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

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

Abstract—Over the last two decades, much research effort has been spent on nearest neighbor search in high-dimensional data sets. Most of the approaches published thus far 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 neighbor 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.

Index Terms—High-dimensional indexing, multimedia indexing, very large databases, approximate searches.

1 Introduction

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

Most of the approaches published thus far, however, have only been tested on rather small collections ranging from tens of thousands of descriptors to a few million (e.g., see [25], [16], [3], [7]) and some have been explicitly shown not to work well at high-dimensions or large scale [1], [14]. 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 [24], for a collection of 1.5 billion global descriptors [17], [21]. In that study, 2,000 workstations were used, presumably having at least a terabyte or two of total main memory. In general, however, Gray has shown [10], [8] that using very large main memory is not economical; that data, which is accessed less frequently than every 5 minutes, should not be kept in main memory.

Locality Sensitive Hashing (LSH) by Indyk et al. [9], [6] has also been considered for large-scale retrieval. LSH is based on the concept of projecting descriptors onto a random line and classifying 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 of these fingerprints in a hash table, it is possible to retrieve similar descriptors with a single disk read.

Ke et al. [13] 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, as many queries batched together can benefit from a single sequential scan, it yields very poor response times. Joly et al. [11] studied an application with 1.5 billion 20D local descriptors. They also completed processing with a sequential scan.

1.2 Contribution of This Paper

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

- First, we propose the Nearest-Vector-tree (NV-tree), a disk-based data structure that gives good approximate answers with a *single random disk read*, even for

- *H. Lejsek and F.H. Ásmundsson are with Eff2 Technologies ehf., Kringlan 1, IS-103 Reykjavík, Iceland. E-mail: {herwig, fridrik}@eff2.net.*
- *B.Þ. Jónsson is with Reykjavík University, Kringlan 1, IS-103 Reykjavík, Iceland. E-mail: bjorn@ru.is.*
- *L. Amsaleg is with CNRS-IRISA, IRISA, Campus de Beaulieu, 35042 Rennes, France. E-mail: Laurent.Amsaleg@irisa.fr.*

Manuscript received 18 July 2007; revised 31 Jan. 2008; accepted 29 Apr. 2008; published online 20 May 2008.
Recommended for acceptance by J. Luo.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number TPAMI-2007-07-0435.
Digital Object Identifier no. 10.1109/TPAMI.2008.130.

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 Scale Invariant Feature Transform (SIFT) descriptors [20]. We show that the SIFT descriptors are very distinctive and have high contrast, even in a collection of 180 million data points. 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 paper is organized as follows: First, we present the NV-tree in Section 2 and its implementation details in Section 3. Then, we present the collection and workload used in our experiments in Section 4. In Section 5, we analyze the ground-truth result quality of our workload and, in Section 6, we describe the performance of the NV-tree. In Section 7, we compare the performance of the NV-tree to that of LSH. We conclude in Section 8.

2 The NV-Tree

The NV-tree is a disk-based data structure designed for efficient approximate k-nearest neighbor search in very large high-dimensional collections. In essence, it transforms costly nearest neighbor searches in the high-dimensional space into efficient unidimensional 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 are highly likely to contain all the close neighbors in the collection.[1]

The curse of dimensionality phenomenon 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, but, given the low cost of disk space, this is a good trade-off.

In this section, we first outline the algorithms for NV-tree creation (Section 2.1) and search (Section 2.2). Then, we consider strategies for projections (Section 2.3) and partitioning (Section 2.4). Finally, we briefly describe insertion to, and deletion from, the NV-tree (Section 2.5), before highlighting key properties of the NV-tree (Section 2.6). The implementation of the NV-tree is described in Section 3.

1. Note that dealing with strongly clustered data sets 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.

2.1 NV-Tree Creation

Overall, an NV-tree is a tree index consisting of: 1) 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 2) 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 2.3.

The projected values are then partitioned into disjunct subpartitions based on their position on the projection line. For each pair of adjacent partitions, an overlapping subpartition, covering 50 percent of both partitions, is created for redundant coverage of the partition borders. Information about all these subpartitions, 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 2.4.

To build subsequent levels of the NV-tree, this process of projecting and partitioning is repeated for all of the new subpartitions using a new projection line at each level, creating the hierarchy of inner nodes. The process stops when the number of descriptors in a subpartition 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 subpartition, and the ordered identifiers are written to a leaf node on disk.

2.2 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 subpartition with center point closest to the projection of the query descriptor. This process of projection and choosing the right subpartition 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, and so forth. 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, 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 Projection Strategies

Projecting high-dimensional data points to random lines was introduced by Kleinberg [12] and subsequently used in several other high-dimensional indexing techniques [7], [6], [18]. 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 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 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 precomputed 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.

2.4 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 subpartition gets the same number of descriptors, and all leaf partitions are of the same size. Although node fan-out 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 large 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, subpartitions are created such that the absolute distance between their boundaries is equal. All of the data points in each interval belong to the associated subpartition. With this strategy, however, the normal distribution of the projections leads to a significant variation in the cardinalities of subpartitions. 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 σ and mean m of the projections along the projection line. Then, a parameter α is used to determine the partition borders as $\dots, m - 2\alpha\sigma, m - \alpha\sigma, m, m + \alpha\sigma, m + 2\alpha\sigma, \dots$. Small adjacent subpartitions are merged until the resulting cardinality hits the leaf node size limit and then written to disk. Subpartitions containing many data points, on the other hand, are subsequently repartitioned. 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 subpartitions 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 subpartition that 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 subpartition 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 subpartitions 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.

