NV-Tree: Nearest Neighbors at the Billion Scale

Herwig Lejsek
Videntifier Technologies
Reykjavík, Iceland
herwig@videntifier.com

Björn Þór Jónsson
School of Computer Science
Reykjavik University, Iceland
bjorn@ru.is

Laurent Amsaleg
IRISA–CNRS
Rennes, France
laurent.amsaleg@irisa.fr

ABSTRACT

This paper presents the NV-Tree (Nearest Vector Tree). It addresses the specific, yet important, problem of efficiently and effectively finding the approximate k-nearest neighbors within a collection of a few billion high-dimensional data points. The NV-Tree is a very compact index, as only six bytes are kept in the index for each high-dimensional descriptor. It thus scales extremely well when indexing large collections of high-dimensional descriptors. The NV-Tree efficiently produces results of good quality, even at such a large scale that the indices cannot be kept entirely in main memory any more. We demonstrate this with extensive experiments using a collection of 2.5 billion SIFT (Scale Invariant Feature Transform) descriptors.

1. INTRODUCTION

Finding the k-nearest neighbors (k-nn) of a single query point in an extremely large collection of high-dimensional vectors is a core problem at the root of many applications, such as content-based image retrieval tasks or fine-grained object recognition. Over the years, it has been observed and demonstrated [15] that only approximate indexing and search strategies are viable when the data collections are very large and/or sufficiently high-dimensional, as approximate high-dimensional approaches are able to trade accuracy against dramatic response time improvements. Following that trend, this paper presents an enhanced version of the NV-Tree [7], which can return the approximate, yet quite accurate, k-nn of individual query points found in a collection of few billions of data items stored on disks.

1.1 Large Scale High-Dimensional Data Sets

Recently, various approaches for k-nn retrieval at large scale were proposed, some even termed as addressing "web-scale problems" [1, 17, 3]. These approaches, however, are considering at most several million high-dimensional descriptors. This is in part because they use standard global description schemes (e.g., MPEG-7, Gist) creating indexed sets made of millions of items. While this is already quite impressive, the problem that is tackled in this paper is finding indexing solutions when the data collections hit the billion scale. In this case, there is no other choice than using more advanced description schemes producing much larger data sets. Note that at that scale it is mandatory to store such data collections on disks, and the indexing algorithms must be able to deal with secondary storage accesses.

Sophisticated description schemes involving local description techniques can generate billions of descriptors from millions of images. This is for example the case with the well-known SIFT description technique [10], which typically creates about one thousand 128-dimensional descriptors for a 384×512 pixel image. With SIFT, collections containing a few million images typically generate high-dimensional data sets containing a few billion points—such collections are typical of what real world photo agencies deal with. In this paper, for example, we use a collection of 2.5 billion SIFT descriptors from 2.5 million images randomly downloaded from Flickr. Using local description techniques thus results in huge collections of high dimensional vectors.

Knowing whether an approximate indexing technique returns high-quality neighbors requires the computation of a meaningful ground truth. This requires (i) a definition of a set of meaningful query descriptors, and (ii) the computation of the exact k-nearest neighbors of each individual query descriptor, for example using a costly and exhaustive sequential scan of the whole collection. From that ground truth, it is possible to evaluate how good a specific approximate technique is, by observing the recall at the level of each individual query descriptor.

It is imperative to design approximate techniques with good recall; the result must contain most of what would be returned in an exact answer. Achieving good recall is often possible, at a cost of lower precision, by accepting false positive as part of the result. For some applications, false positives are acceptable; for other applications, they can simply be removed after-the-fact. In [7], e.g., the method of median rank aggregation [4] from additional indices has shown to work well as a false-positive filter.

1.2 Large Scale Approximate Indexing

The literature contains several very interesting approximate high-dimensional indexing schemes. One very popular approximate method is Locality Sensitive Hashing (LSH) [2]. It is based on random projections to segmented lines acting as hash functions. Multiple hash tables enforce some redundant coverage of the feature space. LSH, however, is by its nature solving a slightly different problem, as it essentially

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICMR '11, April 17-20, Trento, Italy
Copyright ©2011 ACM 978-1-4503-0336-1/11/04 ...$10.00.

answers a range search, gathering points within a predefined ϵ-distance from the query point. In order to adapt LSH for nearest neighbor retrieval it has to be configured with a significantly large ϵ threshold, which occasionally leads to a very large number of false positives.

