trees.

Using more than three NV-trees provides a slight recall improvement, but not as dramatic as going from a single NV-tree to two and three. Further increasing the number of NV-trees is therefore not a worthy option, since it increases the pressure on the storage and main memory. It also increases the retrieval cost as more trees must be probed.

5.5.3 Retrieval Performance

We now turn to the retrieval performance. We measured the response time of each individual query as well as the throughput of the system, determining the number of query

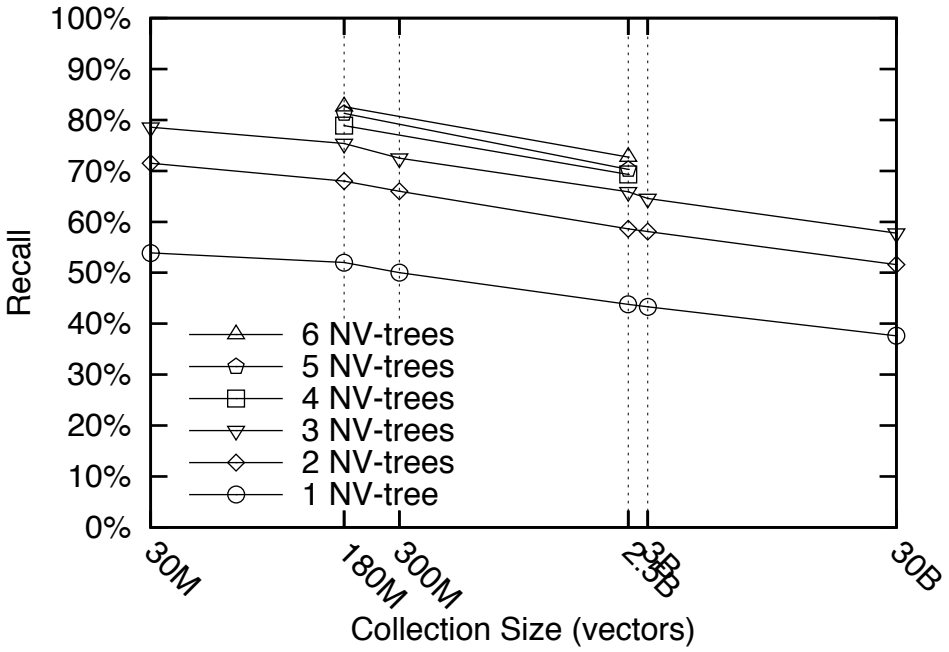

Figure 5.3: Recall, varying dataset sizes, varying the # of NV-trees

vectors it can process per second. As the results are highly dependent on hardware, and memory size in particular, we focus on the key retrieval performance elements: the dominating costs related to the CPU consumption and main memory latency when the NV-tree indices fit in main memory; and the performance of disk reads when the indices can no longer fit within memory.

Recall that the main memory of our server was 144 GB, which means that all the leaves of three NV-trees can fit into memory for all collections except the 30B collection. When the various indices fit entirely in main memory, then answering each query vector is extremely fast. It takes a fraction of a millisecond to process one vector against one NV-tree, and the throughput we observed ranged from 2,000 to 3,000 query vectors per second per tree.

It should be noted, however, that this throughput can be achieved only when each NV-tree index entirely resides in main memory, that is, once all its leaves are in RAM. The leaves can be purposely loaded to memory before running queries, or loaded as a consequence of the querying process. In the latter case, the first queries are slow as they need to fetch data from disks, while subsequent queries are faster as they are more likely to find the data they need in memory, loaded by previous queries.

When using the 30B collection, on the other hand, the main memory can not fit even all the leaves of one NV-tree. Each query vector is likely to access a different random part of

the index and no buffering policy copes with such demanding access patterns, meaning that the system must retrieve data from disks for almost every query vector. The response time is therefore much larger.

The duration of each I/O varies but typically is within a range of 5 to 20 milliseconds. I/Os are largely random and it is extremely complicated to precisely know how they are handled by the NAS NetApp server. It serves many users in parallel, has various level of caches that we can neither control nor observe, and stripes the data across its disks in an opaque manner. Overall, however, about 50 query vectors could be processed per second per tree, as the NV-tree meets the design criterion of one disk read per query.

