Transactional Support for Visual Instance Search

Herwig Lejsek[1], Friðrik Heiðar Ásmundsson[1],
Björn Þór Jónsson[2,3], and Laurent Amsaleg[4]

[1] Videntifier Technologies
[2] Reykjavík University, Iceland
[3] IT University of Copenhagen, Denmark
[4] CNRS–IRISA, Rennes, France
bjorn@ru.is

Abstract. This article addresses the issue of dynamicity and durability for scalable indexing of very large and rapidly growing collections of local features for visual instance retrieval. By extending the NV-tree, a scalable disk-based high-dimensional index, we show how to implement the ACID properties of transactions which ensure both dynamicity and durability. We present a detailed performance evaluation of the transactional NV-tree, showing that the insertion throughput is excellent despite the effort to enforce the ACID properties.

1 Introduction

Visual instance search is the task of retrieving from a database of images the ones that contain an instance of a visual query. It is typically much more challenging than finding images that contain objects of the same category as the object in the query. If the query is an image of a shoe, visual instance search does not try to find images of shoes, which might differ from the query in shape, color or size, but tries to find images of the exact same shoe as the one in the query image.

Industry is very concerned with instance search, as various real world applications require such fine-grained image recognition capabilities. Forensics is a domain of choice, where identifying tiny similar visual elements in images is key to mapping out child abuse networks, or establishing links between various terrorism-related visual materials. Very fine-grained instance search is also involved in some copyright enforcement applications, and others.

Instance search challenges image representations as features extracted from the images must enable fine-grained recognition despite variations in viewpoints, scale, position, illumination, etc. While holistic image representations, where each image is mapped to a single high-dimensional feature vector, are sufficient for coarse-grained similarity retrieval, local features are needed for instance retrieval.

With local features, an image is mapped to a large set of high-dimensional vectors, allowing fine-grained recognition using the multitude of small visual matches between the query instance and the candidate images from the database. Extracting powerful local features from images has been widely studied and many

strategies exist to determine (i) where local features should be extracted ([21, 26]), and (ii) what information each local feature should encode ([22, 3]). All these strategies, however, result in a very large set of local features per image; an instance search system handling a few million images may need to manage a few billion local features. *Scalability* of the high-dimensional indexing techniques used in the context of instance search is therefore essential. Most state-of-the-art high-dimensional indexing solutions assume that the feature vector collection can always fit in memory. Experience from the data management community and from industry shows that this assumption is not valid, as data will eventually outgrow main memory capacity. Furthermore, with SSDs emerging as as viable middle ground between memory and hard disk drives, handling data that extends beyond main memory should be reconsidered.

Image collections grow (often rapidly) as time goes by, and it is important to ensure that the instance search engines probes up-to-date collections. Very few proposals from the literature address dynamicity, however, and most of them can only expand the indexed collection through a complete reconstruction of the index. In the real world, halting a system for re-indexing is not an option; instead, new data items must be dynamically inserted into the index while the system is running. Even the recently proposed Lambda Architecture [20], which separates handling of very recent data from older data, requires dynamic consistency of index maintenance for recent data. Furthermore, resisting failures and enforcing *durability* of the indexed data is very important. Losing the features upon failure or experiencing extended downtime for reconstruction of indices are not acceptable options. Storing the high-dimensional index on disk is thus not only necessary for scalability, but also for the dynamic integrity of the index.

Key Requirements Based on the discussion above, we have identified the following four key requirements a high-dimensional indexing solution dedicated to enabling scalable identification of similar visual instances must meet:

R1 The index must make *efficient use of all available storage resources*, main memory, solid-state devices or hard disks.
R2 The index must offer *stable query processing performance*, so that it can be used as a component of an industry-scale processing chain.
R3 The index must support *dynamic insertion methods* so that the indexed collection can grow while concurrent retrievals are performed.
R4 The index must support the *ACID properties of transactions* (Atomicity; Consistency; Isolation; and Durability), which guarantee the integrity of index maintenance, as well as correct recovery in case of system failures.