Another approach is the Spill-Tree, proposed in 2004 by Liu et al. [9], which is a tree-structure based on splitting dimensions in a round-robin manner, and introducing (sometimes very significant) overlap in the split dimension to improve retrieval quality. Finally, in 2009, Lejsek et al. proposed the NV-Tree, which is a tree structure based on repeated applications of projections to random lines and partitioning of the projected data. The NV-Tree also introduced overlap between partitions to improve retrieval quality, but in a much more controlled manner [7].

Overall, all these method are using some form of space partitioning. They are approximate because they search in only one or few partitions for efficiency reasons. Therefore, due to the curse of dimensionality, it is likely that many co-located points in the high-dimensional space get separated by partition boundaries. This, in turn, badly impacts recall. For that reason, these techniques introduce overlap between partitions in order to compensate for boundary problems which tend to break the neighborhood in space.

In terms of scale, LSH suffers from the required postprocessing step of filtering out false positives, that requires either access to the actual descriptor information, or alternatively a large number of LSH hash tables for aggregation filtering, as shown in [7]. On the other hand, Spill-Tree and NV-Tree introduce such a large amount of additional redundancy, that the size of their index grows exponentially with the collection size, making their practical use difficult. The Spill-Tree has the additional constraint of requiring very many (thousands) processing cores for efficiency. None of these schemes can therefore work seamlessly with millions of images or billions of descriptors.

There is another line of work that is specifically dedicated to indexing local descriptor-based data sets. All those approaches are able to index a million images because they rely on some smart form of descriptor aggregation. A seminal approach is Video Google by Sivic et al. [16], where the many local descriptors of each image are transformed into a unique vector of higher dimensionality. Then, when processing one million images described with SIFT, only one million aggregated descriptors are in fact indexed, instead of about one billion. It is therefore quite easy for all the approaches along this line to cope with extremely large scales. For example, Jégou et al. [5] can index 10 million images in main memory, partly because local descriptors are aggregated per image. Other such approaches include [17, 13]. Overall, these schemes mostly work in main memory settings.

It is absolutely key to note, however, that these approaches are not tackling the same problem as the one addressed in this paper. These schemes do *image recognition* where the many local descriptors for each image are all used during the matching process, even once aggregated. Therefore, these schemes can fully benefit from the high redundancy of the descriptions to facilitate matching. Furthermore, the elegant aggregation schemes used are such that the resulting data collections fit in main memory and do not require accessing disks. Therefore, the search strategies they use can, with very little cost, analyze large amounts of data to produce high quality results (e.g., using multi-probe approaches [14, 6]). In contrast, this is not possible in practice when data is kept on disks, since analyzing large amount of data triggers costly I/Os which severely impact response time.

As we are interested in designing an indexing solution that returns the k-nn of single query points with good recall, we use SIFT to produce extremely large datasets. We do not query the index at the image level, with multiple descriptors returning neighbors that eventually vote for the most similar image. There is no redundancy in the retrieval process.

1.3 Contributions of this Paper

We are not aware of any previous high-dimensional indexing schemes that adequately address the very specific problem of finding efficiently and effectively, in a collection of few billion data points stored on disk, the approximate k-nearest neighbors of individual query points.

The main contribution of this paper is the presentation of an enhanced version of the NV-Tree, that precisely addresses the problem stated above. It is enhanced because previous versions of the NV-Tree included significant overlap in the index to compensate for partitioning problems in the high-dimensional space, while the version presented in this paper is entirely free of overlap. In [11], simulation results strongly indicated that the overlap could be removed by using more than one index and merging the results from the individual indices. As each non-overlapping index is much smaller, the overall performance is improved. This paper extends that work and makes the following major contributions:

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

Overall, our results show that the NV-Tree is an extremely scalable approximate indexing strategy, which yields results of acceptable quality. When the indexed data can be entirely kept in main memory, then the NV-Tree is extremely fast as very little computation is performed to find the k-nn. When the data collection to index is extremely large, then the NV-Tree gracefully adapts to efficient disk-based processing as a single disk access is required per query point. Note that the NV-Tree also includes specific procedures for fast concurrent insertions of new points to the index; these are not presented here due to space constraints.