5.5.4 Discussion

Overall, as before, using three NV-trees is the best option. It proved to work extremely well even when dealing with a truly large-scale collection and a challenging experimental setup: the NV-tree can correctly identify the majority of the nearest vectors of each query vector even when these vectors are lost in up to 30 billion distracting vectors. With three trees, processing cost is moderate, the recall is excellent, and the storage demands are very reasonable: only about 500GB are needed to keep all the leaves of three NV-tree within main memory even for the 30B collection.

We conclude this discussion by reporting on the scalability of the experiments found in the literature. We have extensively reviewed the existing works and retained the ones where the high-dimensional indexing schemes that were proposed, as well as the scale of the experiments run, contributed significantly to advancing the state of the art. In addition to the work proposed in this thesis, we report scale of experiments in six other high-dimensional indexing schemes. Figure 5.4 summarizes the results.

Joly, Frélicot, and Buisson (2003) was among the pioneers to design a video content-based copy identification scheme relying on local features extracted around interest points; experiments using 121M vectors were reported. Douze, Jégou, Sandhawalia, Amsaleg, and Schmid (2009) proposed an indexing strategy that optimizes the trade-off between memory usage and precision. It is tested using a database of 110 million GIST vectors. An early version of the NV-tree is described in (Lejsek, Ásmundsson, Jónsson, & Amsaleg, 2009) (see Chapter 3) and evaluated against 180M descriptors. Jégou et al. (2011) proposed an indexing scheme based on the notion of product quantization and evaluated its scheme by indexing 2B vectors. Lejsek et al. (2011) described the non-ACID NV-tree (see Chapter 4) and the experiments in this paper used 2.5B local feature vectors. Jégou et al. (2012) designed a very elegant local feature aggregation scheme enabling a very compact

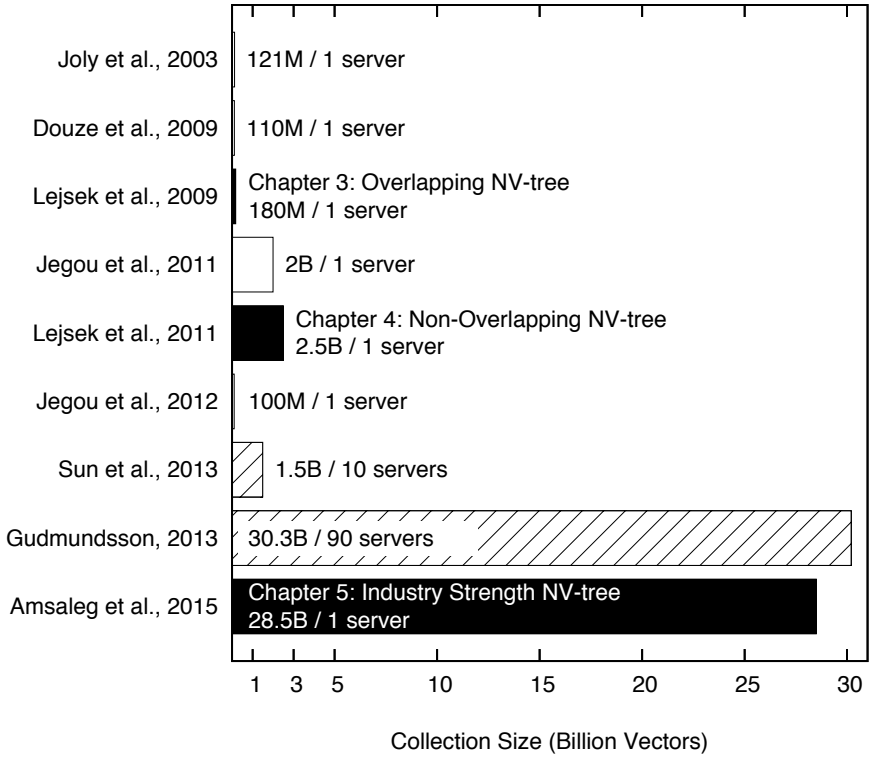

Figure 5.4: Comparison of the scale of experimental collections reported in the literature. Black bars represent work reported in this thesis. Shaded bars represent multi-server configurations, reported here for completeness.

