REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

NV-tree: An Efficient Disk-Based Index for Approximate Search in Very Large High-Dimensional Collections

Herwig Lejsek, Friðrik Heiðar Ásmundsson,
Björn Þór Jónsson, Laurent Amsaleg

RUTR-CS07001 — July 2007

Reykjavík University - School of Computer Science

Technical Report

ISSN 1670-5777

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

NV-tree: An Efficient Disk-Based Index for Approximate Search in Very Large High-Dimensional Collections

Herwig Lejsek*, Friðrik Heiðar Ásmundsson*,
Björn Þór Jónsson*, Laurent Amsaleg†

Technical Report RUTR-CS07001, July 2007

Abstract: Over the last two decades, much research effort has been spent on nearest neighbour search in high-dimensional data sets. Most of the approaches yet published have, however, only been tested on rather small collections. When large collections have been considered, high-performance environments have been used, in particular systems with a large main memory. Accessing data on disk has largely been avoided, because disk operations are considered to be too slow. It has been shown, however, that using large amounts of memory is generally not an economic choice. Therefore, we propose the NV-tree, which is a very efficient disk-based data structure that can give good approximate answers to nearest neighbour queries with a single disk operation, even for very large collections of high-dimensional data. Using a single NV-tree, the returned results have high recall, but contain a number of false positives. By combining two or three NV-trees, most of those false positives can be avoided while retaining the high recall. Finally, we compare the NV-tree to Locality Sensitive Hashing, a popular method for ϵ-distance search. We show that they return results of similar quality, but the NV-tree uses many fewer disk reads.

Keywords: High-dimensional indexing; Very large databases; Approximate searches.

(Útdráttur: næsta síða)

The Eff[2] project on *Efficient and Effective Image Retrieval* (see http://datalab.ru.is/eff2) is a cooperation between researchers at Reykjavík University, Iceland, and the IRISA laboratory in Rennes, France. This work was partially supported by Icelandic Research Fund grant 060036021, INRIA Eff[2] Associate Teams grant, and an ÉGIDE Jules Verne travel grant. We wish to thank Morgunblaðið for the use of their large picture collection, and the authors of LSH and SIFT for giving us access to their implementations.

* Reykjavík University, Kringlan 1, IS-103 Reykjavík, Iceland. herwig/fridrik01/bjorn@ru.is
† IRISA–CNRS, Campus de Beaulieu, 35042 Rennes, France. laurent.amsaleg@irisa.fr

HÁSKÓLINN Í REYKJAVÍK
REYKJAVÍK UNIVERSITY

NV-tréð: Hraðvirkur diskamiðaður vísir fyrir nálgunarleit í stórum margvíðum gagnasöfnum

Herwig Lejsek, Friðrik Heiðar Ásmundsson,
Björn Þór Jónsson, Laurent Amsaleg

Tækniskýrsla RUTR-CS07001, Júlí 2007

Útdráttur: Á síðustu tveimur áratugum hefur mikil vinna verið lögð í leit að næsta nágranna í margvíðum gagnasöfnum. Flestar birtar aðferðir hafa þó aðeins verið prófaðar á fremur smáum gagnasöfnum. Þegar stór söfn hafa verið skoðuð, hefur sérlega öflugur vélbúnaður verið notaður, einkum kerfi með stóru innra minni. Markmiðið hefur verið að forðast diskavinnslu, þar sem hún hefur verið talin of hægvirk. Það hefur þó verið sýnt fram á að notkun mjög mikils innra minnis er ekki hagkvæm. Þess vegna setjum við fram NV-tréð, sem er mjög hraðvirk diskamiðuð gagnagrind sem gefur góða nálgun við næsta nágranna með einum diskalestri, jafnvel í mjög stórum söfnum af margvíðum gögnum. Með einu NV-tré fást niðurstöður með góðum skilum, en þær innihalda einnig mörg röng svör. Með því að nota tvö eða þrjú NV-tré má sía burt flest þessara röngu svara án þess að fækka réttu svörunum. Að lokum berum við NV-tréð saman við LSH, sem er vinsæl hökkunaraðferð fyrir ϵ-fjarlægðar fyrirspurnir. Við sýnum að báðar aðferðir geta skilað álíka góðum svörum, en að NV-tréð notar miklu færri diskalestra.

Lykilorð: Margvíðir vísar; Mjög stór gagnasöfn; Nálgunarleit.

(Abstract: previous page)

Contents

1 Introduction **1**
1.1 Previous Large-Scale Studies 1
1.2 Contribution of this Paper 2

2 The NV-tree **2**
2.1 NV-tree Creation 3
2.2 NV-tree Search 3
2.3 NV-tree Data Structure 4
2.4 Projection Strategies 4
2.5 Partitioning Strategies 5
2.6 Insertions and Deletions 7

3 Experimental Setup **7**
3.1 Descriptors and Queries 7
3.1.1 Descriptor Collection 7
3.1.2 Query Workload 7
3.2 Hardware and Software Configuration 8
3.3 Result Quality Metrics 8
3.3.1 Recall 9
3.3.2 Merging Result Sets 9

4 Analysis of Query Results **10**
4.1 Distance and Result Quality 11
4.2 Contrast and Result Quality 12
4.3 Discussion 13

5 Query Experiments **13**
5.1 Experiment 1: A Single NV-tree 14
5.1.1 Recall 14
5.1.2 False Positives 15
5.2 Experiment 2: Additional NV-trees 15
5.2.1 Performance and Recall 16
5.2.2 False Positives 16
5.3 Discussion 17

6 Locality Sensitive Hashing **18**
6.1 The Concept of LSH 18
6.2 Adapting LSH to Disk 19
6.3 Experimental Evaluation 21
6.4 Discussion 22

7 Conclusions & Future Work **23**

1 Introduction

The applications of nearest neighbour search in high-dimensional space are very diverse, and include context based image retrieval, finding correlations in stock data and searching for similar chemical structures. Nearest neighbour 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.