This paper is organized as follows. Section 2 gives a short presentation of the NV-Tree, while Section 3 details the negative impact of the redundancy on the index size and creation time, as well as on the search process. Section 4 then presents three strategies to improve the quality of the results

returned by the NV-Tree, which were degraded by the lack of redundancy. Section 5 compares both versions of the NV-Tree and shows the overlap-free one outperforms the version with overlap. Then, Section 6 gives indications on the performance of the NV-Tree when indexing a collection made of 2.5 billion SIFT descriptors. Section 7 concludes the paper and discusses future work.

2. THE NV-TREE

The NV-Tree was originally proposed in [7]. It is a disk-based data structure, which builds upon a combination of projections of data points to lines and partitioning of the projected space. By repeating the process of projecting and partitioning, data is eventually separated into small partitions which can easily be fetched from disk with a single disk read, and which are highly likely to contain all the close neighbors in the collection. Since close descriptors may get separated by a partition boundary, the NV-Tree originally added redundancy by allowing the partitions to overlap. We focus here on index creation and retrieval and then briefly discuss the performance results from [7].

2.1 Index Creation

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

When the tree construction 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.[1] The projected values are then partitioned into disjunct sub-partitions based on their position on the projection line. For each pair of adjacent partitions, an overlapping partition, covering 50% of each partition, is created for redundant coverage of partition borders. Information about all the sub-partitions, such as the partition borders along the projection line, forms the root of the NV-Tree.

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

Two partitioning strategies co-exist inside the NV-Tree and proved to work best. First, partitioning is such that the distance between partition boundaries at each level of the tree is equal. The normal distribution of high-dimensional vectors gives partitions with very different cardinalities and dense areas are partitioned deeper than sparse areas. The partitioning strategy changes when reaching the lowest levels of the tree: when a sub-partition fits into six leaf nodes, then data is partitioned one more time according to an equal cardinality criterion (instead of being based on distances). This results in better leaf node utilization and a shallower tree, both of which lead to smaller space requirements.

[1]See [7] for details about projection line selection strategies.

2.2 Nearest Neighbor Retrieval

During query processing, the search first traverses the hierarchy of inner 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 node.

The leaf node is fetched into memory and the query descriptor is projected onto the projection line of the leaf node. The search then starts at the position of the query descriptor projection. The two descriptor identifiers on either side of the projected query descriptor are returned as the nearest neighbors, then the second two descriptor identifiers, etc. Thus, the $k/2$ descriptor identifiers found on either side of the query descriptor projection are alternated to form the ranked k approximate neighbors of the query descriptor.

2.3 Properties of NV-Trees

The NV-Tree indexing scheme has several key properties:

Single disk read: Since leaf nodes 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.

Ranking and no distance calculations: The NV-Tree returns approximate results in a ranked order. Returning a ranked result list has three major consequences. First, 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. Second, since no distance calculations are required, little CPU cost is incurred (scanning lists both directions), even for large collections. Third, as 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.

Consolidated result: False positives can largely be eliminated by aggregating multiple NV-Trees, which are built independently over the same collection. Since each NV-Tree is based on an independent pool of random lines, the contents of the ranked results are very likely to differ, except for the descriptors that are actual near neighbors.

2.4 Performance Summary

The performance analysis of [7] revealed some fundamental differences between the NV-Tree and its competitors. Since LSH is the most commonly cited method, and the strongest competitor in terms of performance, we now briefly outline the key differences between LSH and the NV-Tree.

For fairness reasons, we considered a version of LSH that, like the NV-Tree, only contains descriptor identifiers, but not the descriptors themselves. Note that without that modification, LSH does not cope at all with disk-based processing. The experimental settings were those used in Section 5; they are omitted here due to space considerations. The key observation, however, is that the ground-truth answers, that were used to measure quality, were composed of descriptors that are significantly closer to the query descriptors than their neighbors, regardless of their actual distance.

Using a single NV-Tree index, recall of about 66% was reported, with many false positives. When employing three NV-Trees, and returning only descriptors found in two indices out of three, most of the false positives were eliminated, while recall was improved slightly. In order to return results of the same quality, eight and twenty-four LSH indices were required, respectively, or three times more indices. Since each index requires one disk read at retrieval time, LSH required at least eight times more processing time than the NV-Tree. With LSH, however, the total disk requirements per index were about 2GB, compared to 50GB for each NV-Tree, due to the overlapping of partitions.

A detailed analysis of the results of individual descriptors revealed the underlying reason for the differences in effectiveness. Essentially, since LSH is based on an ϵ-distance query model, it cannot handle the variability in distances to the nearest neighbors; either ϵ is small and recall suffers, or ϵ is large and the number of false positives explodes.