representation in RAM of high-dimensional vectors while preserving most of their distinctive power. Their experiment used 100M super-vectors resulting from the aggregation of several billion local feature vectors. Sun et al. (2013) also relied on the aggregation scheme proposed by Jégou et al. (2012), indexing 1.5B images, but using 10 servers, however. In (Moise, Shestakov, Guðmundsson, & Amsaleg, 2013), about 100M images were indexed, which amounts to managing close to 30B high-dimensional vectors. That work, however, used the MapReduce programming model and the Hadoop framework, with the serch experiments running on 90 computers. While the scale of the experiments in this work is worth noticing, the philosophy of the system is quite different as it is dedicated to processing batches of queries and cannot run interactively.

5.6 Summary

In this chapter we decribed how the NV-tree fulfills industry requirements, by enforcing the ACID properties of transactions for a large class of important applications in the mul-

timedia domain. Experiments show that with our implementation dynamic inserts can be efficiently managed; when the index fits in memory, performance is excellent, but when the index no longer fits in memory, performance degrades gracefully.

We then described detailed performance evaluations of the scalability of the NV-tree using standard image benchmarks embedded in collections of up to 30 billion high-dimensional vectors. As our analysis of the literature shows, these are by far the largest single-server experiments reported anywhere. Overall, we have therefore shown that this extended NV-tree is not only an extremely scalable approximate indexing strategy, but also a unique nearest neighbor search system that meets industrial standards for a database management solution.

Chapter 6

Conclusions

Nearest neighbor search in high-dimensional space is an operation that is fundamental to many applications. Recent progress in approximate high-dimensional indexing has resulted in several approaches which can handle several hundred million to a few billion high-dimensional vectors and exhibit excellent response times. However, when moving from the research lab to the real world, scalability is not the only challenge that must be met. To be useful in industry, a high-dimensional indexing method must also provide dynamicity—the ability to cope with on-line insertions of new high-dimensional items into the indexed collection—and durability—the ability to recover from crashes and avoid losing the indexed data if a failure occurs.

6.1 Summary of Contributions

In this thesis we have proposed and described the NV-tree, an indexing structure which is one instance of these scalable approaches. We described the fundamentals of the NV-tree, different strategies for its construction and different strategies for traversing the tree structure and aggregating the descriptor references.

Our experimental results show that the NV-tree scales very well, starting from collection sizes of 30 million high-dimensional feature vectors up to collection sizes of nearly 30 billion. As our analysis of the literature shows, these are by far the largest single-server experiments reported anywhere. For the collection sizes typically reported in the literature the data could be easily hosted within main memory. Once entering the billion scale, however, this becomes increasingly difficult when memory requirements go into the hundreds or thousands of GB.

There is no doubt that utilizing main memory effectively in the retrieval process is of high importance. The NV-tree supports this by replacing actual distance calculation by rank aggregation, thus compressing the original feature vector down to a single one-dimensional identifier which can be held in main memory much more easily. In the end, however, we believe that data quantity will always win over memory capacity and therefore it is important that performance degrades gracefully as memory capacity is exceeded, which is not the case for alternative approaches, as discussed in Chapter 2.

Searching the NV-tree itself incurs negligible CPU overhead. When the index fits entirely in main memory a single server computer with 2 quad-core CPUs can process between 150 and 250 thousand descriptor queries per second. This is equivalent to about 200 image queries, assuming that the average image yields around one thousand feature vectors.

We have also analyzed the properties of a large-scale copy detection application using the well known SIFT descriptors. We show that the SIFT descriptors are very distinctive and have high contrast, even in large collections. Furthermore, we show that using contrast-based ground truth sets is necessary to obtain meaningful results for all queries. We have shown that the NV-tree returns very good approximate results for this workload, and we believe that the NV-tree can be used for any large-scale application, where the data set can be shown to have contrast and yield meaningful results.

In a comparison against the ϵ-distance based approximate hashing method LSH we discovered that both LSH and the NV-tree are good indexing schemes for nearest neighbor search in high dimensional space. While both methods are built on the concepts of projection to lines and partitioning, however, they have very different properties. The NV-tree is a tree-structure which guarantees fixed size I/O operations and a maximum size on the result set. LSH is hashing based and might in extreme cases return very large result sets. Overall, the NV-tree gives better results using less disk-space, making it the superior approach.