2.5 Insertions and Deletions

In many application settings, dynamic maintenance of the data collection is required, for example when reindexing the collection leads to intolerable interruption of service. Due to the redundancy arising from overlapping partitions, however, each descriptor must be dynamically inserted into (or deleted from) many leaf nodes. Unlike the search, insertion (or deletion) must, at each level, descend into the two subpartitions that contain the projection of the descriptor. In the worst case, a descriptor must thus be inserted into (or deleted from) 2^h leaf nodes, where h is the number of levels in the tree. Although, in the case of the *Unbalanced* tree, the number of affected leaf nodes will be lower in practice, it is still high enough that a careful implementation is required. In Section 3.4, an efficient implementation of insertions and deletions is described.

2.6 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 cannot be calculated, other means of removing 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 6.3.

3 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 leafs of the NV-tree.

The NV-tree is written in C++. The code has been embedded in a server, which listens for requests for searches or insertions. Upon invocation, the 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.

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. Finally, we briefly describe the implementation of insertions and deletions.

3.1 NV-Tree Creation

As described in Section 2.1, 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 2.3. The collection is then sampled yet again, using a larger sample, to determine approximate values for the m and σ parameters, described in Section 2.4 (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) subpartitions and written to temporary files for those subpartitions. This whole process is then repeated for each of the temporary files in a depth-first manner. When a leaf partition is formed, the (*projected value*, *descriptor identifier*) pairs of the leaf partition are sorted in memory by their projected values and written to the leaf node.

3.2 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 subpartitions. 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 subpartitions.
- **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 halfway between the upper and lower partition borders of adjacent child nodes.
- **Projection Line:** As described in Section 2.3, 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.

3.3 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 (*projected value*, *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 lookup 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 16th projected value; this setting is used throughout this paper.

3.4 Insertions and Deletions

Recall that, due to the redundancy arising from overlapping partitions, each new descriptor must be dynamically inserted into many leaf nodes. In order to avoid immediately reading and writing each of those leaf nodes, a memory-based holding area is employed. The holding area contains, for each leaf node, a list of all (*projected value*, *descriptor identifier*) pairs that have been inserted to the node but not written to disk. The search has been modified to scan this holding area also, when reading a leaf node. When the holding area fills, leaf nodes are opportunistically updated and written to disk as they are read by the search process. This way, disk activity due to insertions is minimized.

When a leaf node is full, it must be split. As part of the splitting process, new random lines must be chosen from the line pool for the resulting new leaf nodes. As the actual descriptors are not stored in the leaf node, the projection along this new random line requires costly random accesses to the descriptor collection. It is therefore more profitable to delay and buffer splits. Once the split buffer fills up, the whole collection is sequentially scanned, and the buffered leaf nodes are split in a manner similar to the index creation process.

A key consideration is how many new partitions a leaf node should be split into. With too many new partitions, each will have low utilization and the index will grow fast. Choosing a small number of new partitions will lead to small intermediate nodes, however, which in turn leads to deep NV-trees and fast index growth. To avoid both situations, we have chosen a policy where, when the parent of a full leaf node has low fan-out, the parent node is split by taking the contents of all of the leaf node's siblings into account in the splitting process. This results in a parent node with higher fan-out, without hurting disk utilization.

The deletion of descriptors from the NV-tree is implemented similarly to insertions, by opportunistically propagating the deletions to the appropriate leaf nodes. Additionally, however, a list of deleted descriptors is maintained and used to filter search results, such that the deletion is immediately apparent in the search results. A reference count is maintained to remove the deleted descriptor from the list once all its redundant occurrences have been removed from the disk.

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

4.1 Descriptors and Queries

In this study, we use the well-known SIFT method [19], [20], 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 a 128D vector, which is normalized to a length of 512.

The descriptor collection was obtained by extracting local features from an archive of about 150,000 images obtained from Morgunblaðið, the major newspaper in Iceland (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.

SIFT descriptors are particularly suited for near-duplicate image detection (e.g., see [20]) and have also been shown effective for detecting copyright violations of images [15]. We have therefore created query descriptors by modifying copies of images from the collection, following the approach in [15], using the Stirmark benchmarking tool [22]. 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 [20], [15], 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 that the SIFT descriptors have been shown not to handle well [15].

4.2 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 are 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 resulting 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.

Recent results by Beyer et al. [5] and Shaft and Ramakrishnan [23], 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 data set 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 in [23] 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 data set that fulfill a given contrast criterion.

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, computing SIFT over an image collection produces a contrasted set of descriptors [20]. 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 [20]. 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 $d(n_{100}, q)/d(n_i, q) > c$.[2]

Using this contrast criterion, it is possible to build a ground-truth set for an application. In Section 5, we analyze the quality of these three approaches to generate the ground-truth sets for our application from the results of a sequential scan.

2. In fact, we can generalize Lowe's criterion, by saying that returned neighbor n_i is meaningful with respect to contrast c 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 paper.

5 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 4.1. 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 percent. 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 [15]. 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.

5.1 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 $k = 1$, 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.

5.2 Ground Truth Based on ϵ-Distance

We now analyze how the absolute distance between the query descriptor and returned neighbors affects the result quality. Fig. 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.