3. THE CASE AGAINST REDUNDANCY

The NV-Tree uses overlapping partitions to achieve result quality despite the curse of dimensionality problems. Beside this positive impact on the quality of results, redundant storage impacts negatively the size of the index and its construction time, as well as the ability to use several trees to consolidate the results returned to the users. This section discusses these two negatives impacts and thus motivates the need to remove all redundancy from the NV-Tree as the only viable way to hit the billion scale.

3.1 Index Size and Construction Time

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

For the experiments in [7], a collection of $n = 180$ million descriptors (22.2GB) was indexed with a fixed fan-out of $f = 5$ and an approximate node filling rate of 70% leading to approximately $l = 4036$ descriptors on average per leaf node. Then $d = log_5(\frac{180\times10^6}{4036}) = 6.65$ and the redundancy factor is therefore $r = (\frac{2\times5-1}{5})^{6.65} = 49.9$, which leads to a total storage requirement of 180 million $\times$ 6 Bytes $\times$ 49.9 = 50.2GB, which is quite consistent with the reported storage requirements of about 50GB.

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

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

[2]Recall that leaf nodes only store descriptor identifiers.

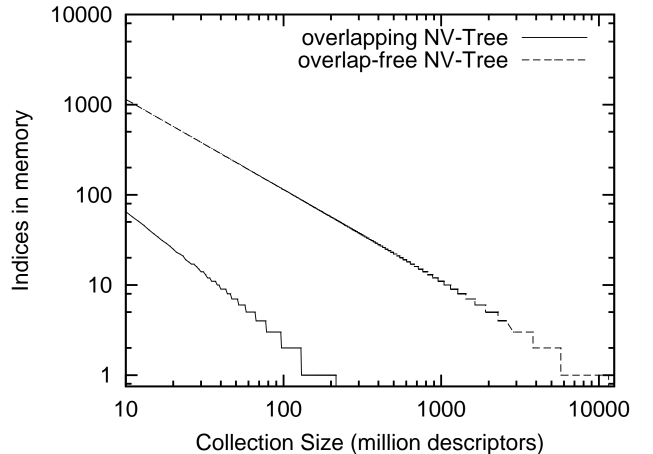

Figure 1: Comparison of overlapping and non-overlapping NV-Trees that fit within 64GB.

3.2 Searching in Multiple Trees

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

It is quite clear that the redundancy in the NV-Tree is in this case a serious problem as each index is very big. For example, each index used in the experimental results of [7] was about 50GB; no index could therefore be entirely cached, and a disk read was invariably needed per index to answer each query. In contrast, without redundancy, that same index would be about 1.1GB. It would easily fit in main memory, as well as a few additional NV-Trees, all together returning high quality results very efficiently since no disk reads would be involved.

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

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

4. OVERLAP-FREE NV-TREE

Removing overlapping from the NV-Tree is the key to scaling up the indexed collections. Not surprisingly, the quality of the results drops with removal of the overlapping; it de-

creases from about 66% (see [7]) to about 48% reported in the experiments below (see Figure 3).

This loss in quality does not affect neighbors that are close to the query point in the high-dimensional space (within a small ϵ threshold) and *well contrasted*, i.e., significantly closer than other data points. It does, however, affect neighbors that are well contrasted, but not close in terms of distance [7]. As we wish to preserve the key advantage of the NV-Tree over index structures based on ϵ thresholds, of being much less affected by the actual distance, we therefore propose to use the following three strategies to address this reduction in the result quality: (i) Creating additional NV-Trees; (ii) Creating deeper NV-Trees with smaller leaf nodes; and (iii) Reading additional leaf nodes.

Note that none of these strategies is particularly novel. It is their combined effect that is the novelty of this paper, however, as they make the NV-Tree truly scalable to very large collections. In fact, we show in Section 5 that the combined strategies improve recall beyond the original results.

4.1 Creating Additional Indices

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

4.2 Deeper Trees with Smaller Leaf Nodes

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

4.3 Reading Additional Leaf Nodes

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

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

5. COMPARATIVE EXPERIMENTS

We have implemented and evaluated all three adaptations to the NV-Tree data structure proposed above, and compared them to the performance results of the overlapping NV-Tree. Note that since the overlapping NV-Tree has already been shown to significantly outperform its competitors [7], this comparison is sufficient. First, Section 5.1 gives an overview of the experimental setup. Then, Section 5.2 presents performance measurements of the index construction and query performance, while Section 5.3 analyses the result quality.