Most of the approaches yet published, however, have only been tested on rather small collections ranging from tens of thousands of descriptors to a few million (e.g., see [BBK98, WSB98, LCGMW02, BAG03, FKS03]) and some have been explicitly shown not to work well at high-dimensions or large scale [AG01, LÁJA05]. Only a handful of studies have considered very large descriptor collections. In all such large-scale studies, however, accessing data randomly on disk has been avoided, because random disk operations have been considered to be too slow.

1.1 Previous Large-Scale Studies

Liu has studied the use of a distributed hybrid Spill-Tree, a variant of the Metric-Tree [Uhl91], for a collection of 1.5 billion global descriptors [Liu06]. In that study two thousand workstations were used, presumably having at least a terabyte or two of total main memory. In general, however, Gray has shown [GP87, GG97] 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.

Locality Sensitive Hashing (LSH) by Indyk et al. [GIM99, DIIM06], has also been considered for large-scale retrieval. LSH is based on the concept of projecting descriptors onto a random line and classifying the locations along this line with different symbols. Doing such projections for many lines, LSH concatenates the symbols to a fingerprint for this specific descriptor. This fingerprint has the property that descriptors, which lie within a fixed ϵ-distance threshold, generate with high likelihood the same fingerprint. By storing all these fingerprints in a hash table it is then possible to retrieve similar descriptors with a single disk read.

Ke et al. [KSH04] studied the use of LSH for a local descriptor scenario. They used LSH to yield a number of potentially matching descriptors, and then scanned the descriptor collection on disk to calculate the precise result. While sequentially scanning the collection may work in a high-throughput scenario, it yields very poor response times.

Joly et al. [JBF07] studied an application with 1.5 billion 20-dimensional local descriptors. They also completed processing with a sequential scan.

1.2 Contribution of this Paper

In this paper we propose the NV-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. This paper makes several major contributions:

- First, we describe the fundamentals of the NV-tree, as well as different strategies for its construction.
- Second, we analyse the properties of a large-scale copy detection application using the well known 128-dimensional SIFT descriptors [Low04]. We show that the SIFT descriptors are very distinctive, and have high contrast, even in collections of 180 million data points. Furthermore, we show that an ϵ-based search cannot expect to retrieve good results for all queries.
- Third, we analyse the performance of the NV-tree and show that the NV-tree works well for such "meaningful" high-dimensional data. 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 LSH, which is currently the most popular high-dimensional indexing method. We show that, a) LSH can actually be used without subsequently performing a sequential scan of the descriptor collection, making LSH more competitive than previously thought, but b) the NV-tree still returns results of similar quality using many fewer disk reads.

The remainder of this paper is organized as follows. First, we present the NV-tree in Section 2. Then we present the collection and workload used in our experiments in Section 3. In Section 4, we analyze the ground-truth result quality of our workload, and in Section 5 we describe the performance of the NV-tree. In Section 6 we compare the performance of the NV-tree to that of LSH. We conclude in Section 7.

2 The NV-tree

The NV-tree (Nearest-Vector-tree) is a disk-based data structure designed to provide efficient approximate nearest neighbour search in very large high-dimensional collections. In essense, it transforms costly nearest neighbor searches in the high-dimensional space into efficient uni-dimensional B^+-tree 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 or "clusters" which can be easily fetched from disk with a single disk read,

and which are highly likely to contain all the close neighbours in the collection.[1] Since, in a very high-dimensional space, such "clusters" may overlap, 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 a higher storage requirement, but given the low cost of disk space this is a very good tradeoff.

In this section we first outline the NV-tree creation (Section 2.1) and search (Section 2.2) algorithms, before describing the data structure in detail (Section 2.3). Then we consider strategies for projections (Section 2.4) and partitioning (Section 2.5). Finally, we describe briefly insertion to, and deletion from, the NV-tree (Section 2.6).

2.1 NV-tree Creation

When the construction of an NV-tree index 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 2.4.

The projected values are then partitioned into disjunct sub-partitions based on their position on the projection line. In addition, overlapping sub-partitions are created for redundant coverage of partition borders. Strategies for partitioning are described in Section 2.5.

This process of projecting and partitioning is repeated for all the new sub-partitions using a new projection line at each level. The process stops when the number of descriptors in a sub-partition falls below a specified limit designed to be disk I/O friendly. At that time, a *leaf partition* is written to disk, containing the descriptor identifiers of the sub-partition ordered by their rank along the projection line of the lead node.

Overall, an NV-tree consists 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) leaf nodes, which are stored on disk and contain the references to the actual descriptors.

2.2 NV-tree Search

During query processing, the query descriptor first traverses the intermediate nodes of the NV-tree. At each level of the tree, the query descriptor is projected to 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 partition.

The leaf partition is then fetched into memory and the query descriptor is projected onto the projection line of the leaf partition. Then the search returns the k descriptor identifiers which are closest to that projection.

[1] Note that dealing with data sets with strongly clustered data and subsequently large result sets is beyond the context of this paper; it is likely that the NV-tree will not cope well with such applications.

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.

2.3 NV-tree Data Structure

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 the following four arrays:

Child: This array points to the child nodes of the intemediate node. The child nodes may in turn be intermediate nodes or leaf nodes.

PartitionBorder: This array keeps track of the borders of each partition, including the overlapping partitions.