Fig. 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

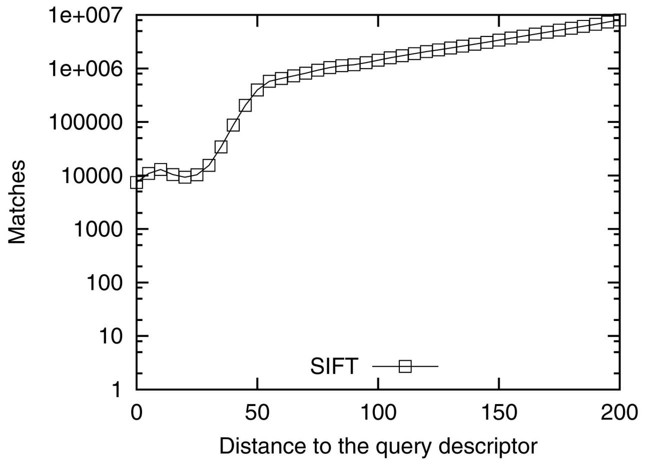

Fig. 1. The distribution of all neighbors based on distance to the query descriptor.

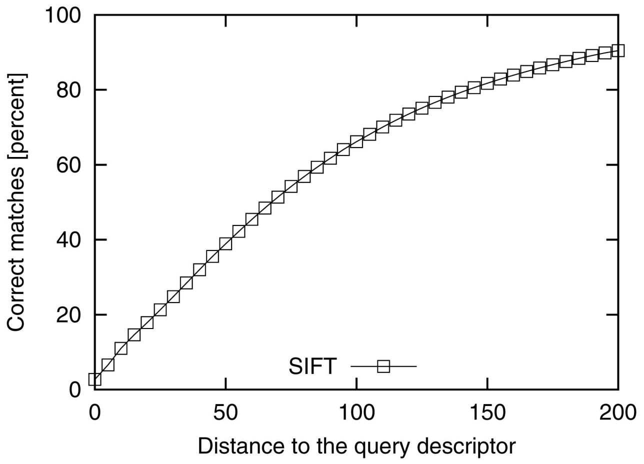

Fig. 2. The cumulative distribution of correct matches based on distance to the query descriptor.

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.

More importantly, however, Fig. 2 shows that fewer than 20 percent 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 that includes a large number of correct matches and only a small number of false matches.

5.3 Ground Truth Based on Contrast

Finally, we consider the effect of contrast on the quality of the ground-truth set. Fig. 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 4.1 as $d(n_{100}, q)/d(n_i, q) > c$. The figure shows that 36 percent of the correct matches are more than five times closer in distance than the 100th 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 percent, pass the contrast threshold. About 20 percent of the correct matches have a contrast threshold lower than 1.5 and are therefore rather hard to detect from the false matches.[3]

Fig. 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 Fig. 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 Figs. 3 and 4, we see that choosing a higher contrast threshold results in lower recall but fewer false matches, and vice versa. Comparing these to Figs. 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.[4] 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 percent of the descriptors in the contrast-based ground-truth set are correct matches and about 24.9 percent are false matches.

5.4 Discussion

The discussion 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 4.2, it is the only approach with solid theoretical underpinnings. As a result, we use contrast-based ground-truth sets in the remainder of this paper. 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 very small and meaningful result sets for nearly all query descriptors.

6 NV-Tree Performance

In this section, we start by describing the NV-tree configurations used in the experiments. Then, we present two experiments that analyze key properties of the NV-tree. In Section 6.2, 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 6.3, we then discuss an experiment with up to three NV-trees, which demonstrates that with such configurations false

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

4. The fact that this number is similar to the number of correct matches in the sequential scan results is purely a coincidence.

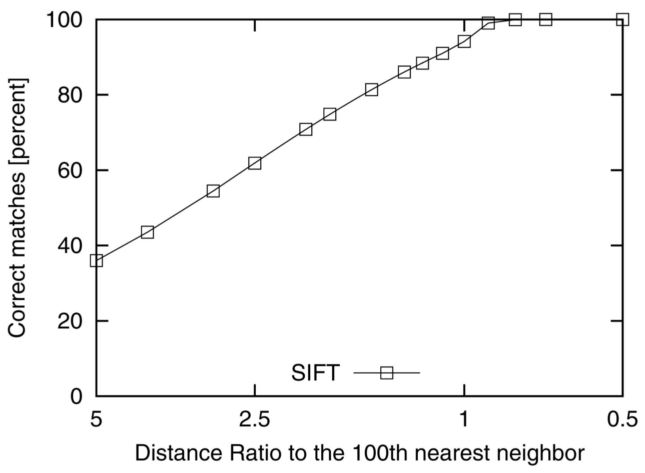

Fig. 3. The cumulative distribution of correct matches based on the contrast threshold.