5.1 Experimental Setup

We used a set of 179,443,881 128-dimensional SIFT descriptors that was obtained by extracting local features from an archive of about 150,000 high-quality press photos. In order to reduce the number of descriptors, each image was resized so that its larger edge was 512 pixels.

In order to evaluate the query performance of the NV-Tree, we extracted 500,000 query descriptors from transformed versions of images from our collection. These transformations are created using the Stirmark benchmarking tool [12] and consist, among others, of rotation, rescaling, cropping, affine distortions and convolution filters. It has been shown that SIFT descriptors cope very well with most of these distortions, meaning that a high percentage of corresponding descriptors are close in the high-dimensional space.

As ground-truth, we used the contrast-based ground truth defined in [7]. We used a sequential scan to calculate the 1,000 nearest neighbors for all 500,000 query descriptors. Then a neighbor n_i was included in the ground truth set when $d(n_{100}, q)/d(n_i, q) > 1.8$. This ground truth definition was shown to be the most meaningful method to measure high-dimensional nearest neighbor structures.[3] The resulting ground truth set contained 248,852 descriptors.

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

5.2 Indexing and Retrieval Performance

The major goal of the non-overlapping NV-Tree is to reduce the index creation time. The creation time for a single non-overlapping NV-Tree is about 2 hours, compared to more than 15 hours for the overlapping NV-Tree [7]. Furthermore, each non-overlapping tree consumes about 1 GB of disk space, compared to 50 GB for the overlapping NV-Tree.

The retrieval time is also reduced significantly due to the smaller index size. Figure 2 shows the retrieval time for

[3]For a detailed explanation of the contrast-based ground truth set, we refer to [7], where the set is defined and underpinned with a detailed analysis.

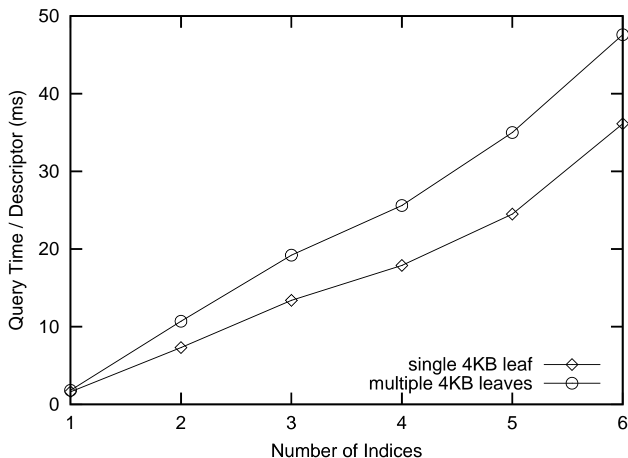

Figure 2: Retrieval time of NV-Tree configurations for a collection of 180 million descriptors.

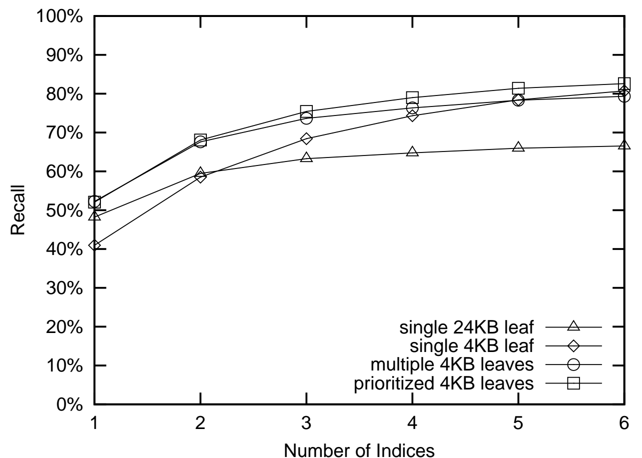

Figure 3: Recall of NV-Tree configurations for a collection of 180 million descriptors.

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

For our disks, a random disk read takes about 12.5 ms, yielding an expected retrieval time of 75 ms for 6 indices. The retrieval time is reduced, however, since the small size of the indices facilitates both buffering and disk locality. In contrast, the retrieval time of the much larger overlapping NV-Tree does indeed grow by about 12.5 ms per index, resulting in 37.5 ms for three indices [7].