Finally, we have adapted the NV-tree to meet industry requirements, by enforcing the ACID properties of transactions for a large class of important applications in the multimedia domain. Experiments show that with our implementation dynamic inserts can be efficiently managed; when the index fits in memory, insertion performance is excellent, but when the index no longer fits in memory, performance degrades very gracefully. This makes the NV-tree the only high-dimensional indexing method reported in the literature which satisfies the three requirements of scalability, dynamicity and durability.

6.2 Industry Impact

As mentioned in the introduction, the technology described in this thesis is already in use at Videntifier Technologies (`www.videntifier.com`). Videntifier's search engine currently indexes and identifies videos from a collection of nearly 150 thousand hours of video, at about 40x real time. Furthermore, about 700 hours of video material is dynamically inserted to the ever-growing index every day.

The database service built upon the NV-tree index already has several large international police organisations as clients, such as INTERPOL, the US National Center of Missing and Exploited Children and the UK Home Office, supporting their efforts in fighting the distribution of child abuse content over the internet and their responsibility to get hold of perpetrators and rescuing the victims. Each of the three organisations holds large reference collections of illegal content that must be organised and made available to their members and associates for intelligence work and forensic investigation. Typical collection sizes are in the order of a hundred thousand video hours and tens of millions of still images.

A separate area of application is the fight against terrorist and hate-speech propaganda. Videntifier also serves two clients in this domain. Very recently the company also looks into business in the multimedia analysis domain, in particular by installing the Videntifier client (including visual descriptor extraction) on end-user devices such as smartTVs and tablet PCs. This kind of development requires massive throughput capabilities in terms of visual identification as eventually many millions of concurrent devices will need to be handled.

Of course the NV-tree is not the only technology that is incorporated in the products and services Videntifier Technologies are offering to their clients, but it is the central component and the company's competitive advantage in the field of visual identification. Other components—partially published in white papers and patents by the Videntifier team–include:

- The *Videntifier descriptors*, a SIFT derivative that has shown to yield fewer, more descriptive features than the original algorithm. Furthermore, the descriptors have 72 dimensions, which is significantly lower than SIFT and thus require less storage space and processing time when calculating the projections (Dađason, Lejsek, Jóhannsson, Jónsson, & Amsaleg, 2010; Lejsek et al., 2006) .
- A *scalable descriptor filter* used for representing video content by a stream of visual descriptors with minimal redundancy and scene-level analysis. This is the basis for

Videntifier's capability to identify content at extremely high accuracy in terms of time locality. In more than 95% of the cases a single video frame is enough to reliably identify the video content down to the correctly matching reference scene in the video.

- An algorithm to reliably *filter false-positive matches* based on the locations of the matching descriptors within the image. This geometric alignment between the query point locations and the reference points is state-of-the-art of verification, but it is not always trivial to extract exact point locations. In the context of videos, in particular, camera movements like panning and zooming can have significant influence on the descriptor locations. Therefore Videntifier Technologies developed an algorithm which follows the camera movements during descriptor extraction and thus tries to match descriptors into a global unified location.

Last but not least there are a range of user interfaces that have been developed to help law enforcement investigators with their work. Such user interfaces are coupled with a command-line interface that provides access to the NV-tree but also a traditional relational database system storing meta-information about the individual video and image files, including the exact timing information of individual scenes within a particular video. In addition to handling static files, Videntifier Technologies also developed an interface offering the capability of dynamically querying and inserting video streams, to identify illegal content in real-time. Such applications are of interest to law enforcement (in particular counter terrorism efforts), but also of interest in the domain of multimedia analysis.

6.3 Future Work

While this thesis presents a complete picture of the NV-tree, as a scalable disk-based high-dimensional index, we nevertheless foresee significant future work in the domain. This work can largely be divided into two directions: further scalability extensions and further applications of the technology. Some of the application work was briefly mentioned above; we therefore focus on the scalability extensions here.

