Size Matters: Exhaustive Geometric Verification for Image Retrieval Accepted for ECCV 2012

Henrik Stewénius, Steinar H. Gunderson, and Julien Pilet

Google Switzerland

Abstract. The overreaching goals in large-scale image retrieval are bigger, better and cheaper. For systems based on local features we show how to get both efficient geometric verification of every match and unprecedented speed for the low sparsity situation.
Large-scale systems based on quantized local features usually process the index one term at a time, forcing two separate scoring steps: First, a scoring step to find candidates with enough matches, and then a geometric verification step where a subset of the candidates are checked.
Our method searches through the index a document at a time, verifying the geometry of every candidate in a single pass. We study the behavior of several algorithms with respect to index density—a key element for large-scale databases. In order to further improve the efficiency we also introduce a new new data structure, called the counting min-tree, which outperforms other approaches when working with low database density, a necessary condition for very large-scale systems.
We demonstrate the effectiveness of our approach with a proof of concept system that can match an image against a database of more than 90 billion images in just a few seconds.

1 Introduction

The growth of content-based image retrieval systems using local features has been driven by better techniques, faster hardware and a knowledge that these systems work well at scale. In this work we only focus on vocabularized local features and ignore the rich literature on other methods. The largest impact on the size growth came from the introduction of visual vocabularies [1], a technique which has been refined in [2–8]. This works aims to further increase the power of this class of systems.

Nowadays, several companies offer large-scale, content-based image retrieval services [9–13]. For example, TinEye [13] reached two billion images in 2011. Unfortunately very little is known about the actual systems.

According to the available literature, a typical vocabulary-based system extracts features from images to produce an inverted index mapping quantized features to a list of images exhibiting the same quantized value—a posting list.

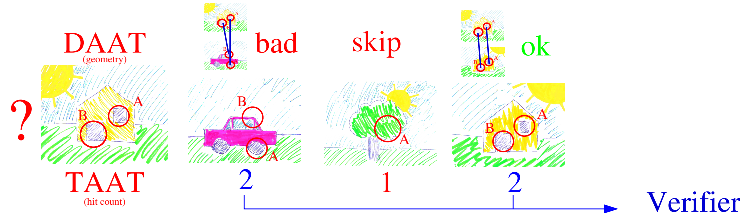

Fig. 1. Features A and B detected in the query image (top) are present in three images. In Term at a Time (TAAT) we first process the posting list for A and then the posting list for B, and accumulate scores. The top candidates are then geometrically verified. In Document at a Time (DAAT) evaluation, posting lists are traversed in parallel, and we do geometric verification for all documents having a sufficient number of matches.

At query time, features are extracted from the query image and quantized. The corresponding posting lists are read, and votes are added for each document in the list, usually using weights computed from features and document statistics. The subset of the documents with the best scores are selected for geometric verification, and a final score is computed, measuring how many and how well query features align with those from the candidate.

We believe that it is important to distinguish systems designed for high quality but where the cost per document is less of a factor, and systems which sacrifice quality in order to go to an extreme scale. One important knob for this is feature counts and vocabulary size.

When scaling such an approach to billions of images, one of two following problems occur, depending on the database density. On one hand, if the feature count per image is high and the vocabulary small, the index density is high and the number of "random" hits in the first stage results in a large number of candidate documents for which the match geometry would ideally be verified. This verification can become expensive, since it typically involves fetching additional data for each tested document. If, on the other hand, the feature count per document is low and the vocabulary size is large, the density is low, and the average number of hits per document will be small, making the vote accumulator very sparse. We show that both low and high densities cause problems for such approaches; in the former case because one has to track many zeroes, and in the latter because the fastest voting structures use too much memory and one has to fall back to sparser, slower ones.

To address these two problems, we propose a novel scoring method that unifies candidate extraction and geometric verification. Our approach efficiently rejects candidates with less than four matches, and validates the geometry of all remaining candidates. The final score is computed in a single step, considering a single document at a time, with the geometry of all its matches, as opposed to pre-scoring with a weighted hit count. In text retrieval, these two approaches are called Document at a Time (DAAT) and Term at a Time (TAAT) [14], or term-ordered evaluation and document-ordered evaluation [15].