Most state-of-the-art methods, such as product quantization, focus on compressing data into main memory. Whereas these methods often provide some guarantees on quality, they neither consider updates nor provide guarantees on query processing performance, thus failing with requirements **R1** to **R4**.

We have previously proposed the NV-tree, an approximate high-dimensional indexing method that is designed from a data management perspective and can

thus deal with feature collections that outgrow main memory [14, 15]. When there are more image features that can fit in RAM, the NV-tree guarantees using at most a single disk read per index to get the approximate results. By providing a disk-based query performance guarantee for the NV-tree, requirements **R1**, and **R2** above are satisfied.

In [14], an insertion procedure is described and evaluated for an early version of the NV-tree. Subsequent works, which propose significant improvements to the NV-tree, also discuss dynamic maintenance of the index [15, 27]. In order to make the NV-tree a fully transactional dynamic index, we propose transactional index maintenance procedures, and show that the resulting extended NV-tree supports both **R3** and **R4**.

Contributions This article makes the following major contributions to the domain of scalable high-dimensional indexing for visual instance search:

1. We show how to adapt the NV-tree to transactional processing of insertions and deletions, guaranteeing the well-known ACID properties of transactions.
2. We evaluate insertion performance in this transactional setting and show that insertion throughput is excellent: when the index fits in memory, the index can take full advantage, but even in the disk-bound case each insertion requires only a small fraction of a disk write on average.

The technology described in this article is already in use at Videntifier Technologies, one of the main players in the forensics arena with technology deployed at such clients as Interpol. Their search engine targets fine-grained visual instance search as it is used for investigations that, for example, aim to dismantle child abuse networks. The search engine can index and identify very fine-grained details in still images and videos from a collection of 150 thousand hours of video, typically scanning videos at 40x real-time speed, and about 700 hours of video material can be dynamically added to the index every day.

2 Related Work

Only *approximate* high-dimensional indexing solutions remain efficient at very large scale. Approximate indexing methods trade quality off for response time, and follow three different major directions. Due to space constraints we outline only the most scalable of these methods here; for more details see [16].

Quantization One line of work is based on indexing data clusters such as the hierarchical k-means decomposition of the data collection: Voronoi cells are created to partition and store the high-dimensional vectors, and the cells are organized as a multi-level tree to facilitate traversal and improve response time [7]. Many variants of this basic idea have been proposed (e.g., see [17, 29, 30]). One algorithm from this category has been extended to cope with collections of up

to 43 billion feature vectors, using distributed processing with "big data" techniques such as Spark [24, 9]. A more sophisticated indexing method, still using data clusters at its core, is called product quantization [10]. Product quantization decomposes the high-dimensional space into low-dimensional subspaces that are indexed independently. This produces compact code words representing the vectors that, together with an asymmetric approximate distance function, exhibit good performance for a moderate memory footprint. Several variants of product quantization have been published; in particular, Sun et al. [32] proposed an indexing scheme based on product quantization that uses ten computers to fit in memory the 1.5 billion images collection they index. The inverted multi-index by Babenko and Lempitsky [1] uses product quantization at its core but achieves a much denser subdivision of the space by using multiple inverted indices. The experiments reported in [1], using the BIGANN dataset [11] that contains one billion SIFT descriptors, show that the approach can determine short candidate lists with superior recall. They have recently extended their method for deep learning features [2]. None of these methods have proposed methods to support either dynamic updates or ACID properties of transactions.

Hashing A second line of work developed around the idea of hashing. The earliest notable hashing-based method proposed was Locality Sensitive Hashing (LSH) [5]. Essentially, LSH uses a large number of hashing functions to project the high-dimensional vectors onto segmented random lines. At query time, these hash tables are probed with the query vector, and the candidates from all hash tables are then aggregated to find the true neighbors. The performance of such hashing schemes is highly dependent on the quality of the hashing functions. Hence, many approaches have been proposed to improve hashing [28, 35, 12]. As with the quantization-based methods, none of these hashing-based methods support either dynamic updates or ACID properties of transactions.