SearchBorder: This array keeps track of the half-way values of projections between the boundaries of adjacent partitions. These values are used to direct the search to the sub-partition with center-point closest to the projection of the query descriptor.

ProjectionLine: This array stores a pointer to the description of the projection line of each sub-partition.

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.

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 a B^+-tree which stores (*projected value*, *descriptor identifier*) pairs. Upon creation, leaf nodes are typically filled to 67% of capacity, leaving room for insertions (see Section 2.6).

The B^+-tree of the leaf nodes can either be dense or sparse. By storing a sparse B^+-tree 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 optimisation, however, we observed next to no influence on the result quality. Our implementation therefore typically only stores every 16th projected value; this setting is used throughout the paper.

2.4 Projection Strategies

Projecting high-dimensional data points to random lines has been introduced by Kleinberg [Kle97] and subsequently used in several other high-dimensional indexing techniques [FKS03, FKM$^+$04, DIIM06, LMGY04]. 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 the high-dimensional space 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 *Random* strategy picks random lines, as described above; this strategy is simple and data independent.

Retrieval quality, however, can be improved with data-dependent generation of lines, for example using the well known Principal Component Analysis (PCA) [Pea01]. 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 very 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 creation of the NV-tree, a large set of isotropic, quasi-orthogonal random lines is generated (typically 1,000) and kept in a *line pool* in main memory. During line selection, the partition about to be projected is first sampled with a small sample. The data points in this 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 efficient process several times (the efficiency stems from the small sample sizes) a single line is finally elected and subsequently used as the projection line of the partition.

Instead of choosing the single best possible line for the partition, determined by costly PCA calculations, this process picks a "reasonably good" line from the large line pool by using many cheap projection calculations over small samples. Overall, we have observed that the *Approximate PCA* strategy outperforms the *Random* strategy, and that the larger the pre-computed line set is the better the results, although improvements are barely noticable beyond 1,000 lines.

2.5 Partitioning Strategies

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

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

In general, we have observed that the projections of data points onto a line typically follow a normal distribution. As a result, the absolute distance between partition boundaries vary 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 this strategy, we calculate the standard deviation s_d and mean m of the projections along the projection line. Then a parameter α is used to determine the partition borders as $\ldots, m - 2s_d\alpha, m - s_d\alpha, m, m + s_d\alpha, m + 2s_d\alpha, \ldots$; overlapping partitions are created similarly. Small adjacent sub-partitions are merged until the resulting cardinality hits the leaf partition size limit and then written to disk. Sub-partitions containing many data points, on the other hand, are subsequently re-partitioned.

Typically, 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 single 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 have devised the *Hybrid* strategy. This strategy generally follows the *Unbalanced* strategy until a sub-partition is of a size that could fit in l number of leaf partitions (we have found $l = 6$ to be a good number). Then the *Balanced* strategy is used to construct the leaf partitions. As a result, leaf partitions are better utilized and the tree is shallower, both of which result 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 typically occupy less disk space but may yield lower recall.

2.6 Insertions and Deletions

The NV-tree handles updates as descriptors can be inserted into or deleted from an existing tree. New descriptors are stored in a temporary holding area which is organized by partitions and included in the search process. When the holding area fills, it is written to the index. On overflow, partitions are split into new sub-partitions and the tree structure is updated accordingly. Note that new descriptors must be inserted to several partitions, due to data redundancy.

A list of deleted descriptors is maintained and used to filter search results. Deletion is propagated to disk opportunistically when a partition holding the deleted descriptor is fetched. A reference count is maintained to remove the deleted descriptor from the list once all its redundant occurrences have been removed from disk.

3 Experimental Setup

In this section, we describe the experimental environment used in this paper. First, we describe the descriptor collection and query workload used in all the experiments (Section 3.1). Then, we describe the hardware and software configurations used (Section 3.2). Finally, we describe the result quality metrics studied in our analysis (Section 3.3).

3.1 Descriptors and Queries

In this study we use the well-known SIFT (Scale Invariant Feature Transform) method proposed by Lowe [Low99, Low04], which is *the* 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 an 128 dimensional vector, which is normalised to a length of 512.

3.1.1 Descriptor Collection

The descriptor collection was obtained by extracting local features from an archive of about 150 thousand high-quality press photos and consisted of 179,443,881 SIFT descriptors. 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.

3.1.2 Query Workload

As pointed out by Beyer et al. [BGRS99], it is necessary to use a meaningful query workload in order to draw any meaningful conclusions from the results. Using descriptors from the collection or descriptors from other images is not meaningful, as the SIFT descriptors are designed for copy detection. We have therefore focused on a copy detection workload, where modified copies of images from the collection are used to create query descriptors.

As in [LÁJA06], we created several transformed versions of images from our collection, using the Stirmark benchmarking tool [Pet01]. We then randomly selected 500,000 individual query descriptors from the transformed images. The transformations include rotation, rescaling, cropping, affine distortions and convolution filters. It has been shown [Low04, LÁJA06] that the SIFT descriptors cope rather well with most of these distortions at an image level, meaning that a good 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. Note, however, that the transformations also include distortions which the SIFT descriptors have been shown not to handle well [LÁJA06].

3.2 Hardware and Software Configuration

All experiments 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.

For all experiments we used the following NV-tree configuration:

- We used the *Hybrid* partitioning strategy and set the leaf partition size to six disk pages (24 KB). The hybrid strategy generally yields about 5% better results than a balanced strategy, but equivalent to the unbalanced, while only requiring about 20% more space than the balanced partitioning strategy.
- We used the *Approximate PCA* best lines strategy and generated an initial line pool of 1000 lines, where each pair of lines has a minimum angle of 72 degrees. This strategy generally yields about 10% better results than random lines.
- We used leaf nodes with sparse indices, which generally cuts the index size in half (by reducing the height of the tree) without affecting result quality.
- We retrieved 1,000 descriptors from each NV-tree (in one experiment we vary this number).