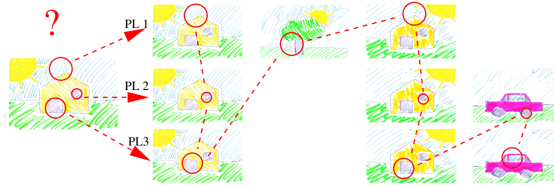

Fig. 2. Document at a Time (DAAT) evaluation: Posting lists are traversed in parallel, the next element is always lowest document identifiers of the remaining documents. Geometric verification can be done for all images with a minimal number of matches.

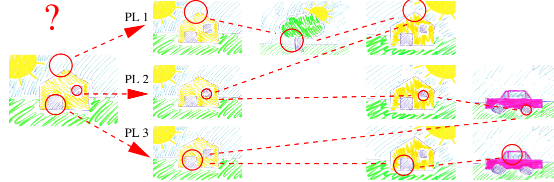

Fig. 3. Term at a Time (TAAT) evaluation: Posting lists are traversed one by one; scores are usually computed by accumulation.

To implement our DAAT approach in a computationally efficient manner, we propose a new data structure we call the *counting min-tree* (CMT). It can be used to merge a set of sorted posting lists and count how many lists each document appears in. The counting min-tree allows our DAAT approach to quickly process large databases while extracting the data needed for geometric verification for all remaining candidates.

We compare the performance of TAAT, heap-based DAAT, and CMT-based DAAT under varying database density. We show that our counting min-tree outperforms other approaches.

We validate our approach on a system that quickly retrieves images from an index containing more than 90 billion images. The images were obtained by crawling the web.

2 Related Work

Document at a Time scoring was introduced in 1995[14] for text search and permits retrieval-time checking of the relative position of the query terms in the result. Kaskiel et al. [15] showed that document-ordered query processing is efficient for retrieving text passages. Word positioning in text is the one-dimensional equivalent of geometric consistency testing between images.

Sivic and Zisserman [1] combined the invariance of local features [16] with conventional retrieval techniques applied to vector-quantized SIFT [17] descriptors. The quantization speed and vocabulary size has been significantly improved by using hierarchical k-means [5] or faster search strategies, such as a kd-tree over the k-means trained clusters [6]. Similarly, the introduction of soft assignment [6, 4] has increased matching power, at the expense of additional processing.

Several authors have tried to enforce geometric consistency of the matches using multiple bins during voting [2, 18, 19]. However, the types of geometric constraints which can be enforced this way are relatively limited compared to the exhaustive geometric verification of DAAT.

Jegou et al [2] constructed a local Hamming embedding for each posting list to find the best matches within the posting list at query time. It improves quality and reduces CPU load by scoring fewer images, at the cost of increased memory usage. This approach can be combined with DAAT scoring, and will lower the index density without the drawbacks of a larger vocabulary.

2.1 TAAT scoring variants

TAAT scoring is usually implemented as score accumulation into a vector or a map. It can be implemented using pure counts or by using weights and multiple bins.
Hit counting: The simplest possible TAAT scoring scheme is hit counting. We will use this scoring variant in the timing experiments on synthetic data.

In order to implement this, we invert the document database to a set of posting lists. At scoring time we then go over the posting lists associated with the features in the query and for each document mentioned in a posting list, we increment the corresponding entry in the accumulator.
TF-IDF Scoring: One way of how to do TF-IDF (Term-Frequency/Inverse Document Frequency) scoring for images is laid out in Section 4 of [5]. Basically, the addition from simple hit counting is that each posting list has a weight and each document a normalizer.

In order to use TF-IDF the document weights have to be kept in memory or be attached to the posting lists. For very large indexes, both options are inconvenient, either due to memory used or a significant increase in posting list sizes.

3 Approach

In this section, we present our approach to large-scale image retrieval.
Feature detection and quantization: We believe that the system described will work with any quantized local features framework. Our system considers local maxima of Laplacian of Gaussian as features [20]. It then describes each feature using a descriptor similar to the one presented in [21], where we have removed rotational invariance.

Given a training set of such descriptors, a vocabulary of 5 million vectors for LoG maximums and 5 million vectors for LoG minimums is trained using k-means. At runtime, quantization is achieved using a kd-tree that accelerates the search for the appropriate quantization vector [6].
Indexing: Our indexing process produces an inverted index that allows fast retrieval. First features are extracted and quantized from the database images, yielding sets of quantization value, position and scale. The index is then grouped

by quantization value. As a result, each quantization value is linked to a set of entries. Each entry has a document identifier, a normalized image position, and a normalized scale. Within each posting list, entries are sorted by document identifier.