Tree Structures A third approach is based on the idea of a search tree structure. The NV-tree is one proponent of this group. Fagin et al. [6] introduced the concept of median rank aggregation. They project the entire data collection on multiple random lines and index the ranked identifiers of the data points along each line, discarding the actual feature vectors. This ranking turns the high-dimensional vectors into simple sets of values which are inserted to B^+-trees. These B^+-trees are probed at search time, and the nearest neigbors of the query are returned according to their aggregated rankings. The major drawback of that algorithm is the excessive search across the individual B^+-trees [6].

Tao et al. [33] proposed another method for accessing high-dimensional data based on B^+-trees, called the locality sentitive B-tree or LSB-tree. The LSB-tree approach inherits some of the properties of LSH, but in addition projects the hashed points onto a Z-order curve. Quality guarantees can be enforced using multiple LSB-trees in combination, forming an LSB-forest [33].

Muja and Lowe [25] proposed, via the FLANN library, a series of high-dimensional indexing techniques based on randomized KD-trees, k-means in-

dexing and random projections. Another approach in the category of search trees is the Metric tree [34]; a variant named Spill-tree 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 [18].

Many of the tree-based methods are either based on B^+-trees, which support dynamic updates and transactions, or have proposed specific methods to address updates. Most of these methods, however, have only been used for relatively small collections. As far as we are aware, this article presents the first study that is considering updates at a scale of a billion features or more, and also the first study to consider transactional properties at this scale.

3 The NV-tree

The NV-tree [14, 15] is a disk-based high-dimensional index, based 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 separated into small partitions which can be easily fetched from disk with a single read, and which are likely to contain all the close neighbors in the collection. We briefly describe the NV-tree creation process, its search procedure, its dynamic insert process and and then enumerate some salient properties of the NV-tree.

Index Creation Overall, an NV-tree is a tree index consisting of a hierarchy of small *inner nodes*, which guide the vector search to the appropriate leaf node, and larger *leaf nodes*, which contain references to actual vectors. The leaf nodes are further organised into *leaf-groups* that are disk I/O units, as described below.

When tree construction starts, all vectors from the collection are first projected onto a single projection line through the high-dimensional space ([14] discusses projection line selection strategies). The projected values are then partitioned in 4 to 8 partitions based on their position on the projection line. Information about the partitions, such as the partition borders along the projection line, forms the first inner node of the tree—the root of the tree. To build the subsequent levels of the NV-tree, this process of projecting and partitioning is repeated recursively for each and every partition, using a new projection line for each partition, thus creating the hierarchy of smaller and smaller partitions represented by the inner nodes.

At the upper levels of the tree, with large partitions, the partitioning strategy assigns equal distance between partition boundaries at each level of the tree. The partitioning strategy changes when the vectors in the partition fit within 6×6 leaf nodes of 4 KB each. In this case, all the vectors from that partition are partitioned into a *leaf-group* made of (up to) 6 inner nodes, each containing (up to) 6 leaves. In this leaf-group, partitioning is done according to an equal cardinality criterion (instead of an equal distance criterion). Finally, for each leaf node, projection along a final random line gives the order of the vector identifiers and the ordered identifiers are written to disk. It is important to note that the vectors themselves are *not* stored; only their identifiers.

Indexing a collection of high-dimensional vectors with an NV-tree thus creates a tree of nodes keeping track of information about projection lines and partition boundaries. All the branches of the tree end with leaf-groups with (up to) 36 leaf nodes, which in turn store the vector identifiers.

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 vector 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 vector until the search reaches a leaf-group, which is then fully fetched into RAM, possibly causing one single disk I/O. Within that leaf-group, the two nodes with center-point closest to the projection of the query vector are identified. The best two leaves from each of these two nodes are then scanned in order to form the final set of approximate nearest neighbors, with their rank depending on their proximity to the last projection of the query vector. The details of this process can be found in [15].