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 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 and not explicitly managing file locations.

3.3 Result Quality Metrics

In a copy detection workload, at most one descriptor can be considered a correct match to any query (in the case of a non-copy, there is no correct match). The remaining results

should be considered false positives. In general, we desire high recall with a small number of false positives.

3.3.1 Recall

In a controlled environment, the single correct match to any descriptor is easily identified. An interesting question, however, is whether a general quality metric can be constructed that determines those correct matches well. In this paper we consider the following three general metrics for recall quality:

Rank-Based Results: We can consider the results of a k-nearest neighbour query as a result metric. The key question then is how to pick k such that recall is high, without incurring many false positives.

ϵ-Based Results: We can also consider the results of an ϵ-range query as a result metric. The key question is then how to pick ϵ such that recall is high, without incurring many false positives.

Contrast-Based Results: According to Beyer et al. [BGRS99] a nearest neighbour must be significantly closer to the query point than most of the other points in the dataset in order to be considered meaningful. Lowe proposes to consider the nearest neighbour n_1 a meaningful result, when $d(n_2, q)/d(q, n_1) > 1.8$ [Low04]. Since we are working with collections which are several orders of magnitude larger than that used by Lowe, we prefer to look at the hundredth neighbor. We say that returned neighbor n_i is meaningful with respect to contrast c (default value of c is 1.8) when $d(n_{100}, q)/d(q, n_i) > c$.

In Section 4 we analyse the ground-truth results obtained from a sequential scan and show that selecting k or ϵ such that either of the first two approaches gives good results is impossible, and that only the contrast based metric gives a good indication of actual result quality.

3.3.2 Merging Result Sets

In order to be able to use the contrast-based quality metric above, it is important to produce a ranking of the query results. With the NV-tree, even a single index can produce a ranking without looking at the actual descriptors. With LSH, which is set-based, this is not possible.

The primitive method used by many nearest neighbour methods (such as LSH) is to load the entire descriptors from the result set into memory, calculate the actual distance to the query descriptor and rank/filter based on this distance. Unless the whole descriptor set fits in memory, however, this solution is far too expensive solution as it requires a random I/O operation for each candidate descriptor loaded from disk. The alternative approach of storing the actual descriptors inside each partition was already dismissed by Ke et al. [KSH04], as it increases the disk I/O requirements by several orders of magnitude.

The conventional solution is batch processing (e.g., see [KSH04, JFB03]). After several queries have been issued and their result lists have been retrieved the descriptor file is

scanned sequentially and all the result lists are ranked/filtered. This is acceptable when throughput is more important than response time and the collection is small. For our descriptor collection, however, the response time would be on the order of minutes and growing linearly with collection size. For many practical applications, batch processing is inacceptable as scanning time grows linearly with collection size.

We propose two simpler solutions, which are based on the concept of rank aggregation and which are applicable when more than two indices are used (or more than two hash-tables, in the case of LSH). When the results are set-oriented (with no inherent rank, as in LSH), the descriptors that occur in the most indices are considered the most relevant; descriptors that occur in only one index are then omitted.

When the results are rank-oriented (such as with the NV-tree), approaches such as median rank aggregation can be used, in addition to the above method. With median rank aggregation, the descriptors that are close to the query descriptor in more than half of the indices are considered closest to the query descriptor.

Note, however, that we aggregate over two or more result sets from different indices which are largely disjunct, and may even be completely disjunct (this might, for example, occur when no close neighbour exists). Therefore no guarantee can be given that an aggregated search of two or more indices returns any neighbours at all. Both approaches described above work well, however, for filtering false positives from the result set returned by more than one index, as the likelyhood that random false positives appear in the result sets of two or more indepedent indices is low.

4 Analysis of Query Results

Lowe has analyzed a small collection of SIFT descriptors and shown that it has high contrast [Low04]. We are not aware, however, of such analysis of any large scale collection of SIFT descriptors; this is the goal of this section.

We have taken the descriptor collection and query workload described in Section 3.1 and run a sequential scan to calculate the 1,000 nearest "ground truth" neighbors for each query descriptor. As there are 500,000 query descriptors, this yields 500 million neighbors in all. Recall that for each query descriptor of this workload, precisely one descriptor in the collection is the correct match; the remainder should be considered false positives. In our collection, 248,852 query descriptors found a correct match among the top 1,000 neighbors, which is a recall of slightly less than 50%. While this may at first seem low, it is still a good recognition performance considering that some query descriptors originated from severely modified images [LÁJA06].

In the remainder of this section, we analyse the distance distribution of both the correct matches and the false positives. In the process, we show that: a) the collection exhibits good contrast and therefore the workload is "meaningful" as defined by Beyer et al. [BGRS99]; b) the correlation between contrast and absolute distance to nearest neighbor is surprisingly weak, which indicates that ϵ-based search is not appropriate for high-quality results; and

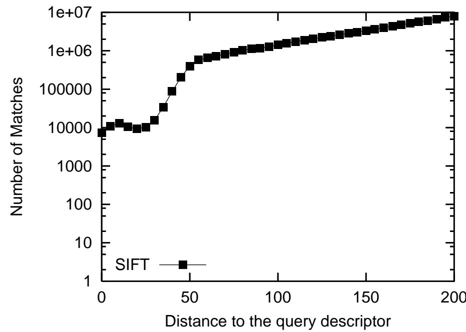

Figure 1: The distribution of false positives based on distance to the query descriptor.