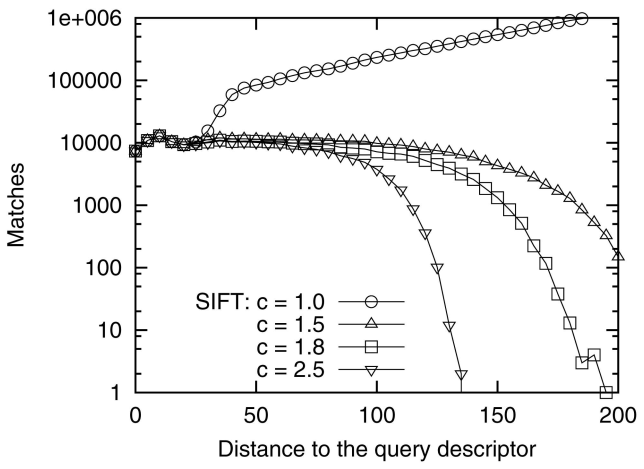

Fig. 4. Distribution of neighbors passing the contrast criterion by distance to query descriptor, for various contrast thresholds.

positives can be largely eliminated, while keeping most of the high recall. Finally, we present experimental results for insertions in Section 6.4.

6.1 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 Kbytes). As leaf nodes are sparse (keeping a projected value for every 16th descriptor identifier), a maximum of 5,579 descriptors identifiers can be stored per leaf. Leaf nodes are typically filled to 67 percent 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 60 Mbytes of main memory.
- We used the *Approximate PCA* strategy to select the random lines. We generated an initial line pool of 1,000 lines,[5] where each pair of lines has a minimum angle of 72 degrees. Starting with a very small sample from the partition (typically 0.01 percent), 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 percent over random lines.
- We used the *Hybrid* partitioning strategy with α set to 0.55. Before partitioning, a sample of about 5 percent of the points in a partition are used to determine m and σ (see Section 2.4). The *Hybrid* strategy generally yields about 5 percent higher recall than a *Balanced* strategy but equivalent to the *Unbalanced*, while only requiring about 20 percent 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 Gbytes 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 percent to 3 percent 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.

Nevertheless, one NV-tree needs about 12.5 ms 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 three NV-trees are used, the response time is about 38.5 ms.

6.2 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.

6.2.1 Recall

Fig. 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 that 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 Fig. 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

5. 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.

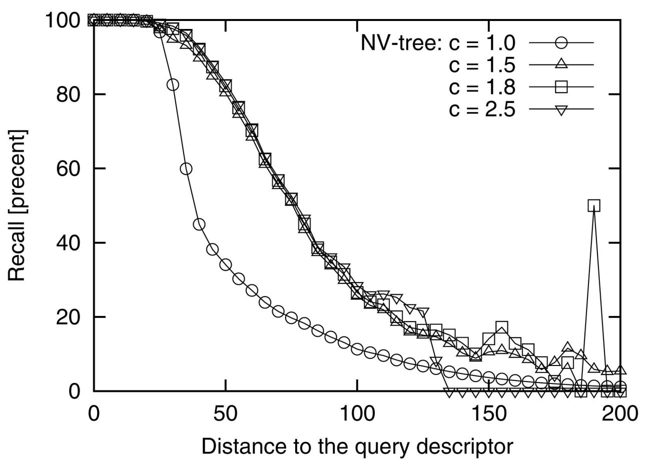

Fig. 5. Recall for a single NV-tree retrieving 1,000 nearest neighbors per query.

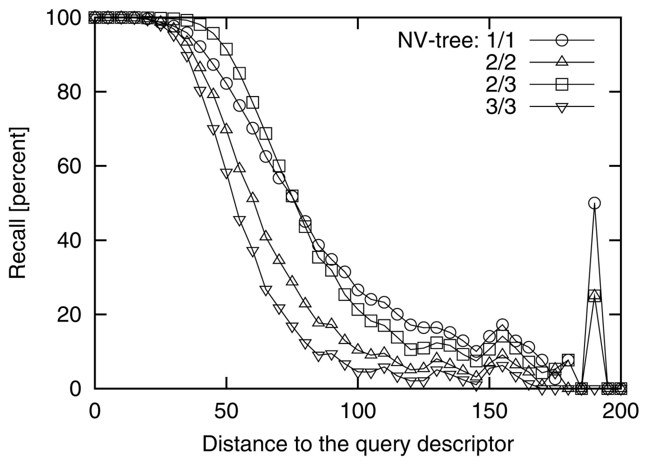

Fig. 6. Recall by aggregating the result lists of two or three independent NV-trees ($c = 1.8$).

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 of 25-100. Turning back to Fig. 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 percent 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.

6.2.2 *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.

6.3 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 the case of two indices first. A technique called median rank aggregation [7] 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 retrieval performance and recall, and then focus on false positives.

6.3.1 *Performance*

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.

6.3.2 *Recall*

Fig. 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 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,

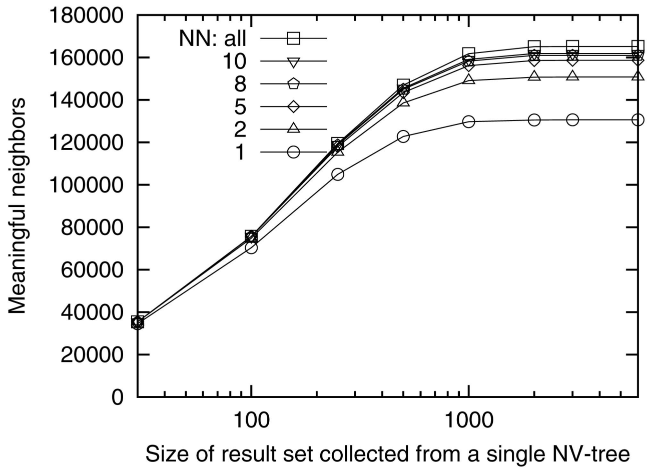

Fig. 7. Meaningful neighbors of 2/3 NV-trees based on number of neighbors retrieved.