A key emphasis of the scalability efforts is the adaptation of the NV-tree to solid state disks (SSDs). This work, which is of high importance for being able to scale the database to even larger collection sizes while still maintaining high query and insertion throughput, is already in progress. In order to fully utilize the power of SSDs, the algorithm was rewritten, in particular the "leaf-group database" which is no longer required because individual descriptors can be easily ready from the collection file due to the superb small

random read capabilities of modern SSDs. One particular feature of SSD storage is that the maximum IO throughput, which is on the order of hundreds of thousands of IO operations per second through the PCI-express interface, can only be achieved through massive parallelization of the IO requests. Equipped with such hardware, our next scalability step is to measure collection sizes of 50–100 billion descriptors. First measurements have shown to be very promising; the index creation, in particular, was significantly faster—an important feature in scalability, as indexing time is a major bottleneck for reaching yet larger collection sizes.

Another focus is to experiment with alternative description schemes. On one hand, customers demand better robustness against affine transformations, in particular aspect-ratio changes which are only handled to some degree with traditional SIFT based features. On the other hand, as mentioned in the introduction, several recent description schemes exist that aggregate the information from a whole image or video frame into a single very compact, high-dimensional descriptor. These descriptors have many desirable properties: a) they generally provide good recognition power; b) while they are of a significantly higher dimensionality, they are only as many as the images or frames to be indexed, allowing more multimedia items to be represented inside main memory; and c) each query is only represented by a single query descriptor, helping to reduce query processing costs. In order to implement truly web-scale industry applications, it is necessary to implement content-based retrieval using these recent and sophisticated aggregated descriptors, but with a disk-based and ACID compliant indexing method, such as the NV-tree. This represents the most important future work in this area.

Bibliography

Amsaleg, L. (2014). *A database perspective on large scale high-dimensional indexing.* Habilitation à diriger des recherches, Université de Rennes 1.

Amsaleg, L., Jónsson, B. Þ., & Lejsek, H. (2015). *High-dimensional indexing in the real world: The case for a database perspective.* (Submitted to IEEE Transactions on Pattern Analysis and Machine Intelligence)

Andoni, A., & Indyk, P. (2005, June). *E^2LSH 0.1 - User Manual.* Massachusetts Institute of Technology.

Arandjelovic, R., & Zisserman, A. (2013). All about vlad. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 1578-1585). Portland, OR, USA: IEEE Computer Society.

Baluja, S., & Covell, M. (2006). Content fingerprinting using wavelets. In *Proceedings of the IET Conference on Multimedia* (p. 198–207). London England: The Institution of Engineering and Technology.

Bawa, M., Condie, T., & Ganesan, P. (2005). LSH forest: self-tuning indexes for similarity search. In *Proceedings of the ACM International Conference on World Wide Web* (p. 651-660). Chiba, Japan: ACM.

Bellmann, R. (1961). *Adaptive control processes: A guided tour*. Princeton, NJ, USA: Princeton University Press.

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is "nearest neighbor" meaningful? In *Proceedings of the International Conference on Database Theory* (p. 217-235). Jerusalem, Israel: Springer-Verlag.

Brown, M., & Lowe, D. G. (2002). Invariant features from interest point groups. In *Proceedings of the British Machine Vision Conference* (p. 656-665). Cardiff, Wales: British Machine Vision Association (BMVA).

Chang, E. Y. (2011). *Foundations of large-scale multimedia information management and retrieval: Mathematics of perception*. Berlin, Germany: Springer Verlag.

Chierichetti, F., Panconesi, A., Raghavan, P., Sozio, M., Tiberi, A., & Upfal, E. (2007). Finding near neighbors through cluster pruning. In *Proceedings of the ACM In-*

ternational Symposium on Principles of Database Systems (p. 103-112). Beijing, China: ACM.

Dađason, K., Lejsek, H., Jóhannsson, Á. Þ., Jónsson, B. Þ., & Amsaleg, L. (2010). GPU acceleration of Eff^2 descriptors using CUDA. In *Proceedings of the ACM International Conference on Multimedia* (p. 1167-1170). Firenze, Italy: ACM.

Datar, M., Indyk, P., Immorlica, N., & Mirrokni, V. (2006). *Locality-sensitive hashing using stable distributions*. Cambridge MA, USA: MIT Press.

Datta, R., Joshi, D., Li, J., & Wang, J. Z. (2008). Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys*, *40*(2), 5:1–5:60.