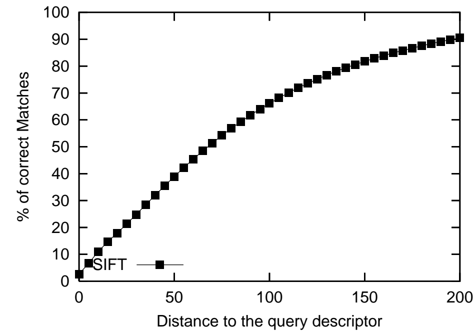

Figure 2: The cumulative distribution of correct matches based on distance to the query descriptor.

c) in order to obtain such high-quality results, the search index must support ranking of results, which is a key strength of the NV-tree.

4.1 Distance and Result Quality

We start with an analysis of how the absolute distance between the query descriptor and returned neighbours affects the result quality. Figure 1 shows the distribution of all 500 million neighbours depending on the distance of each neighbour to the query descriptor. The x-axis shows the absolute distance (corresponding to varying ϵ), while the y-axis shows the number of neighbours with approximately that distance. We observe that the number of descriptors stays rather uniform and small at the beginning (recall that these neighbors answer 500,000 queries). Once the distance surpasses a threshold of 25, however, we can see an exponential increase in the number of neighbors at each distance (note the logarithmic scale). Recall that in our application almost all of these descriptors are false positives.

Figure 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.

Taking a closer look at the individual results we observe that when correct matches appear among the 1,000 nearest neighbours, they are in most cases ranked first and are at a much closer distance than the prevailing number of other neighbours, regardless of their absolute distance.

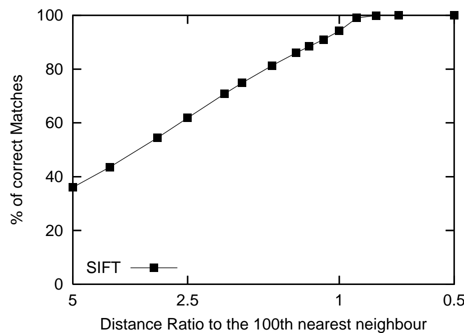

Figure 3: The cumulative distribution of correct matches based on contrast.

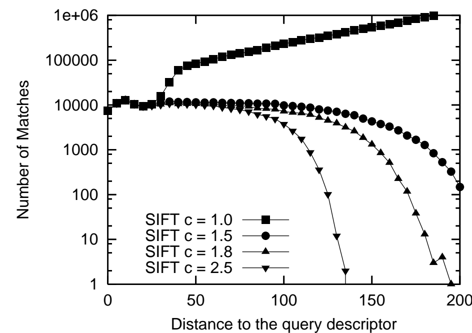

Figure 4: Distribution of false positives by distance to query descriptor, for various contrast thresholds.

All those observations show that it is impossible to select a global ϵ that gives results of high quality for this application, and that a recall metric based on ϵ-distances can just give a vague indication of result quality for a nearest neighbour search method, as it does not consider whether the nearest neighbours retrieved are actually meaningful or not.

4.2 Contrast and Result Quality

Figure 3 shows an analysis of the correct matches based on different thresholds of the contrast criteria. 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 3.1 as $d(n_{100}, q)/d(q, n_i) > c$. The figure shows that 36% of the correct matches are more than five times closer in distance than the 100th nearest neighbour in the result list. For $c = 1.8$, which is the value that Lowe recommended, about 75% of the correct matches 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 positives.[2]

Figure 4, on the other hand, shows the effects of the contrast filtering 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 1 shows an exponential increase in the number of descriptors (similar behavior to Figure 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.

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

Comparing Figures 3 and 4, we see that choosing a higher contrast threshold results in lower recall but fewer false positives, and vice versa. Comparing these to Figures 1 and 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,[3] which means that about 75% of the descriptors returned are correct matches and about 25% are false positives. This ratio proves the distinctiveness of the SIFT descriptors, and shows that we can expect very small and meaningful result sets for nearly all query descriptors.

4.3 Discussion

The discussion above shows that there is clearly some correlation between the absolute distance of a nearest neighbour and the contrast ratio, as the farther away a neighbour is from the query descriptor, the less likely it is that the contrast ratio is higher than the selected threshold. We have shown, however, that this correlation is much weaker than previous studies have assumed.

It is therefore clear that recall metrics based on ϵ-distances are not suitable for measuring the quality of a real nearest neighbour search. Essentially, the problem is that while the ϵ is necessarily a global parameter, the nearest neighbors can sometimes be close and sometimes far, so the contrast depends very much on the local distribution of data. Therefore, recall metrics based on a contrast ratio are much better applicable to measuring the quality of the results.

Finally, we conclude that, in order to be effective, high-dimensional indices must support the contrast criterion well. We observe that LSH, due to its ϵ-based hashing and set based query processing, does not. The NV-tree, on the other hand, does support the contrast criterion well through its use of rank-based partitioning and query processing. This will be further demonstrated in Sections 5 and 6.

5 Query Experiments

In this section we present two experiments which demonstrate key properties of the NV-tree. In Section 5.1 we discuss an experiment with a single NV-tree index, which shows that the NV-tree yields high recall, especially with close neighbors. In Section 5.2 we then discuss an experiment with up to three NV-trees, which demonstrates that with such configurations false positives can be largely eliminated, while keeping most of the high recall.

For both experiments we use the NV-tree configuration described in Section 3.2. We created a total of three indices for these experiments. The index creation took about 16 hours per index, and each index required about 50 GB of disk space (about twice the size of the collection).

[3] The fact that this number is similar to the number of correct matches is purely by coincidence.

5.1 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. Such a single descriptor query takes on average 12.5 ms, 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.