While the NV-tree is stored on disk, the hierarchy of inner nodes is read into memory once query processing starts, and remains fixed in memory. The larger leaf nodes, on the other hand, are read dynamically into memory as they are referenced. If the NV-tree fits into memory, the leaf nodes remain in memory and disk processing is avoided, but otherwise the buffer manager of the operating system may remove some leaf nodes from memory.

Insertions Insertion to NV-tree leaf nodes proceeds as follows. First, the leaf node where to insert a new vector identifier is identified. The position within that leaf is also determined and the insert is performed if the leaf is not full. As for most dynamic data structures, leaf nodes at index creation time are not filled completely (they are between 50% and 85% full, and about 70% full on average) in order to leave space for such insertions. A filled leaf node must be split in order to provide more storage capacity within the tree. During a split operation, a leaf-group is considered as a unit, and all the features of the leaf-group are re-organized using the same process as during index construction. In particular, when the size of the leaf-group exceeds the capacity of 6×6 leaf nodes, the group is split into 4 to 8 new leaf-groups, depending on the distribution of the features.

During leaf-group re-organization, new projection lines are chosen for internal nodes and the new leaf nodes. As leaf nodes only contain vector identifiers, the vectors must be retrieved from disk for re-projection. In [13], it is shown that the most efficient option for handling re-projections is to maintain an independent feature database for each NV-tree, organized in the same manner as the leaf-groups, to directly read the relevant features.

Properties of NV-trees The experiments and analysis of [15] show that the NV-tree indexing scheme has the following properties:

- *Random Projections and Ranking:* The NV-tree uses random projections to turn multi-dimensional vectors into single-dimensional values indexed by B^+-trees. Efficient implementations of dynamic B^+-trees are well known. The NV-tree does not fetch full vectors from disk to compute distances. In contrast, ranking is used, which basically amounts to scanning a list.
- *Single Read Performance Guarantee:* As leaf-groups have a fixed size, the NV-tree guarantees query processing time of a single read regardless of the size of the vector collection. Larger collections need deeper NV-trees but the intermediate nodes fit easily in memory and tree traversal cost is negligible.
- *Compact Data Structure:* The NV-tree stores in its index the identifiers of the vectors, not the vectors themselves. This amounts to about 6 bytes of storage per vector on average. The NV-tree is thus a very compact data structure. Compactness is desirable as it maximizes the chances of fitting the tree in memory, thus avoiding disk operations.
- *Consolidated Result:* Random projections produce numerous false positives that can be almost all eliminated by an ensemble approach. Aggregating the results from a few NV-trees, which are built independently over the same collection, dramatically improve result quality.

The NV-tree, however, is not a transactional index. The next section specifies mechanisms to enforce the ACID properties of transactions within the NV-tree.

4 Transactional NV-tree

Large collections of media objects, and the corresponding collections of high-dimensional vectors, are typically dynamic and require efficient insertions. For the typical web-scale application of visual instance search, however, it is safe to assume that (a) updates are made centrally, and (b) that throughput is more important for this update thread than response time. For these applications, it is feasible to batch insertions such that only one insertion thread is running at each time, which simplifies the implementation of the insertion process. Of course, however, insertions and searches must run concurrently.

This section focuses on insertions to the NV-tree index. We outline the insertion operation, then describe enforcement of the ACID properties of transactions (Atomicity; Consistency; Isolation; and Durability), and finally consider the correctness and performance of the proposed methods. Note that while deletions will be rare in practice, they can be implemented using techniques very similar to those implementing insertions. We therefore briefly describe the differences for deletions, where appropriate. Updates are implemented as feature deletions followed by feature insertions.