Douze, M., Jégou, H., Sandhawalia, H., Amsaleg, L., & Schmid, C. (2009). Evaluation of gist descriptors for web-scale image search. In *Proceedings of the ACM International Conference on Image and Video Retrieval* (pp. 19:1–19:8). Santorini, Greece: ACM.

Fagin, R., Kumar, R., & Sivakumar, D. (2003). Efficient similarity search and classification via rank aggregation. In *Proceedings of the ACM International Conference on Management of Data* (p. 301-312). San Diego, CA, USA: ACM.

Florack, L. M. J., ter Haar Romeny, B. M., Koenderink, J. J., & Viergever, M. A. (1994). General intensity transformation and differential invariants. *Journal of Mathematical Imaging and Vision*, *4*(2), 171-187.

Fraundorfer, F., Stewénius, H., & Nistér, D. (2007). A binning scheme for fast hard drive based image search. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 1 - 6). Minneapolis, MN, USA: IEEE Computer Society.

Fukunaga, K., & Narendra, P. M. (1975). A branch and bound algorithms for computing k-nearest neighbors. *IEEE Transactions on Computers*, *24*(7), 750 - 753.

Ge, T., He, K., Ke, Q., & Sun, J. (2014). Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *36*(4), 744 - 755.

Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases* (p. 518-529). Edinburgh, Scotland: VLDB Endowment.

Gray, J., & Graefe, G. (1997). The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, *26*(4), 63-68.

Gray, J., McJones, P. R., Blasgen, M. W., Lindsay, B. G., Lorie, R. A., Price, T. G., et al. (1981). The recovery manager of the System R database manager. *ACM Computing Surveys*, *13*(2), 223-243.

Gray, J., & Putzolu, F. (1987). The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the ACM*

International Conference on Management of Data (pp. 395–398). San Francisco, CA: ACM.

Guðmundsson, G. Þ. (2013). *Parallelism and distribution for very large scale content based image retrieval*. Unpublished doctoral dissertation, Université de Rennes 1.

Guðmundsson, G. Þ., Amsaleg, L., & Jónsson, B. Þ. (2012). Impact of storage technology on the efficiency of cluster-based high-dimensional index creation. In *Proceedings of the international conference on database systems for advanced applications* (p. 53-64). Busan, South Korea.

Guðmundsson, G. Þ., Jónsson, B. Þ., & Amsaleg, L. (2010). A large-scale performance study of cluster-based high-dimensional indexing. In *Proceedings of the VLS-MCMR workshop* (p. 31-36). Florence, Italy: ACM.

Heo, J., Lin, Z., & Yoon, S. (2014). Distance encoded product quantization. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 2139 - 2146). Columbus, OH, USA: IEEE Computer Society.

Hinneburg, A., Aggarwal, C. C., & Keim, D. A. (2000). What is the nearest neighbor in high dimensional spaces? In *Proceedings of the International Conference on Very Large Data Bases* (p. 506-515). Cairo, Egypt: VLDB Endowment.

Jain, P., Kulis, B., & Grauman, K. (2008). Fast image search for learned metrics. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 1 - 8). Anchorage, AK, USA: IEEE Computer Society.

Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *33*(1), 117 - 128.

Jégou, H., Perronnin, F., Douze, M., Sanchez, J., Pérez, P., & Schmid, C. (2012). Aggregating local image descriptors into compact codes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *34*(9), 1704 - 1716.

Jégou, H., Perronnin, F., Douze, M., Sánchez, J., Pérez, P., & Schmid, C. (2012, September). Aggregating local image descriptors into compact codes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *34*(9), 1704-1716.

Jin, Z., Hu, Y., Lin, Y., Zhang, D., Lin, S., Cai, D., et al. (2013). Complementary projection hashing. In *Proceedings of the IEEE International Conference on Computer Vision* (p. 257-264). Barcelona, Spain: IEEE Computer Society.

Joly, A., & Buisson, O. (2008). A posteriori multi-probe locality sensitive hashing. In *Proceedings of the ACM International Conference on Multimedia* (p. 209-218). Vancouver, BC, Canada: ACM.

Joly, A., Frélicot, C., & Buisson, O. (2003). Robust content-based video copy identification in a large reference database. In *Proceedings of the ACM International*

Conference on Image and Video Retrieval (p. 414-424). Urbana-Champaign, IL, USA: ACM.