Finally, Figure 2 shows that the overhead of reading additional leaf partitions is about 40% (with or without priority assignment), despite the fact that all leaf partitions are designed to fall within a single disk read. The first reason is that one logical I/O may end up as two physical I/Os. The second reason is due to disk buffering, as the likelihood of finding a single leaf in buffers is significantly higher than that of finding a whole range of subsequent pages.

5.3 Result Quality

We now study the impact of the proposed techniques on result quality. Figure 3 shows the recall of the different NV-Tree configurations. As before, the x-axis shows the number of indices used to answer queries, but the y-axis now shows the recall relative to the sequential scan.

Figure 3 shows that recall is relatively low for a single non-overlapping NV-Tree in any configuration. With additional indices, however, recall is improved significantly to the point where it exceeds the 65.8% recall reported in [7]. Furthermore, the figure shows that while the major quality improvements are caused by adding indices and a partitioning level, further improvements are seen by reading additional partitions, in particular with the priority-based scheme. Overall, we observe the best trade-off between performance and quality with 3 indices (75.4% total recall). Adding further indices yields up to 82.6% recall, but at a cost of higher retrieval time (×3) and larger database size (×2).

Detailed analysis shows that with a single non-overlapping index, many close neighbors are lost, which the overlapping NV-Tree can find easily due to the redundancy in the leaf partitions. By aggregating results from more than one non-overlapping NV-Tree, however, results are improved across all distance ranges; close neighbors are always found, and more distant neighbors are more likely to be found than with the overlapping NV-Tree. Note that, in contrast, the strategy of adding additional trees did not show any additional benefits in terms of recall for the overlapping NV-Tree [7].

6. LARGE-SCALE EXPERIMENTS

In this section we present our detailed experiments on a collection of 2.5 billion SIFT descriptors. First, Section 6.1 gives an overview of the experimental setup. Then, Section 6.2 presents query performance measurements, while Section 6.3 analyses the result quality.

6.1 Experimental setup

This descriptor collection has been obtained from 2.5 million images downloaded from the Flickr image sharing website, plus the images used in Section 5. The images were processed as before, resulting in a total of 2,485,568,191—nearly 2.5 billion—128-dimensional SIFT descriptors.

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

As the experimental setup used in the previous section was quite dated, we obtained a moderate server computer containing two Intel Xeon E5420 CPUs running at 2.50GHz, 12MB L2 Cache, and 32 GB of DDR 2 main memory, running at a clock speed of 667 Mhz. Due to the high storage requirements for such a large descriptor collection, as well as for performance, the server was equipped with 6 hard drives. Three 1.5 TB hard drives are used for storing the NV-Tree

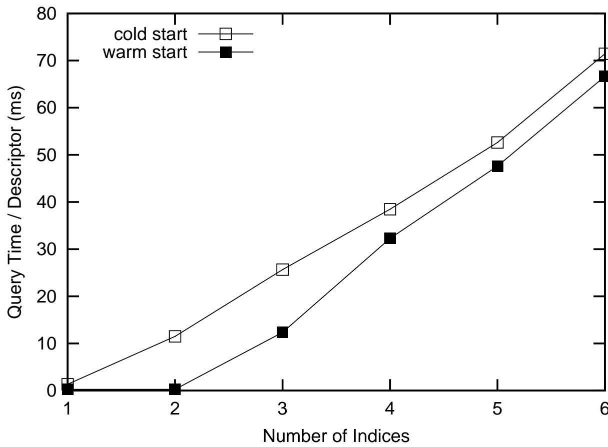

Figure 4: Retrieval time of NV-Tree configurations for a collection of 2.5 billion descriptors.

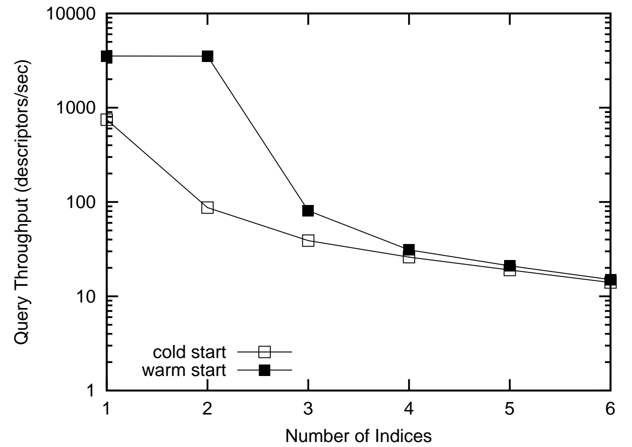

Figure 5: Throughput of NV-Tree configurations for a 2.5 billion descriptor collection.

indices, one for the operating system and related tasks, and two to store the descriptor collections and result files.