Due to serialization of inserts, two insertion transactions will never conflict, which means that a simple locking mechanism based on tree-traversals is sufficient to enforce *isolation*. Because insertions are never aborted and they never deadlock, ensuring *atomicity* is only needed when the system crashes. Furthermore, since at most one insert transaction is running concurrently, enforcing

durability is greatly simplified. Finally, since there are no constraints on the vectors, as such, the notion of *consistency* simply implies that the results always reflect the status after the last committed transaction.

We start by considering isolation and consistency for a single NV-tree. We then consider atomicity and durability, before addressing some practical issues relating to using multiple NV-trees.

Isolation and Consistency Isolation is implemented by adapting a standard locking algorithm from the B^+-tree literature [8, 31]. A search thread starts by obtaining a read lock on the root of the NV-tree. Before accessing a child node, the thread must obtain a read lock on that node. At that point, the lock on the parent can be released. Finally, the leaf-group selected for retrieval is locked and only released after all necessary identifiers have been retrieved from the leaves. Note that locks are implemented using pthread mutexes; each internal node contains the mutexes for all its children and the leaf-groups are locked as a unit since they are treated as a unit during both retrieval and node splits. As the overhead of obtaining mutexes is low, locking is always activated.

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

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

Deletions are implemented in the same manner as insertions, except that a list of deleted media items is maintained to avoid returning partially deleted items; when all feature vectors from a media item have been deleted, it can be removed from this list.

Atomicity and Durability For atomicity and durability, we adopt the standard write-ahead logging (WAL) protocol [23]. The WAL protocol uses a transaction log (or write-ahead log) which contains sufficient information to recover in case of failures. The WAL protocol has two rules to ensure the correctness of transactions:

1. The log entry for any modification must be written to disk before the modified data is written to disk.
2. All log entries for a transaction must be written to disk before the transaction can be committed.

The first rule—sometimes called the *undo* rule—ensures that any change written to the disk before it is committed can later be removed from the database, thus supporting atomicity. The second rule—the *redo* rule—ensures that committed changes can be redone in case of crashes, thus ensuring durability.

Each split can result in a large number of disk operations and splits are therefore heavily buffered. It is best for performance to manage multiple log files where log records can be appended independently and in parallel. There is one log file per NV-tree plus a global log file for the correctness of the overall recovery process.

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

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

Practical Issues with Multiple NV-trees When inserting to multiple NV-trees, each tree should preferably be located on a separate hard drive (as should the log files) so that the full write-back capacity of the disks can be used for the leaf-group DB thread. In order to fully use the capacity of the disks, however, it is important to decouple the insertion process (as well as logging and checkpointing) for each NV-tree. Each NV-tree can thus be inserting from a different transaction, but they must all process the transactions in the same order. Since transactions may progress differently across different trees, more than one uncommitted transaction may have inserted vectors to some trees before a crash. Due to the ordering of transactions, however, the last NV-tree to finish a transaction decides the commit time and transactions will therefore commit in the same order, and all the techniques described above are unaffected by this change. Using decoupling, disk utilization was improved from about 40% up to 75% to 80%, without violating the previously described ACID properties.

Correctness and Recovery Performance Since our techniques are built on standard building blocks from the database literature, which have been shown to enforce the ACID properties, a formal proof of correctness is beyond the scope of this paper. In the following, however, we give a brief outline of how such a proof would be structured.

A sufficient condition for enforcing isolation is *serializability*. Recall that we assume that insertion transactions stem from a single, serialized thread. Then the only conflicts that can arise are between this single insertion transaction and the (potentially many) retrieval transactions. As we use standard B^+-tree locking for the data structure consistency, which is known to enforce serializability, and further ensure that retrieval transactions can only see insertions from transactions that committed before the retrieval started, isolation is fully enforced. And, as discussed above, since there are no constraints on the vectors, isolation is sufficient to enforce consistency for the class of applications considered here.

By definition, the WAL protocol enforces both atomicity and durability. The fact that the log is stored in multiple files does not change this property, as long as sufficient information is stored in the log entries to redo operations in the correct order. As described in detail above, the recovery operations have been carefully ordered to ensure correctness. The proposed method therefore enforces both atomicity and durability.