3.1 Retrieval and document at a time scoring

Given a query image, our system extracts and quantizes features. Then, it fetches the posting lists corresponding to these features. The counting min-tree presented in the next paragraph merges the posting lists. It scans the posting lists and groups all matches for each document, one document a time. If an image has fewer than four hits, it is discarded immediately. Otherwise, the geometric consistency of matches is verified, yielding the final score for this image. Then, the counting min-tree provides matches for the next document, and scoring continues.

The system finally sorts results based on their score and returns the best matches.

Our geometric verifier computes a four parameter transformation consisting of stretch and translation along the x- and y-axes, using RANSAC [22] to eliminate outliers, the minimal set for computing such a transformation is two points. The verification uses several rejection criteria: The location and scale of features must match the computed transformation and the transformation must be reasonable. All tests are applied in both transformation directions. Since feature location and scale are embedded in the posting lists, this geometric verification does not require any additional data to be fetched or features to be matched. This makes it very fast, which allows for exhaustive geometric verification.

3.2 Counting Min-Tree

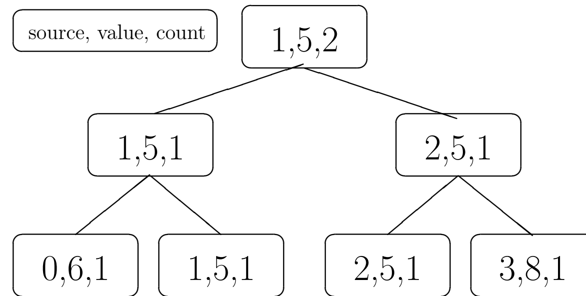

Fig. 4. The counting min-tree maintains a state where each node remembers where its information came from, the value and the count. Every time we remove the element from the root, we step the posting list associated with the current source and propagate the information from the leaf all the way up to the root.

The task of the counting min-tree is to merge the posting lists triggered by the query image. In other words, it groups the information from the posting lists document. The task is made easier by the fact that the posting lists are already sorted by document identifier.

Merging k sorted lists, containing a total of n elements, can be done in $O(n \log(k))$ time by using a binary min-heap. The heap keeps track of which list to take the next element from, as in the merge phase of a merge sort [23].

However, updating a heap for each element is costly in terms of CPU power. Instead, we propose a novel structure called a *counting min-tree*, based on the existing but somewhat lesser-known min-tree structure, sometimes also called a winner tree.

The min-tree, like the min-heap, is realized using a binary tree, but differs from the heap in that it only stores elements in the leaf nodes, of which there is a fixed number—one for each posting list triggered by the query. For simplicity, we use a complete binary tree, so that the number of nodes is always a power of two, with the extra nodes always having a special highest "end-of-file" document identifier.

Each inner node contains the smallest document identifier of its two children, as well as an index into which leaf this document identifier comes from. Therefore, the root element identifies both the least-advanced posting list and its document identifier, allowing for immediate identification of which posting list to step.

We further augment the min-tree into a counting min-tree by including a count in each inner node. The count contains how many leaf nodes below it has the given document identifier; for instance, if an inner node has two leaves below it with document identifiers 100 and 200, its document identifier will be 100 and the count 1, but if the leaves' document identifiers are 100 and 100, the inner node will have a document identifier of 100 and a count of 2.

The count is, like the document identifier, propagated up the tree all the way to the root. The root will then, in turn, not only contain the document identifier of the least-advanced posting lists and its index, but also how many other posting lists point to the same document identifier. Identifying candidates for scoring then simply amounts to looking at the root node; if its count is at least four, we have a candidate for scoring, and if not, we can continue stepping the least-advanced posting list.

The counting min-tree has two main advantages over a heap. First, it is significantly faster, and already at the first hit on a document we know how many hits the document will have. Second, knowing the number of hits without having to advance over all the hits simplifies the logic of collecting geometry data from the posting lists.

4 Scaling Analysis

In this section, we first define a density index. We then study the behavior of several algorithms when this density changes. To do so, we use a synthetic

benchmark. We show that, under conditions necessary for large-scale systems, our counting min-tree outperforms other solutions.

4.1 Index Density

In inverted index-based systems, the retrieval computation burden is proportional to the number of posting list entries the system has to process for a given query.