6.2 Indexing and Retrieval Performance

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

Figures 4 and 5 show the retrieval time and throughput, respectively, for the two buffering approaches. Note that only a single CPU core was used in each case. As the figures show, query processing is very efficient using one or two indices when buffers are full; each query descriptor requires only 0.3 ms of processing time, yielding a remarkable throughput of 3,500 queries per second. Once the indices do not fit into memory, however, retrieval time increases significantly, to 12 ms for three indices and up to 50 ms for six indices, with a corresponding reduction in throughput.

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

6.3 Result Quality

Figure 6 shows the recall for the large collection, compared to the collection used in the previous section. As the figure shows, recall is about 8% lower across the range of indices. Nevertheless, the recall is still quite acceptable, and nearly equivalent with the recall of a single overlapping NV-Tree for the 180 million descriptor collection.

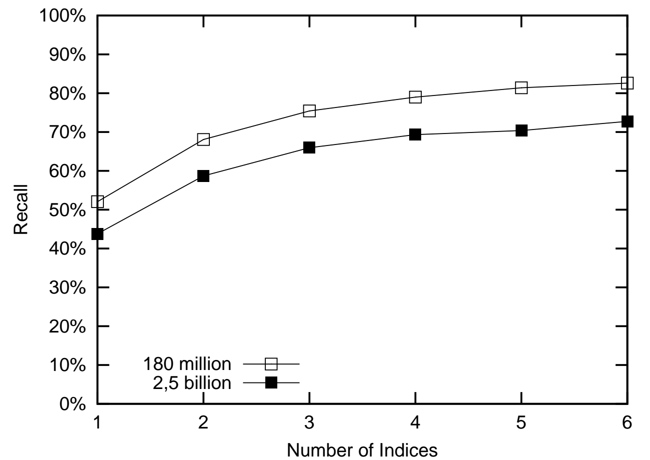

Figure 6: Recall of NV-Tree configurations for a collection of 2.5 billion descriptors.

7. CONCLUSION

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

There are many avenues for future work. As mentioned in the introduction, due to space constraints, this paper has focused solely on query processing. We are planning to show that insertions are also very efficient, as each descriptor is now only stored in one leaf partition. We are also preparing experiments using an order of magnitude larger collection. Finally, we have been exploring the application of the index structure to video identification [8].

8. REFERENCES

[1] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubsky, and P. Zezula. Building a web-scale image similarity search system. *Multimedia Tools and Applications*, 47, 2010.

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

[3] M. Douze, H. Jégou, H. Sandhawalia, L. Amsaleg, and C. Schmid. Evaluation of gist descriptors for web-scale image search. In *Proc. ACM CIVR*, Santorini, Greece, 2009.

[4] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proc. ACM SIGMOD*, San Diego, CA, USA, 2003.

[5] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *Proc. IEEE CVPR*, San Francisco, CA, USA, 2010.

[6] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *Proc ACM Multimedia*, Vancouver, BC, Canada, 2008.

[7] H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE TPAMI*, 31(5), 2009.

[8] H. Lejsek, H. Thormódsdóttir, F. H. Ásmundsson, K. Dadason, Á. T. Jóhannsson, B. T. Jónsson, and L. Amsaleg. Videntifier™ Forensic: Large-scale video identification in practise. In *Proc. MiFor*, Florence, Italy, 2010.

[9] T. Liu, A. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proc. NIPS*, Vancouver, BC, Canada, 2004.

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

[11] A. Ólafsson, B. T. Jónsson, L. Amsaleg, and H. Lejsek. Dynamic behavior of balanced NV-trees. *Multimedia Systems*, 17(2), 2011.

[12] 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.

[13] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proc. CVPR*, Minneapolis, MN, USA, 2007.

[14] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In *Proc. CVPR*, Anchorage, AK, USA, 2008.

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

[16] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Proc. ICCV*, Nice, France, 2003.

[17] Z. Wu, Q. Ke, M. Isard, and J. Sun. Bundling features for large scale partial-duplicate web image search. In *Proc. CVPR*, Miami, FL, USA, 2009.