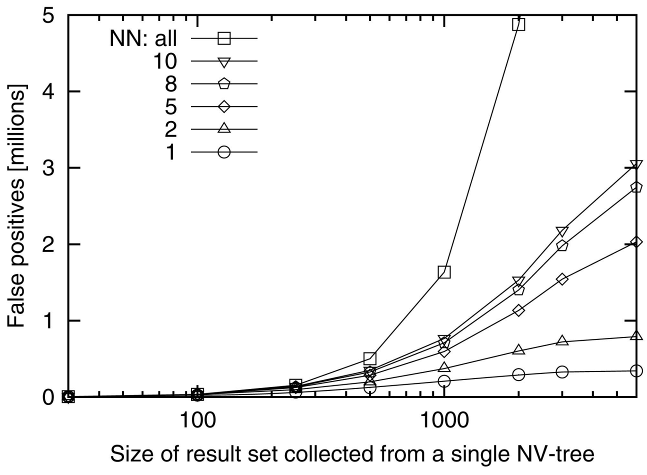

Fig. 8. False positives of 2/3 NV-trees based on number of neighbors retrieved.

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.

6.3.3 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 given as follows: Each of the three NV-trees is probed to yield a (ranked) result set of a specific size. Then, these result 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 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 false positives.

Fig. 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 the overall shape of the lines first, 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 Fig. 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 result sets achieve only minor improvements, but as we will see in a moment, they increase the number of false positives significantly.

Fig. 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 percent of all queries return more than 10 nearest neighbors and 2.5 percent 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 Figs. 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 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 1,000 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.

6.4 Experiment 3: Insertions

To measure the performance of insertions, as well as the quality of the resulting index, we first indexed a 36 million descriptor subset of our collection (about 20 percent of the collection) and created a single initial NV-tree index. Then, we created an insertion stream of the remaining 144 million descriptors and measured the insertion performance at regular intervals. Finally, we measured the retrieval quality of the resulting index and compared it to the results of Section 6.2 for an index built from scratch. In the following, we briefly describe the results of this experiment.

6.4.1 Recall

The recall of the resulting NV-tree (not shown) is 65.6 percent. This can be compared with the recall of an

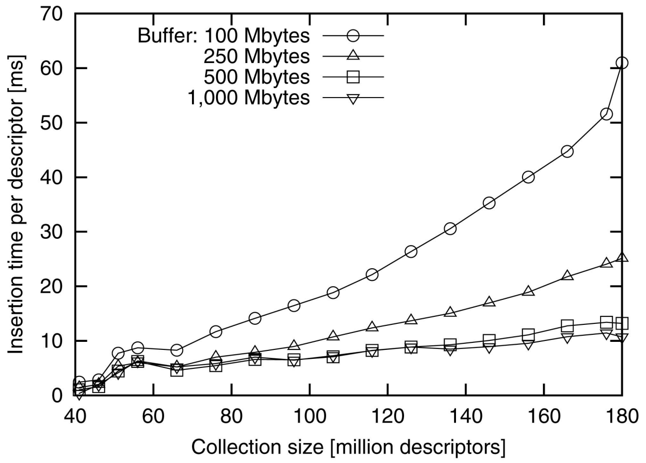

Fig. 9. Average time to insert a descriptor as the index grows, for various buffer sizes.

NV-tree built from scratch, which is 65.8 percent (see Section 6.2). The recall is thus essentially the same regardless of construction method. Given that the final index after insertions contains five times as many descriptors as the initial index, this is a strong result.

6.4.2 *Efficiency*

Turning to the efficiency of insertions, the key factor deciding the performance is the buffer size of the holding area for descriptors, with the split buffer having a smaller impact. Recall that the holding area is used to postpone actual insertions into leaves such that disk operations can be performed opportunistically and the cost of disk operations amortized over a large number of insertions. We have experimented with buffer sizes ranging from 100 to 1,000 Mbytes (about 5 percent to 50 percent of the server's memory), of which 80 percent have been allocated to the holding area and 20 percent to the split buffer.

Fig. 9 shows the average time to insert a single descriptor. The x-axis shows the size of the collection that is indexed at each time, while the y-axis shows the insertion time in milliseconds. Overall, the figure shows that with limited buffer sizes, the insertion time grows significantly as the index becomes larger. With a buffer size of 500 Mbytes, however, the insertion time only goes up to about 12 ms, which is about the cost of a random I/O. Beyond a buffer size of 500 Mbytes, few further benefits are seen.

We also analyzed the breakdown of the insertion costs (not shown). With a 500 Mbyte holding area buffer, the cost of updating leaf nodes on disk was about 40 percent of the cost, and the cost of splitting leaf nodes was about 60 percent of the cost.

Overall, the time to insert all 144 million descriptors with a 500 Mbyte buffer was 13.4 days. This time can be compared to the cost of building the index, which was less than 16 hours. While the cost of bulk loading is thus much lower, the cost of insertions is still quite reasonable. Even when the index contained nearly 180 million descriptors, we observed an insertion throughput of about 5,000 descriptors per minute.

6.5 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 indices.

In general, the NV-tree returns more than 99 percent 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.