Proving the correctness of the *implementation* of our method, on the other hand, is of course extremely difficult, if at all possible. The implementation has been tested very methodically, however, by pausing operations in certain places and crashing the computer; in all cases has recovery been successful. The recovery performance depends on the frequency of the checkpoints, but with reasonable checkpoint frequency the database is always fully recovered within a matter of minutes even with very large collections.

5 Performance of Index Maintenance

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

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

To illustrate these two cases, we used a machine with only 32GB of main memory. We used a small subset of a very large collection of images from Flickr to first compute 36 million SIFT vectors [19], which were indexed with three

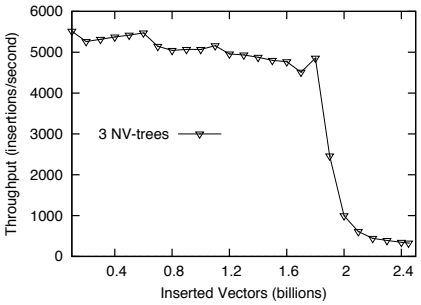

Fig. 1. Insertion throughput.

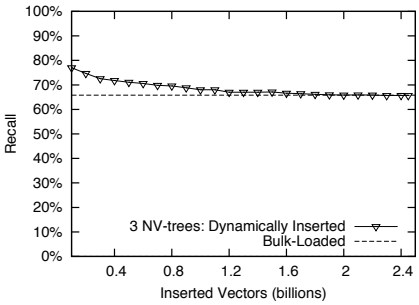

Fig. 2. Retrieval quality.

NV-trees. This is a tiny collection which can be indexed very quickly, and the resulting NV-trees together occupy slightly more than 500MB. We then ran sequences of 1,000 insertion transactions, where each transaction inserted 100,000 new vectors into the three NV-trees, which means that each sequence inserted 100 million new vectors. We then observed the time taken for each sequence to complete. We repeated this process and ran multiple sequences until each NV-tree contained nearly 2.5 billion vectors, occupying about 328 GB each.

Insertion Throughput Figure 1 shows the evolution of the insertion throughput (measured by vectors inserted per second) for the duration of this workload. In the beginning of this workload, all three NV-trees fit into main memory and the throughput is excellent, around five thousand vectors per second. After running 18 such transactions, thus inserting 1.8 billion vectors, the 3 NV-trees no longer fit in main memory. After that point, Figure 1 clearly shows the insert behavior corresponding to the second case discussed above, where the rate of inserts slows down significantly due to conflicting disk operations. It should be noted, however, that with throughput of 500 vectors per second, each insertion only takes 2 ms, which is significantly less than one disk operation per insertion, even though the descriptors are inserted to three NV-trees simultaneously.

The most important aspect of this experiment is not the reduced performance of inserts after 1.8 billion vectors have been inserted and the index no longer fits in memory; by adding more memory, larger indices can be stored in RAM. Rather it is the fact that even when the collection no longer fits in memory, and must be stored on traditional HDDs, dynamic maintenance of the index is still possible as the insertion throughput degrades gracefully. And, of course, performance of index maintenance could be vastly improved if SSDs were used instead of HDDs.

Evolution of Retrieval Quality To evaluate the query performance of the NV-tree, we borrow the ground truth defined by [14]. A sequential scan was used to determine the 1,000 nearest neighbors of 500,000 query vectors, all coming from a very large collection of SIFTs. The resulting 500M neighbors were then

analysed to identify 248,212 vectors as being meaningful nearest neighbors of the query points (as defined by [4]).

To reuse that workload, we included these 248,212 vectors in the database of 36 million other vectors used previously. Once this database was created, we ran the same 500,000 queries as in [14] and computed their recall, i.e., we counted how many of these 248,212 ground truth vectors were found. We repeated that same workload after every insertion transaction (of 100 million vectors), to observe how the quality of the answers evolves as the database grows.