5.1.1 Recall

Turning to the result quality, Figure 5 shows the recall of the search for various level of contrast filtering. The x-axis shows the distance from the retrieved neighbors to the query descriptors, and the y-axis shows the portion of meaningful neighbors in each distance category. For this experiment the values on the y-axis are computed by checking the 1,000 nearest neighbors returned by the NV-tree against the results obtained by a sequential scan having various contrast thresholds; since the NV-tree does not store distance information, it is not possible to calculate contrast directly on the NV-tree results.

Let us examine first the results for $c = 1.0$, where the top 100 returned descriptors are included in the answer. Overall, with this setting, descriptors which are closer to the query descriptor than 25 are always found. For larger distances, the recall drops very significantly. Recalling, however, the corresponding line from Figure 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 lines for $c \in \{1.5, \ldots, 2.5\}$, we observe that descriptors fulfilling these contrast criteria have much higher recall than general near neighbors. 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 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 in fact finding most of the meaningful neighbors. In fact, the NV-tree is able to find 65.8% of all meaningful neighbors, which is only slightly lower than the 75% found by the costly sequential scan.

Interestingly, the contrast threshold does not affect quality in the range from 0 to 100, because the NV-tree supports the contrast based quality metric very well and finds most of the meaningful neighbors. A few 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 the papers we use $c = 1.8$, as proposed by Lowe.

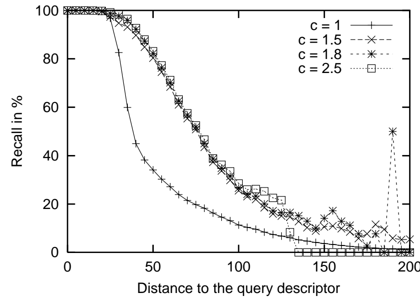

Figure 5: Recall for a single NV-tree retrieving 1000 Nearest Neighbors per query.

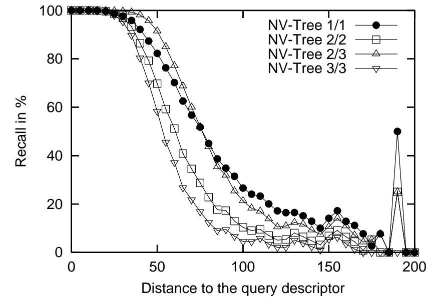

Figure 6: Recall by aggregating the result lists of two or three independent NV-trees.

5.1.2 False Positives

Turning to the number of false positives, we observe that when 1,000 nearest neighbors are retrieved, at least 999 of them are false positives (see Section 3.3). 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, there are no means to filter out the false positives. For a local descriptor application, where many query descriptors "vote" on the similarity of a single image (e.g., see [LÁJA06, KSH04]), such false positives are easily tolerated as they tend to be distributed among many non-relevant images. For user-oriented, single descriptor applications, however, only a handful of false positives can be tolerated; while a user can easily scan, say, six to ten results, several hundreds are definitely too many. While it is possible to request only a handful of nearest neighbors from the single NV-tree, this 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.

5.2 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 [FKS03] can be used to combine the two ranked lists from the two indices. While a precise description of median rank aggregation is outside the scope of this paper, it 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 the retrieval performance and recall, and then focus on the reduction of the false positives.

5.2.1 Performance and Recall

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

Figure 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. As the figure shows, the overall shape of the recall curves is similar when using more indices, The 2/2 and 3/3 strategies always perform worse than 1/1. This is due to the fact that it is more difficult for an indexed descriptor to land in two or three corresponding partitions by chance, than it is to land in the appropriate partition of a single NV-tree.

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 remainder of this section we therefore focus on the 2/3 strategy.

5.2.2 False Positives

The major motivation for doing search on more than one NV-tree, however, was not to obtain higher recall but to filter out the high 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 before. This time, however, only a few such "aggregated" neighbors are retrieved; we even consider retrieving only a single such neighbor. The expectation 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 of the false positives.

Figure 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. 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 very 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 7, we see that recall is improved significantly when going from one to two aggregated neighbors. By returning just one aggregated neighbour, we obtain a very reasonable recall of more than

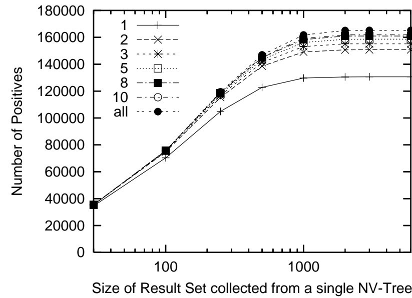

Figure 7: Positives of 2/3 NV-trees based on number of neighbors retrieved.

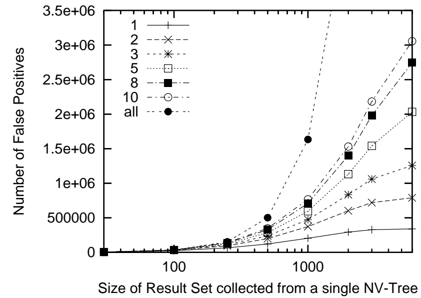

Figure 8: False Positives of 2/3 NV-trees based on number of neighbors retrieved.

130,000 meaningful neighbours. By returning two, recall improves to over 155,000 and with 10 aggregated neighbors we reach over 160,000 meaningful neighbours. Larger results sets achieve only minor improvements, but as we will see in a moment they increase the number of false positives significantly.

Figure 8 shows the number of false positives seen in 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 number of 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 neighbours, and 2,5% even more than 100 neighbours; these query descriptor are clearly landing in very dense areas.