7 Comparison to Related Work

The two major data structures most related to the NV-tree are the Spill-tree [18], [17], [21] and LSH [9], [6]. In this section, we first briefly compare the NV-tree to the Spill-tree before focusing on LSH in the remainder of the section.

7.1 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 repartitioning 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 percent. 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.

7.2 Locality Sensitive Hashing

In the remainder of this section, we focus on LSH, 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 LSH. 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 LSH 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 in [9] and then later extended to l_p norm in [6]. Most of its applications, however, have used rather small collections, which could easily fit in memory.[6]

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.

6. A disk-based strategy was developed by Ke et al. [13]. 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.

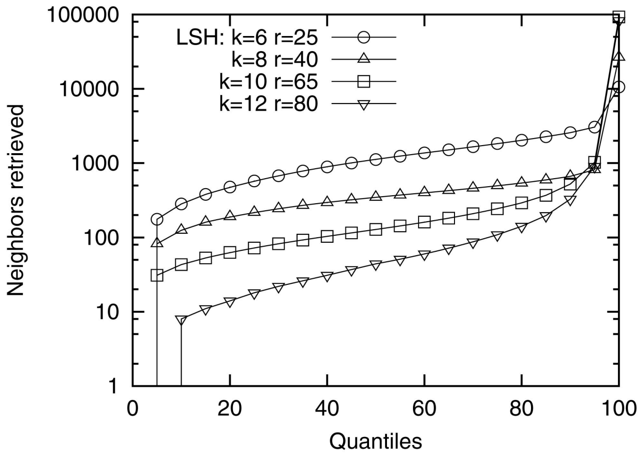

Fig. 10. Distribution of result set size for LSH with three hash tables ($L = 3$).

7.3 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 Gbytes, 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 [2] 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 hash tables 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).

Fig. 10 shows 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

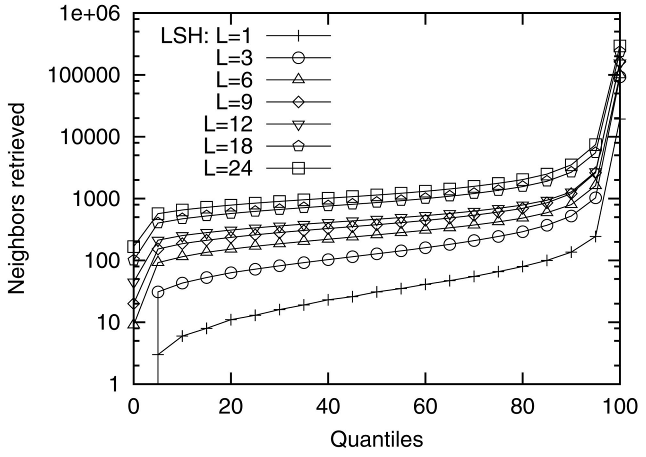

Fig. 11. Distribution of result set size for LSH with varying number of hash tables $(k = 10, r = 65)$.

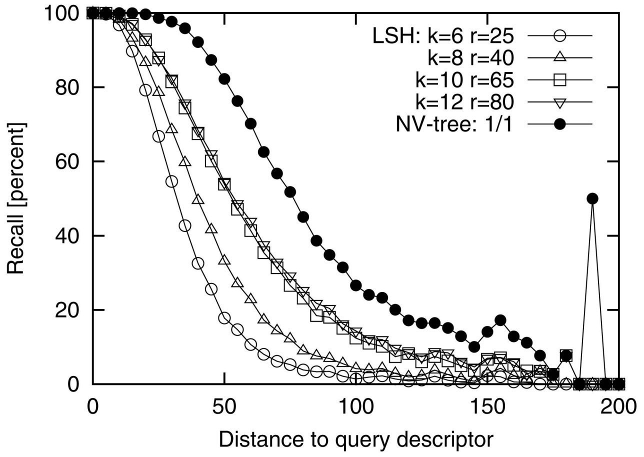

Fig. 12. Recall for different LSH setups (varying word size and radius) with three hash tables $(L = 3)$.

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 percent to 10 percent 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, 445 neighbors 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 hash tables L was varied. Fig. 11 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 afterward, 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 nonoverlapping nature of LSH, however, each hash table requires only about 2.1 Gbytes of disk space, which is significantly lower than the storage needed for a single NV-tree.

7.4 Recall of LSH

We now turn to a comparison of the LSH and NV-tree data structures. Fig. 12 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 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 search. This was already known from the design of LSH, but we have observed this fact in our evaluation.

Furthermore, Fig. 12 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 $r = 65$.

Fig. 13 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 Kbytes. 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.

Fig. 13 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."

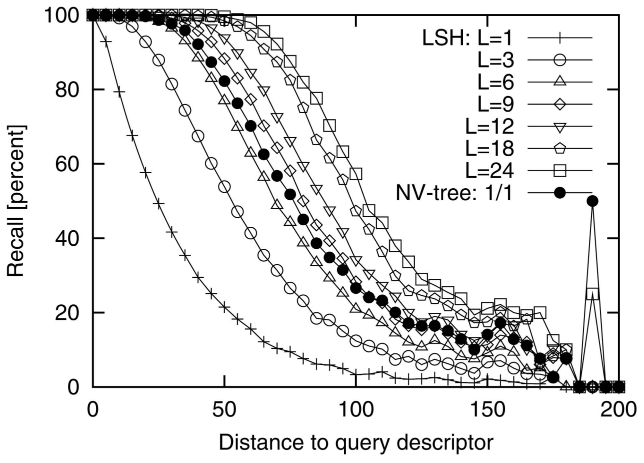

Fig. 13. Recall for LSH with varying number of tables $(k = 10, r = 65)$.