Figure 2 plots the recall percentage from the 500,000 queries described above, as the collection grows in size. The figure shows a configuration where the results are aggregated from three NV-trees. As the figure shows, recall drops slowly as the collection grows, which was expected. For comparison, the figure also contains a dashed line indicating the result quality when the NV-trees for the 2.5 billion vectors are constructed from scratch via bulk loading. As the figure shows, the results for the dynamically created NV-trees and the bulk-loaded NV-trees are identical, meaning that dynamicity has no impact on result quality.

6 Conclusion

Visual instance search is the task of retrieving from a database of images the ones that contain an instance of a visual query. So far, only local image features are powerful enough to support such fine-grained recognition. Recent progress in approximate high-dimensional indexing has resulted in approaches handling several hundred million to a few billion high-dimensional vectors with excellent response times. These methods typically rely on residing in main memory for performance. We argue, however, that data quantity will always win over memory capacity in the long term. Therefore, high-dimensional indexing solutions that are truly concerned with the scalability of the feature collections they manage must address collection sizes beyond RAM capacity and efficiently utilize disks for extending storage.

Furthermore, we argue that scalability is not the only challenge that must be met as high-dimensional indexing methods must also provide dynamicity—the ability to cope with on-line insertions of features into the indexed collection, and durability—the ability to recover from crashes and avoid losing the indexed data if a failure occurs. As far as we know, no nearest neighbor algorithm published so far is able to cope with all three requirements: scale, dynamicity and durability.

In this article, we have extended an existing disk-based high-dimensional index, the NV-tree, such that it enforces the ACID properties of transactions. Experiments show that with our implementation dynamic inserts can be efficiently managed: when the index fits in memory, performance is excellent, but when the index no longer fits in memory, performance degrades very gracefully.

Indeed, the technology described in this paper is in use at Videntifier Technologies, one of the main players in the forensics arena, with technology deployed at such clients as Interpol. Their search engine is currently able to index and identify videos (at about 40x real time) from a collection of nearly 150 thou-

sand hours of video, and about 700 hours of video material can be dynamically inserted to the index every day.

References

1. Babenko, A., Lempitsky, V.S.: The inverted multi-index. IEEE Transactions on Pattern Analysis and Machine Intelligence 37(6), 1247–1260 (2015)
2. Babenko, A., Lempitsky, V.S.: Efficient indexing of billion-scale datasets of deep descriptors. In: Proc. CVPR. Las Vegas, NV, USA (2016)
3. Bay, H., Ess, A., Tuytelaars, T., Gool, L.V.: Speeded-up robust features (SURF). Computer Vision and Image Understanding 110(3), 346 – 359 (2008)
4. Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Proc. ICDT. Jerusalem, Israel (1999)
5. Datar, M., Indyk, P., Immorlica, N., Mirrokni, V.: Locality-sensitive hashing using stable distributions. MIT Press, Cambridge MA, USA (2006)
6. Fagin, R., Kumar, R., Sivakumar, D.: Efficient similarity search and classification via rank aggregation. In: Proc. ACM SIGMOD. San Diego, CA, USA (2003)
7. Fukunaga, K., Narendra, P.M.: A branch and bound algorithms for computing k-nearest neighbors. IEEE Transactions on Computers 24(7), 750–753 (1975)
8. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco, CA, USA (1993)
9. Guðmundsson, G.Þ., Amsaleg, L., Jórnsson, B.Þ., Franklin, M.J.: Towards engineering a web-scale multimedia service: A case study using Spark. In: Proc MMSys. Taipei, Taiwan (2017)
10. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence 33(1), 117–128 (2011)
11. Jégou, H., Tavenard, R., Douze, M., Amsaleg, L.: Searching in one billion vectors: re-rank with source coding. In: Proc. ICASSP. Prague, Czech Republic (2011)
12. Jin, Z., Hu, Y., Lin, Y., Zhang, D., Lin, S., Cai, D., Li, X.: Complementary projection hashing. In: Proc. ACM ICCV. Barcelona, Spain (2013)
13. Jónsson, B.Þ., Amsaleg, L., Lejsek, H.: SSD technology enables dynamic maintenance of persistent high-dimensional indexes. In: Proc. ACM ICMR. New York, NY, USA (2016)
14. Lejsek, H., Ásmundsson, F.H., Jónsson, B.Þ., Amsaleg, L.: NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(5), 869 – 883 (2009)
15. Lejsek, H., Jónsson, B.Þ., Amsaleg, L.: NV-Tree: Nearest neighbours at the billion scale. In: Proc. ACM ICMR. Trento, Italy (2011)
16. Lejsek, H., Jónsson, B.Þ., Amsaleg, L., Ásmundsson, F.H.: Dynamicity and durability in scalable visual instance search. arXiv abs/1805.10942 (2018), `https://arxiv.org/abs/1805.10942`
17. Li, C., Chang, E., Garcia-Molina, H., Wiederhold, G.: Clindex: Clustering for approximate similarity search in high-dimensional spaces. IEEE Transactions on Knowledge and Data Engineering 14(4), 792–808 (2002)
18. Liu, T., Moore, A., Gray, A., Yang, K.: An investigation of practical approximate nearest neighbor algorithms. In: Proc. NIPS. Vancouver, BC, Canada (2004)

19. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal on Computer Vision 60(2), 91–110 (2004)
20. Marz, N., Warren, J.: Big Data: Principles and best practices of scalable real-time data systems. Manning Publication co (2015)
21. Mikolajczyk, K., Tuytelaars, T., Schmid, C., Zisserman, A., Matas, J., Schaffalitzky, F., Kadir, T., Gool, L.V.: A comparison of affine region detectors. International Journal on Computer Vision 65(1), 43–72 (2005)
22. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(10), 1615–1630 (2005)
23. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems 17(1), 94–162 (1992)
24. Moise, D., Shestakov, D., Guðmundsson, G.Þ., Amsaleg, L.: Indexing and searching 100M images with Map-Reduce. In: Proc. ACM ICMR. Dallas, TX, USA (2013)
25. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Transactions on Pattern Analysis and Machine Intelligence 36(11), 2227–2240 (2014)
26. Nowak, E., Jurie, F., Triggs, B.: Sampling strategies for bag-of-features image classification. In: Proc. ECCV. Graz, Austria (2006)
27. Ólafsson, A., Jónsson, B.Þ., Amsaleg, L., Lejsek, H.: Dynamic behavior of balanced NV-trees. Multimedia Systems 17(2), 83–100 (2011)
28. Paulevé, L., Jégou, H., Amsaleg, L.: Locality sensitive hashing: A comparison of hash function types and querying mechanisms. Pattern Recognition Letters 31(11), 1348–1358 (2010)
29. Philbin, J., Chum, O., Isard, M., Sivic, J., Zisserman, A.: Object retrieval with large vocabularies and fast spatial matching. In: Proc. CVPR. Minneapolis, MN, USA (2007)
30. Philbin, J., Chum, O., Isard, M., Sivic, J., Zisserman, A.: Lost in quantization: Improving particular object retrieval in large scale image databases. In: Proc. CVPR. Anchorage, AK, USA (2008)
31. Srinivasan, V., Carey, M.J.: Performance of B-tree concurrency control algorithms. In: Proc. ACM SIGMOD. Denver, Colorado, USA (1991)
32. Sun, X., Wang, C., Xu, C., Zhang, L.: Indexing billions of images for sketch-based retrieval. In: Proc. ACM Multimedia. Barcelona, Spain (2013)
33. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. ACM Transactions on Database Systems 35(3), 20:1–20:46 (2010)
34. Uhlmann, J.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters 40(4), 175–179 (1991)
35. Zhang, D., Agrawal, D., Chen, G., Tung, A.: HashFile: An efficient index structure for multimedia data. In: Proc. ICDE. Hannover, Germany (2011)