Combining the results shown in Figures 7 and 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 120,000 descriptors, respectively. On the other hand, with these configurations, the false positives drop again by another order of magnitude. For the 3/3 setup collecting a maximum of five neighbours at a result set size of 1000 gives only 16,000 false positives with a recall of 119,500 meaningful neighbours. Therefore, if false positives must be reduced at all costs, then this setup is the right choice.

5.3 Discussion

Overall, these experiments show that the NV-tree is a very good data structure for approximate nearest neighbour 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 we get more than 99% recall for a radius of 25 or smaller. For neighbours beyond this distance the detection rate drops significantly, but overall about two thirds of the meaningful neighbors are found. In fact, a very benefical attribute of the NV-tree is that it finds meaningful neighbours with much higher probability than general ones. Using a single NV-tree, the returned results thus have high recall, but contain a high number of false positives. By combining three NV-trees, those false positives can largely be eliminated while retaining most of the high recall.

6 Locality Sensitive Hashing

In this section we discuss Locality Sensitive Hashing (LSH) which we believe to be the most competitive method to our proposed NV-tree. In Section 6.1 we describe the algorithm behind Local Sensitive Hashing. Section 6.2 presents our adaptation of LSH to a disk-based setting and explains how the various parameters affect performance and quality of the search. In Section 6.3 we compare LSH to the NV-tree. We conclude with a discussion in Section 6.4.

6.1 The Concept of LSH

Unlike most other nearest neighbour 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 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 in [GIM99] and then later extended to l_p norm in [DIIM06]. 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 to a set of k random lines through the search space. As in the NV-tree, 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 descriptors referenced by the same fingerprint, they are hashed via a standard hash function into a hash table. Since LSH does not apply overlapping

[4] A disk-based strategy was developed by Ke et al. [KSH04]. 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.

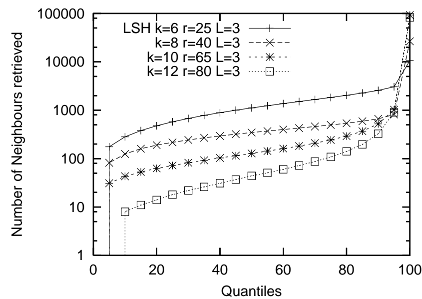

Figure 9: Distribution of result set size for LSH with three hash tables ($L = 3$).

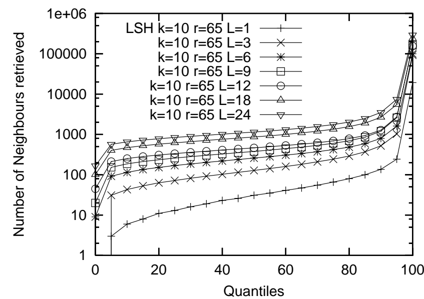

Figure 10: Distribution of result set size for LSH with varying number of hash tables ($k = 10$, $r = 65$).

and the likelihood of separating two close neighbours also increases with the fingerprint length k, it needs several of these 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 evaluated, all descriptors in the result set are sorted according to their distances to q and returned.

6.2 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. Our descriptor collection consumes about 22 GB, however, making this approach impossible. Furthermore, keeping the collection on disk and performing a random disk read to fetch each descriptor in the result is also unacceptable. Instead, for filtering false positives we propose to count the number of occurrences of each descriptor in the result sets from all the hash tables and to rank the result accordingly. Close neighbors are likely to be found by many hash functions, and their occurence count will therefore be high.

For our experimental evaluation we adapted the original LSH implementation [AI05] 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 optimising the LSH algorithm. We can, however, make a reasonably 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 in 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 10 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 neighbours 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 neighbours 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 neighbours, 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 neighbours, but in the worst case 10,572 nearest neighbours. The setup with $k = 12$ and $r = 80$, on the other hand, returns on average on 445 neighbours, but can return as many as 83,041.

The setup with $k = 10$ and $r = 65$ was chosen in the continuation, and the number of such hash tables L was varied. Figure 10 shows that the increase in nearest neighbours 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 from the file 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.

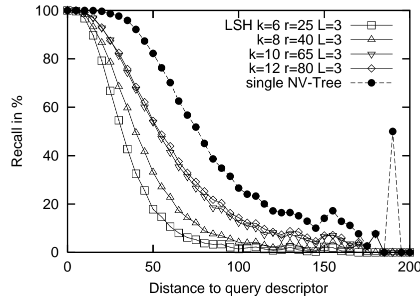

Figure 11: Recall for different LSH setups (varying word size and radius) with 3 Hashtables.

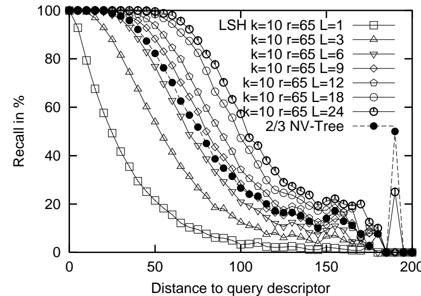

Figure 12: Recall for LSH (k=10, r=65) with varying number of tables.

6.3 Experimental Evaluation

We now turn to a comparison of the LSH and NV-tree data structures. Figure 11 shows a comparison of the recall of three LSH hash tables to a single NV-tree. As the figure shows, LSH yields with this setting significantly lower recall than that provided by the NV-tree. LSH has, on the other hand, the desirable property that it retrieves in most cases a significantly lower number of false positives. Finally, we point out that LSH is makes absolutely no distiction between low and high contrast, as it is an ϵ-approximate search. This was already known from the way LSH is designed, but we have observed this fact in our experimental evaluation.

Furthermore, Figure 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 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 128 kByte. 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 will have a much higher response time.

Figure 12 shows that the point where the LSH search outperforms the NV-tree search 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".