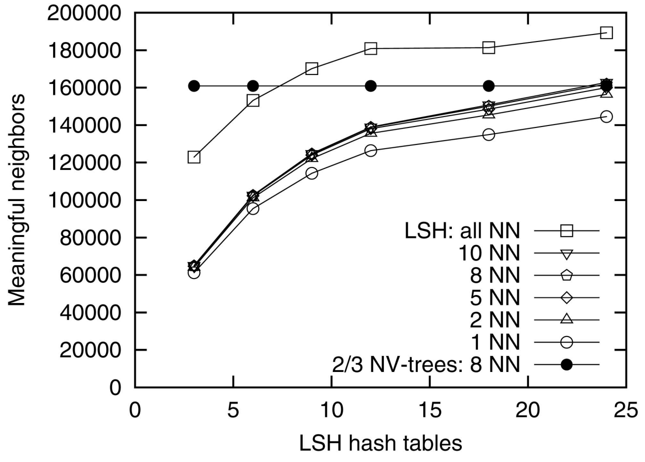

Fig. 14. Meaningful neighbors for the NV-tree and LSH $(k = 10, r = 65)$.

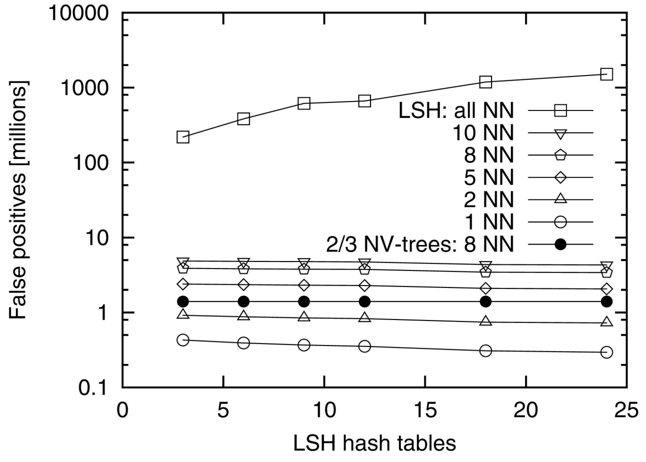

Fig. 15. False positives for the NV-tree and LSH $(k = 10, r = 65)$.

7.5 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 7.3, 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 that have no internal ranking. Instead, we have taken the approach used in [4] 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 the aggregated nearest neighbors.

Fig. 14 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 further 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 Fig. 15, 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 result sets for all queries. The generation of a small and meaningful answer set is then just a matter of ranking the neighbors.

7.6 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.

When false positives are tolerated, the NV-tree is about eight times faster but uses 50 Gbytes of disk space versus 8×2.1 Gbytes $= 16.8$ Gbytes for LSH, or three 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 Gbytes of disk space while LSH needs about 24 disk reads from 50.4 Gbytes 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 hash table 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.

8 Conclusions

In this paper, we have proposed 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. Furthermore, searching the NV-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, even in a collection of 180 million data points. 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 and LSH are two very 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 that 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 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.

Directions for future work include a theoretical study of the approximation properties of the NV-tree, an analysis of the impact of redundancy on result quality, further comparisons to competing data structures, and a performance study at even larger scales.

ACKNOWLEDGMENTS

This work was partially supported by Icelandic Research Fund Grant 060036021, by an INRIA Eff^2 Associate Teams grant, and by an ÉGIDE Jules Verne travel grant. The authors would like to thank Morgunblaðið for the use of their large picture collection, the authors of LSH and SIFT for giving them access to their implementations, and the anonymous reviewers for their excellent comments that significantly improved the presentation of this paper.

REFERENCES

[1] L. Amsaleg and P. Gros, "Content-Based Retrieval Using Local Descriptors: Problems and Issues from a Database Perspective," *Pattern Analysis and Applications,* vol. 4, nos. 2/3, pp. 108-124, 2001.

[2] A. Andoni and P. Indyk, E^{2}LSH *0.1—User Manual,* June 2005.

[3] S.-A. Berrani, L. Amsaleg, and P. Gros, "Approximate Searches: *k*-Neighbors + Precision," *Proc. 12th ACM Int'l Conf. Information and Knowledge Management,* pp. 24-31, 2003.

[4] S. Baluja and M. Covell, "Content Fingerprinting Using Wavelets," *Proc. IET Conf. Multimedia,* 2006.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is 'Nearest Neighbor' Meaningful?" *Lecture Notes in Computer Science,* vol. 1540, pp. 217-235, 1999.

[6] M. Datar, P. Indyk, N. Immorlica, and V. Mirrokni, *Locality-Sensitive Hashing Using Stable Distributions.* MIT Press, 2006.

[7] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient Similarity Search and Classification via Rank Aggregation," *Proc. ACM SIGMOD '03,* pp. 301-312, 2003.

[8] J. Gray and G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb," *SIGMOD Record,* vol. 26, no. 4, pp. 63-68, 1997.