Kalantidis, Y. S., & Avrithis, Y. (2014). Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 2329 - 2336). Columbus, OH, USA: IEEE Computer Society.

Katayama, N., & Satoh, S. (1997). The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM International Conference on Management of Data* (p. 369-380). Tucson, AZ, USA: ACM.

Ke, Y., Sukthankar, R., & Huston, L. (2004). Efficient near-duplicate detection and sub-image retrieval. In *Proceedings of the ACM International Conference on Multimedia* (p. 869-876). New York, NY, USA: ACM.

Kleinberg, J. (1997). Two algorithms for nearest-neighbour search in high dimensions. In *Proceedings of the ACM Symposium on Theory of Computing* (p. 599-608). El Paso, TX, USA: ACM.

Laptev, I. (2005). On space-time interest points. *International Journal on Computer Vision*, *64*(2-3), 107–123.

Lejsek, H. (2010). Video and image identification: Tools for supporting forensic investigations. In *Proceedings of the Multimedia in Forensics Workshop (MiFor).* Firenze, Italy: ACM. (Keynote lecture)

Lejsek, H., Ásmundsson, F. H., Daðason, K., Jóhannsson, Á. T., Jónsson, B. Þ., & Amsaleg, L. (2009). Videntifier™ Forensic: A new law enforcement service for automatic identification of illegal video material. In *Proceedings of the Multimedia in Forensics Workshop (MiFor).* Beijing, China: ACM.

Lejsek, H., Ásmundsson, F. H., Jónsson, B. Þ., & Amsaleg, L. (2006). Scalability of local image descriptors: A comparative study. In *Proceedings of the ACM International Conference on Multimedia.* Santa Barbara, CA, USA: ACM.

Lejsek, H., Ásmundsson, F. H., Jónsson, B. Þ., & Amsaleg, L. (2009). NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *31*(5), 869 - 883.

Lejsek, H., Jónsson, B. Þ., & Amsaleg, L. (2011). NV-Tree: Nearest neighbours at the billion scale. In *Proceedings of the ACM International Conference on Multimedia Retrieval* (pp. 54:1–54:8). Trento, Italy: ACM.

Lejsek, H., Jónsson, B. Þ., & Amsaleg, L. (2015). *NV-tree: Database solution for scalable nearest neighbor retrieval.* (Submitted to ACM Transactions on Database Systems)

Lejsek, H., Thormóðsdóttir, H., Ásmundsson, F. H., Daðason, K., Jóhannsson, Á. T., Jónsson, B. Þ., et al. (2010). Videntifier™ Forensic: Large-scale video identification in practise. In *Proceedings of the Multimedia in Forensics Workshop (MiFor).* Firenze, Italy: ACM.

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

Lindeberg, T. (1994). Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics*, *21*(2), 224–270.

Lindeberg, T. (1998). Feature detection with automatic scale selection. *International Journal on Computer Vision*, *30*(2), 77–116.

Liu, T. (2006). *Fast nonparametric machine learning algorithms for high-dimensional massive data and applications*. Unpublished doctoral dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

Liu, T., Moore, A., Gray, A., & Yang, K. (2004). An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of the Conference on Neural Information Processing Systems* (pp. 825–832). Vancouver, BC, Canada: Neural Information Processing Systems Foundation.

Liu, T., Rosenberg, C., & Rowley, H. (2007). Clustering billions of images with large scale nearest neighbor search. In *Proceedings of the IEEE Workshop on Applications of Computer Vision (WACV)* (p. 28-33). Austin, TX, USA: IEEE Computer Society.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Proceedings of the IEEE International Conference on Computer Vision* (p. 1150-1157). Corfu, Greece: IEEE Computer Society.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal on Computer Vision*, *60*(2), 91-110.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., & Li, K. (2007). Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases* (p. 950-961). Vienna, Austria: VLDB Endowment.

Mikolajczyk, K., & Schmid, C. (2003). A performance evaluation of local descriptors. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 257-263). Madison, WI, USA: IEEE Computer Society.

Mikolajczyk, K., & Schmid, C. (2005). A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*(10), 1615–1630.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (1992). ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, *17*, 94–162.

Moise, D., Shestakov, D., Guðmundsson, G. Þ., & Amsaleg, L. (2013). Indexing and searching 100M images with Map-Reduce. In *Proceedings of the ACM International Conference on Multimedia Retrieval.* Dallas, TX, USA: ACM.