In order to estimate the number of elements scanned during a single query, let N be the number of documents in the database, n_q the number of features in the query image, n_d the average number of features per indexed document, and V the vocabulary size. For simplicity, and without loss of generality, we use in the experiments $n_q = n_d$. We denote the index imbalance factor [24, 2]

$$u = V \sum_{i=1}^{V} p_i^2 \ , \tag{1}$$

where p_i is the fraction of features quantized to word i. Using Chebyshev's sum inequality, we see that u has a lower bound of 1, reached when $p_i = 1/V$.

We define the index density

$$D = \frac{u n_q n_d}{V} \ . \tag{2}$$

The number of elements the system has to process for a query is

$$M = N \frac{u n_q n_d}{V} = ND \ . \tag{3}$$

On average, $M = ND$ is the number of matches between query and indexed features.

The value D can be used to categorize systems as sparse or dense. A high density factor improves retrieval at the cost of a heavier computation and storage burden. A sparse index leads to a lighter system with a potentially lower recall.

The time taken by the retrieval stage is M/r, where r is the number of entries the system processes per second.

In the following, we measure the process rate r of several algorithms, when processing databases of different sizes (N) and densities (D).

4.2 Synthetic Experiment

In order to compare the raw speed of several different scoring strategies, we define a synthetic experiment with a database of N synthesized documents, each with n random features from a vocabulary of size V. The query is a synthesized document with n features drawn from the same uniform distribution. The scoring task is to retrieve from the database all the documents with at least four hits.

We evaluated two DAAT implementations and two TAAT ones:

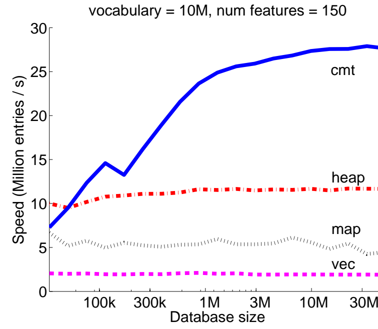

Fig. 5. On synthetic data and a sparse setup with a large vocabulary ($10M$) and a small feature count (150), the advantage of using the counting min tree (CMT) is growing with increased database size.

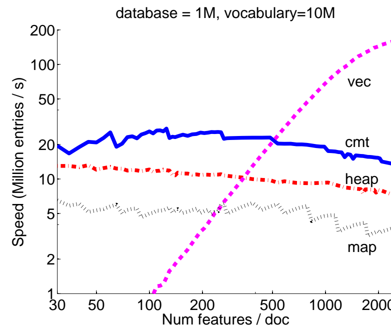

Fig. 6. On synthetic data, the counting min-tree (CMT) is the fastest option for pure hit-counting as long as the problem remains sparse. Above a certain density voting into a vector becomes faster.

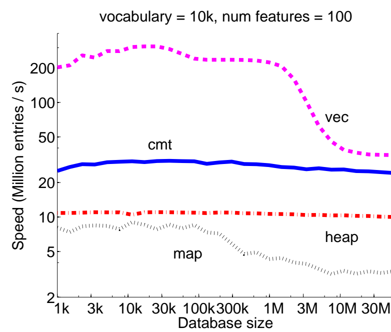

Fig. 7. On synthetic data and very dense setup with a very small vocabulary (10k) and a small feature count (100), the vector based hit counter has a massive advantage in terms of raw speed.

CMT DAAT using a counting min-tree.
heap DAAT implemented using a heap.
map TAAT with a sparse vote accumulator, implemented using a hash table [1].
vec TAAT with a dense vote accumulator, implemented using an array of 8-bit integers[2]

For each algorithm, we measure the number r of posting list entries processed per second in dense and sparse scenarios. For both TAAT algorithms, r includes the time required to vote and the time to scan the accumulator for documents with at least four hits. The DAAT algorithms achieve the same task in a single pass.

We compiled our benchmark with GCC 4.4.3, and ran it in 64-bit mode on one core of an Intel E6600 2.4GHz CPU.

The goal of our first experiment is to evaluate how the algorithms behave when database density changes. In this scenario, the database size is $N = 1$ million images. The vocabulary contains 10 million words, as for the real data experiment of Section 5. The number of features per image is equal for both query and reference images: $n_d = n_q = n$. We let n vary from 30 to 2000. By doing so, we change the index density.