[9] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proc. 25th Int'l Conf. Very Large Data Bases,* pp. 518-529, 1999.

[10] J. Gray and F. Putzolu, "The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time," *Proc. ACM SIGMOD '87,* pp. 395-398, 1987.

[11] A. Joly, O. Buisson, and C. Frélicot, "Content-Based Copy Detection Using Distortion-Based Probabilistic Similarity Search," *IEEE Trans. Multimedia,* vol. 9, no. 2, pp. 293-306, Feb. 2007.

[12] J. Kleinberg, "Two Algorithms for Nearest-Neighbour Search in High Dimensions," *Proc. 29th Ann. ACM Symp. Theory of Computing,* pp. 599-608, 1997.

[13] Y. Ke, R. Sukthankar, and L. Huston, "Efficient Near-Duplicate Detection and Sub-Image Retrieval," *Proc. ACM Multimedia Conf.,* pp. 869-876, 2004.

[14] H. Lejsek, F.H. Ásmundsson, B.Þ. Jónsson, and L. Amsaleg, "Efficient and Effective Image Copyright Enforcement," *Proc. Journées Bases de Données Avancées,* 2005.

[15] H. Lejsek, F.H. Ásmundsson, B.Þ. Jónsson, and L. Amsaleg, "Scalability of Local Image Descriptors: A Comparative Study," *Proc. ACM Multimedia Conf.,* pp. 589-598, 2006.

[16] C. Li, E.Y. Chang, H. Garcia-Molina, and G. Wiederhold, "Clindex: Clustering for Approximate Similarity Search in High-Dimensional Spaces," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 4, pp. 792-808, July/Aug. 2002.

[17] T. Liu, "Fast Nonparametric Machine Learning Algorithms for High-Dimensional Massive Data and Applications," PhD thesis, School of Computer Science, Carnegie Mellon Univ., 2006.

[18] T. Liu, A. Moore, A. Gray, and K. Yang, "An Investigation of Practical Approximate Nearest Neighbor Algorithms," *Proc. Neural Information Processing Systems,* pp. 825-832, 2004.

[19] D.G. Lowe, "Object Recognition from Local Scale-Invariant Features," *Proc. Int'l Conf. Computer Vision,* pp. 1150-1157, 1999.

[20] D.G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Int'l J. Computer Vision,* vol. 60, no. 2, pp. 91-110, 2004.

[21] T. Liu, C. Rosenberg, and H.A. Rowley, "Clustering Billions of Images with Large Scale Nearest Neighbor Search," *Proc. IEEE Workshop Applications of Computer Vision,* pp. 28-33, 2007.

[22] F.A.P. Petitcolas et al., "A Public Automated Web-Based Evaluation Service for Watermarking Schemes: StirMark Benchmark," *Proc. Electronic Imaging, Security and Watermarking of Multimedia Contents III,* pp. 575-584, 2001.

[23] U. Shaft and R. Ramakrishnan, "Theory of Nearest Neighbors Indexability," *ACM Trans. Database Systems,* vol. 31, no. 3, pp. 814-838, 2006.

[24] J.K. Uhlmann, "Satisfying General Proximity/Similarity Queries with Metric Trees," *Information Processing Letters,* vol. 40, no. 4, pp. 175-179, 1991.

[25] R. Weber, H. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th Int'l Conf. Very Large Data Bases,* pp. 194-205, 1998.

Herwig Lejsek received the MSc degree in computer science from Reykjavík University, Iceland, in 2005 and the Dipl-Ing degree in computer engineering from Vienna University of Technology, Austria, in 2006. Subsequently, he enrolled as a PhD student at Reykjavík University, where he is working under the supervision of Björn Þór Jónsson. Since January 2008, he has been the CEO of Eff2 Technologies ehf, a start-up company emerging from Reykjavík University's research laboratories. His research work focuses primarily on high-dimensional data structures and content-based multimedia retrieval.

Friðrik Heiðar Ásmundsson received the MSc degree in computer science from Reykjavík University, Reykjavík, Iceland, in 2006. After working at deCODE Genetics for 18 months, he cofounded Eff2 Technologies ehf, where he currently works on large-scale video copyright protection systems. His research focuses on content-based multimedia retrieval, distributed high-dimensional data structures, and parallel programming.

Björn Þór Jónsson received the PhD degree in 1999 from the University of Maryland, College Park, where his research focused on semantic caching and buffer management. After working in industry for 18 months, in August 2000, he joined Reykjavík University, Reykjavík, Iceland, where he is currently an associate professor in the School of Computer Science. His research work focuses primarily on the performance of content-based multimedia retrieval as well as the performance and tuning of relational database systems. He was a cocreator of the Computer Vision Meets Databases (CVDB) workshop series.

Laurent Amsaleg received the PhD degree from the University of Paris 6, Paris, in 1995. He worked on relational and object-oriented databases, garbage collection, microkernels, and single-level stores. He then spent 18 months in the Database Group at the University of Maryland, College Park, designing flexible database query execution strategies (Query Scrambling). Subsequently, he received a full-time research position at the Centre National de la Recherche Scientifique (CNRS), France and joined IRISA, Rennes, France. His research primarily focuses on the performance of content-based multimedia retrieval, which include the design of multidimensional indexing techniques. He was a cocreator of the Computer Vision Meets Databases (CVDB) workshop series.