Moise, D., Shestakov, D., Guðmundsson, G. Þ., & Amsaleg, L. (2013). Indexing and searching 100M images with Map-Reduce. In *Proceedings of the ACM International Conference on Multimedia Retrieval* (p. 17-24). Dallas, TX, USA: ACM.

Muja, M., & Lowe, D. G. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *36*(11), 2227 - 2240.

Nickerson, B. G., & Shi, Q. (2008). On k-d range search with patricia tries. *SIAM Journal of Computing*, *37*(5), 1373–1386.

Nistér, D., & Stewénius, H. (2006). Scalable recognition with a vocabulary tree. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 2161-2168). New York, NY, USA: IEEE Computer Society.

Ólafsson, A., Jónsson, B. Þ., Amsaleg, L., & Lejsek, H. (2011). Dynamic behavior of balanced NV-trees. *Multimedia Systems*, *17*(2), 83-100.

Paulevé, L., Jégou, H., & Amsaleg, L. (2010). Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, *31*(11), 1348-1358.

Petitcolas, F. A. P., Steinebach, M., Raynal, F., Dittmann, J., Fontaine, C., & Fates, N. (2001). A public automated web-based evaluation service for watermarking schemes: StirMark benchmark. In *Proceedings of electronic imaging, security and watermarking of multimedia contents iii* (p. 575-584). San Jose, CA, USA: SPIE.

Philbin, J., Chum, O., Isard, M., Sivic, J., & Zisserman, A. (2007). Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 1 - 8). Minneapolis, MN, USA: IEEE Computer Society.

Philbin, J., Chum, O., Isard, M., Sivic, J., & Zisserman, A. (2008). Lost in quantization: Improving particular object retrieval in large scale image databases. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 1 - 8). Anchorage, AK, USA: IEEE Computer Society.

Robinson, J. (1981). The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM International Conference on Man-*

agement of Data (p. 10-18). Ann Arbor, MI, USA: ACM.

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

Shestakov, D., Moise, D., Guðmundsson, G. Þ., & Amsaleg, L. (2013). Scalable high-dimensional indexing with Hadoop. In *International Workshop on Content-Based Multimedia Indexing* (p. 207 - 212). Veszprém, Hungary.

Sun, X., Wang, C., Xu, C., & Zhang, L. (2013). Indexing billions of images for sketch-based retrieval. In *Proceedings of the ACM International Conference on Multimedia* (p. 233-242). Barcelona, Spain: ACM.

Tao, Y., Yi, K., Sheng, C., & Kalnis, P. (2009). Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the ACM International Conference on Management of Data* (p. 563-576). Boston, MA, USA: ACM.

Torralba, A., Fergus, R., & Freeman, W. T. (2008). 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *30*(11), 1958 - 1970.

Uhlmann, J. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, *40*(4), 175-179.

Wang, J., Kumar, O., & Chang, S. (2010). Semi-supervised hashing for scalable image retrieval. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition* (p. 3424 - 3431). San Francisco, CA, USA: IEEE Computer Society.

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

Weiss, Y., Torralba, A., & Fergus, R. (2008). Spectral hashing. In *Proceedings of the Conference on Neural Information Processing Systems* (p. 1753-1760). Vancouver, BC, Canada: Neural Information Processing Systems Foundation.

Xioufis, E. S., Papadopoulos, S., Kompatsiaris, Y., Tsoumakas, G., & Vlahavas, I. P. (2014). A comprehensive study over VLAD and product quantization in large-scale image retrieval. *IEEE Transactions on Multimedia*, *16*(6), 1713 - 1728.

Zäschke, T., Zimmerli, C., & Norrie, M. C. (2014). The ph-tree: A space-efficient storage structure and multi-dimensional index. In *Proceedings of the ACM International Conference on Management of Data* (p. 397-408). Snowbird, UT, USA: ACM.

Zhang, D., Agrawal, D., Chen, G., & Tung, A. (2011). Hashfile: An efficient index structure for multimedia data. In *Proceedings of the IEEE International Conference on Data Engineering* (p. 1103 - 1114). Hannover, Germany: IEEE Computer Society.

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539