Our second experiment concentrates on the sparse case. The vocabulary has 10 million words. The number of features is $n = 150$. This configuration matches the one of our real data experiment.

The third scenario considers a high density case: the vocabulary has 10k words. The number of features per image is set to $n = 100$. This configuration is very costly for large-scale databases, due to the number of entries processed. However, the tested algorithms, exhibit interesting behaviors in these conditions, which are presented next.

4.3 Synthetic Results

The advantage of DAAT scoring over TAAT scoring depends on the sparsity of the problem.

Figure 6 shows how algorithms behaves when the database density changes. When the index is dense, TAAT scoring with a dense accumulator outperforms other methods. When the density is low, as required for large scale databases, the counting min-tree is faster. The counting min-tree is almost always faster than heap-based DAAT which is always faster than TAAT implemented with a hash table. The counting min-tree requires a small, constant time initialization. That explains why its performance degrade compared to other methods when processing extremely short posting lists. The results of this experiment clearly show that one should consider the importance of database density when designing a vocabulary-based image retrieval system.

[1] We use an `unordered_map<uint32, int>`, which is new for C++11. We found that it outperforms `map<uint32, int>` and `hash_map<uint32, int>` in all cases.
[2] For small databases 32-bit integers are slightly faster, but they scale less well.

Figure 5 depicts the sparse database scenario. It shows that the dense accumulator for TAAT is the slowest option. This is because reading out the results and resetting the vector become computationally heavy $O(N)$ compared to traversing the posting lists $O(Nn^2/V)$. A sparse accumulator is a significantly faster option, but it is remains slower than DAAT approaches. The rate at which both TAAT implementations process hits remain roughly constant with respect to the database size. In this sparse database scenario, DAAT scoring outperforms vector based TAAT scoring, since they avoid the $O(N)$ step of finding all elements with a sufficient score in a vector.

Figure 7 shows results of the dense index experiment. In these conditions, TAAT with a dense accumulator outperforms other methods since the $O(N)$ cost of finding the matches in the accumulator is majored by the $O(Nn^2/V)$ cost for processing the posting lists.

By dividing a large problem into smaller sub-problems it is possible to take the highest speed of each method and linearize to larger problems. This also makes it possible to keep the memory requirements of the vector and map based scorers at a reasonable level.

5 Proof of Concept on Real Data

In order to show that this can run at scale we used the method described above build a low invariance index of 94 billion images obtained as part of a large web-crawl. This means that many of the images are small and that there is a high number of near-duplicates in the set, however, the set is still amazingly rich and provides an extraordinary coverage of what is on the internet.

We extracted quantized features for 94 billion images. For each image we extracted (up to) 200 (average 135) non-rotational invariant features and quantized using 10M cells. This gives a total count of $1.3 \cdot 10^{13}$ features, with a feature distribution very close to random which was compressed to $63TB$ with a per feature cost of 5.4 bytes. This feature processing was run integrated with other feature processing on a very large cloud.

We wrote a single posting-list index over these features to a networked file system.

To query against the index, we set up a single machine to which we send the quantized features of the query image. The query is processed in three stages: In the first stage, the posting lists are fetched from the networked storage and decompressed. In the second stage, we advance the posting lists in DAAT-order using a counting min-tree, and documents with at least four hits are scored by geometric consistency. In the last stage, we sort the results by score and return the top 1000 results to the caller.

Unfortunately the time to fetch and decompress is larger than the actual time needed for scoring. For a typical 100 feature query we need to fetch and decompress 65M entries (that is 350MB) which takes just below 2 seconds. The scoring parallelizes very well and runs at around 12.5M entries per core-second on eight cores, for a scoring time of around 0.7 seconds.