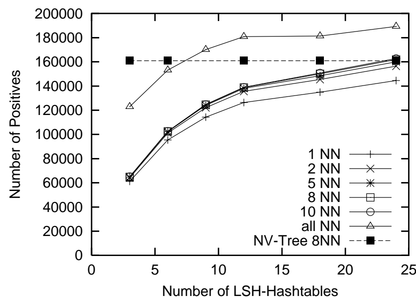

Figure 13: Positive matches for the NV-tree and LSH ($k = 10$, $r = 65$).

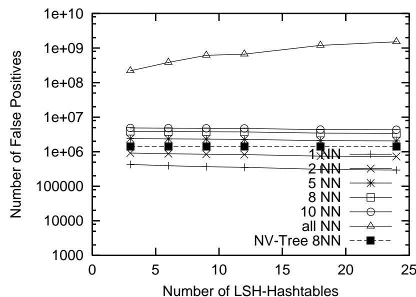

Figure 14: False positives for the NV-tree and LSH ($k = 10$, $r = 65$).

As we did for the NV-tree, it is also possible to filter false positives from the LSH search. In this case we need to aggregate the result sets of the individual LSH hash tables. As explained in Section 3.3.2 LSH is not rank-based, so rank aggregation is not applicable. Therefore we simply count the number of appearances of descriptors in the individual result sets and rank the descriptors based on this count. Then we take a fixed number of neighbours from this sorted list and declare these as the n nearest neighbours.

Figure 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 neighbour sets from 24 different LSH hash tables. Again, this is a ratio of roughly 1:8 in favor of the NV-tree.

Looking further at the false positives shown in Figure 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 neighbours, 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 neighbours.

6.4 Discussion

As we have seen in the experiments, LSH and NV-tree can give quite similar quality for nearest neighbour 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 in order to have a smaller index on disk.

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 need to filter away 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 can be very large. This behaviour may lead to unpredictably large result sets of up to 100,000 neighbours for our setup or unpredictably small result sets, which in turn leads to unpredictable I/O sizes.

7 Conclusions & Future Work

In this paper we have shown that the NV-tree as well as LSH are two very good indexing schemes for nearest neighbour search in high dimensional space. Both methods are built on the concepts of projection to lines and partioning, but they have, however, entirely 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. Regarding running time the 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. As future work we aim to further study the NV-tree, especially when reducing or even suppressing partitions in overlap and do more comparisons with LSH at even larger scales.

References

[AG01] L. Amsaleg and P. Gros. Content-based retrieval using local descriptors: Problems and issues from a database perspective. *Pattern Analysis and Applications*, 4(2/3):108–124, 2001.

[AI05] Alexandr Andoni and Piotr Indyk. E^{2}LSH 0.1 - User Manual, June 2005.

[BAG03] S.-A. Berrani, L. Amsaleg, and P. Gros. Approximate searches: k-neighbors + precision. In *Proceedings of ACM CIKM*, pages 24–31, New Orleans, LA, 2003.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The Pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of ACM SIGMOD*, pages 142–153, Seattle, WA, 1998.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[DIIM06] M. Datar, P. Indyk, N. Immorlica, and V. Mirrokni. *Locality-sensitive hashing using stable distributions.* MIT Press, 2006.

[FKM+04] Ronald Fagin, Ravi Kumar, Mohammad Mahdian, D. Sivakumar, and Erik Vee. Comparing and aggregating rankings with ties. In *Proceedings of PODS*, pages 47–58, Paris, France, 2004.

[FKS03] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of ACM SIGMOD*, pages 301–312, San Diego, CA, USA, 2003.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[GIM99] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of VLDB*, pages 518–529, Edinburgh, Scotland, 1999.

[GP87] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of ACM SIGMOD*, pages 395–398, San Francisco, CA, 1987.

[JBF07] A. Joly, O. Buisson, and C. Frélicot. Content-based copy detection using distortion-based probabilistic similarity search. *IEEE Transactions on Multimedia*, to appear, 2007.

[JFB03] Alexis Joly, Carl Frélicot, and Olivier Buisson. Robust content-based video copy identification in a large reference database. In *Proceedings of CIVR*, pages 414–424, Urbana-Champaign, IL, USA, 2003.

[Kle97] J. Kleinberg. Two algorithms for nearest-neighbour search in high dimensions. In *Proc. 27th Annual ACM Symp. on Theory of Computing*, pages 599–608, 1997.

[KSH04] Y. Ke, R. Sukthankar, and L. Huston. Efficient near-duplicate detection and sub-image retrieval. In *Proceedings of ACM Multimedia*, pages 869–876, New York, NY, USA, 2004.

[LÁJA05] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. Efficient and effective image copyright enforcement. In *Proceedings of BDA*, Saint Malo, France, 2005.

[LÁJA06] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg. Scalability of local image descriptors: A comparative study. In *Proc. ACM Multimedia Conference*, Santa Barbara, CA, USA, 2006.

[LCGMW02] C. Li, E.Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clindex: Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4), 2002.

[Liu06] Ting Liu. *Fast Nonparametric Machine Learning Algorithms for High-dimensional Massive Data and Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2006.

[LMGY04] T. Liu, A. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of Neural Information Processing Systems (NIPS 2004)*, Vancouver, BC, Canada, 2004.

[Low99] D. G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, pages 1150–1157, Corfu, Greece, 1999.

[Low04] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[Pea01] Karl Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11):559–572, 1901.

[Pet01] F. A. P. Petitcolas et al. A public automated web-based evaluation service for watermarking schemes: StirMark benchmark. In *Proc. of Electronic Imaging, Security and Watermarking of Multimedia Contents III*, San Jose, CA, USA, 2001.

[Uhl91] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[WSB98] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB*, New York City, USA, 1998.

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is