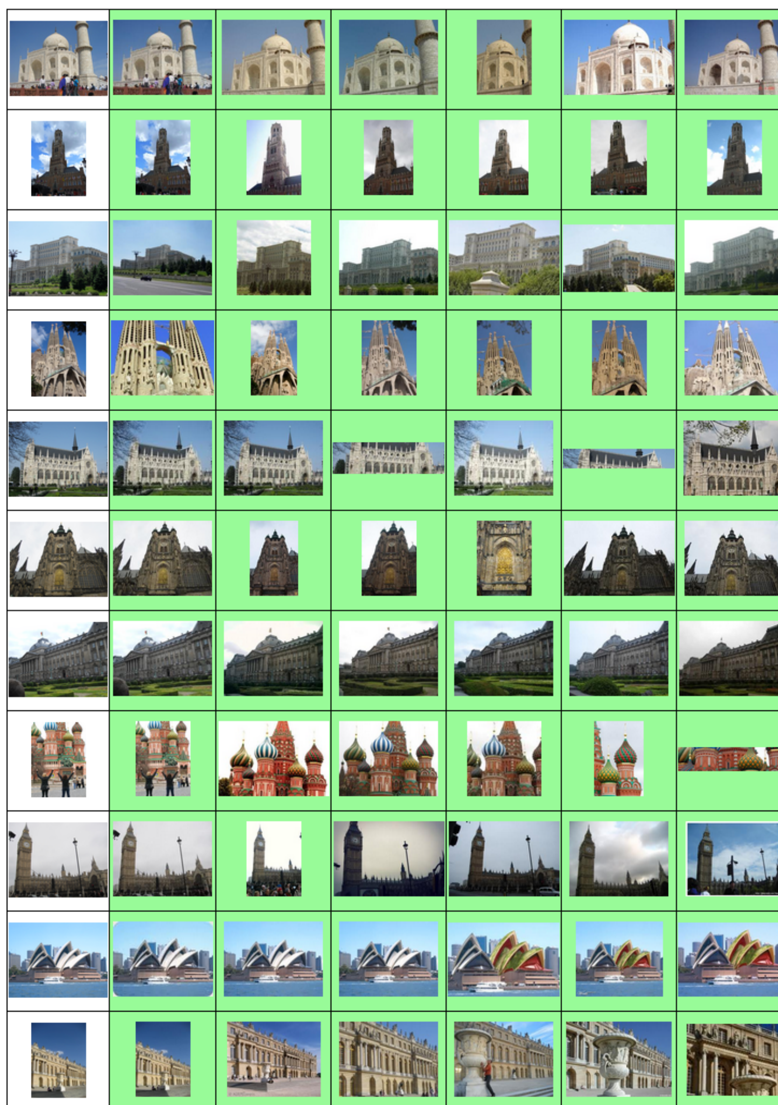

Fig. 8. The query is in the first column; the remaining columns are the first few results in score order. From top to bottom: The Taj Mahal, the Belfry of Bruges, the Palace of the Romanian Parliament, La Sagrada Familia, Notre Dame, St. Vitus Cathedral, the Royal Palace of Brussels, Saint Basil's Cathedral, Big Ben, the Sydney Opera House, and the Versailles. Matching famous landmarks is rather easy since the database contains a very large number of images of these so recall does not have to be perfect.

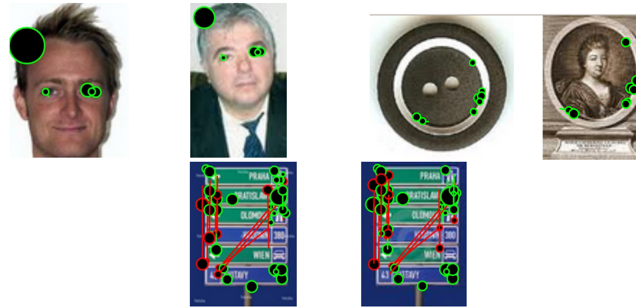

Fig. 9. Common classes of mismatches. Faces matches other faces, round object match round objects and text matches text.

The difference in scoring speed from the synthetic experiment mainly comes from the fact that in the synthetic experiment the posting lists are zero terminated 32 bit integer arrays while we here deal with more complicated structures which handle 64 bit docids and also hold payloads. Another source of slowdown is that the scoring in the synthetic experiment was simple hit-count scoring while we here do far more complicated scoring. However, since we are considerably much faster than the network, we have demonstrated that we can keep up with the data feed. A system running at 30M entries per core-second would need 160MB/s input per core to stay fed, a rate which would be hard to sustain from any other source than RAM. Already at you need almost 70MB/s per core.

Sharding the system will change the amount of data which needs to be processed per machine but will not change the balance between processing power and io-needs.

The experimental system has considerable amounts of RAM but only requires the memory needed to hold the posting lists relevant for the current query.

Quality Problems Driven By Scale: When running all-against-all on the UKY [5] database, a single false positive corresponds to a pairwise false positive rate of 10^{-8}. With a larger database, this rate corresponds to ten false positives per query per per billion images in the database.

Studying the mismatches on a very large image database is instructive, and most fall into one of four categories: faces, round objects, text and watermarks. In order to prepare Figures 9, we have increased the number of query features in order to make these cases clearer.

If You Have the Images, You Can Index Them: Even though the task of indexing billions of images might sound gigantic, it is worth pointing out that the task of creating an index of this type is in most cases much smaller than the task of collecting the images.

Given a feature detector, descriptor, and quantization pipeline running at 30 images per second, each CPU core can index about three million images per

day. The highly parallel nature of the computation and the dropping price of multi-core systems makes it affordable to scale to very large corpora.

6 Conclusions

We have proposed a DAAT approach to image retrieval, efficiently implemented using a counting min-tree. Our method verifies the geometry of every candidate in a single pass. Our proof of concept setup is able to query a database of more than 90 billion images in a few seconds.

By studying the speed of multiple algorithms as a function of the database density, we have shown that our approach efficiently handles the extreme scoring sparsity required to query very large databases, at an acceptable cost.

It was our plan to show a great win from geometric scoring of all possible hits, however, for the hit-density which we have in our proof of concept system there are very few "random" documents with at least four hits and it would have been affordable to fetch these and verify. We suspect that there would be a clear win for a system operating in a denser setting.

It is our hope that this work will have demonstrated that if you can acquire an image collection, then the step to indexing it is not very far.

References

1. Sivic, J., Zisserman, A.: Video Google: A text retrieval approach to object matching in videos. ICCV. (2003)
2. Jégou, H., Douze, M., Schmid, C.: Hamming embedding and weak geometric consistency for large scale image search. ECCV. (2008)
3. Jegou, H., Schmid, C., Harzallah, H., Verbeek, J.: Accurate image search using the contextual dissimilarity measure. PAMI **32** (2009) 2–11
4. Schindler, G., Brown, M., Szeliski, R.: City-scale location recognition. CVPR. (2007)
5. Nistér, D., Stewénius, H.: Scalable recognition with a vocabulary tree. CVPR. (2006)
6. Philbin, J., Chum, O., Isard, M., Sivic, J., Zisserman, A.: Object retrieval with large vocabularies and fast spatial matching. CVPR. (2007)
7. Philbin, J., Isard, M., Sivic, J., Zisserman, A.: Descriptor learning for efficient retrieval. ECCV. (2010)
8. Inria, searching for similar images in a database of 10 million images - example queries. (http://bigimbaz.inrialpes.fr/)
9. Google Goggles. (http://www.google.com/mobile/goggles/)
10. Google. Search By Image. (http://www.google.com/insidesearch/searchbyimage.html)
11. Baidu. Search by Image. (http://stu.baidu.com/)
12. Sogou. Search by Image. (http://pic.sogou.com/)
13. TinEye blog. (http://blog.tineye.com/)
14. Turtle, H., Flood, J.: Query evaluation: strategies and optimizations. Information Processing & Management **31** (1995) 831–850
15. Kaszkiel, M., Zobel, J., Sacks-davis, R.: Efficient passage ranking for document databases. ACM Transactions on Information Systems **17** (1999) 406–439

16. Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide-baseline stereo from maximally stable extremal regions. Image and Vision Computing **22** (2004) 761–767
17. Lowe, D.: Distinctive image features from scale-invariant keypoints. IJCV **60** (2004) 91–110
18. Lin, Z., Brandt, J.: A Local Bag-of-Features Model for Large-Scale Object Retrieval. ECCV. (2010)
19. Zhang, Y., Jia, Z., Chen, T.: Image retrieval with geometry-preserving visual phrases. CVPR. (2011)
20. Mikolajczyk, K., Schmid, C.: An affine invariant interest point detector. ECCV. (2002)
21. Zheng, Y.T., Zhao, M., Song, Y., Adam, H., Buddemeier, U., Bissacco, A., Brucher, F., Chua, T.S., Neven, H.: Tour the world: Building a web-scale landmark recognition engine. CVPR. (2009)
22. Fischler, M., Bolles, R.: Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM **24** (1981) 381–395
23. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition. (1998)
24. Fraundorfer, F., Stewénius, H., Nistér, D.: A binning scheme for fast hard drive based image search. CVPR. (2007)