# Distributed Kd-Trees for Retrieval from Very Large Image Collections

Mohamed Aly[1]
malaa@vision.caltech.edu
Mario Munich[2]
mario@evolution.com
Pietro Perona[1]
perona@caltech.edu

[1] Computational Vision Group, Caltech
Pasadena, CA 91125 USA
[2] Evolution Robotics
Pasadena, CA 91106 USA

**Abstract**

Distributed Kd-Trees is a method for building image retrieval systems that can handle hundreds of millions of images. It is based on dividing the Kd-Tree into a "root subtree" that resides on a root machine, and several "leaf subtrees", each residing on a leaf machine. The root machine handles incoming queries and farms out feature matching to an appropriate small subset of the leaf machines. Our implementation employs the MapReduce architecture to efficiently build and distribute the Kd-Tree for millions of images. It can run on thousands of machines, and provides orders of magnitude more throughput than the state-of-the-art, with better recognition performance. We show experiments with up to 100 million images running on 2048 machines, with run time of a fraction of a second for each query image.

## 1 Introduction

Large scale image retrieval is an important problem with many applications. There are already commercial applications, like Google Goggles, that can recognize images of books, DVDs, landmarks among other things. A typical scenario is having a large database of book covers, where users can take a photo of a book with a cell phone and search the database which retrieves information about that book and web links for purchasing the book for example. The database should be able to handle millions or even billions of images of objects, imagine for example indexing all books and DVD covers (movies, music, games, ... etc.) in the world.

There are two major approaches for building such databases: Bag of Words (BoW) [1, 2, 8, 16, 17, 18] and Full Representation (FR) [2, 11]. In the first, each image is represented by a histogram of occurrences of quantized features, and search is usually done efficiently using an Inverted File (IF) structure [19]. The second works by getting the approximate nearest neighbors for the features in the query image by searching all the features of the database images using an efficient search method (e.g. Kd-Trees [11]). The first has the advantage of using an order of magnitude less storage, however its performance is far worse, see [2] for a benchmark. Hence, we focus on FR with Kd-Trees at its core.

© 2011. The copyright of this document resides with its authors.
It may be distributed unchanged freely in print or electronic forms.
BMVC 2011 http://dx.doi.org/10.5244/C.25.40

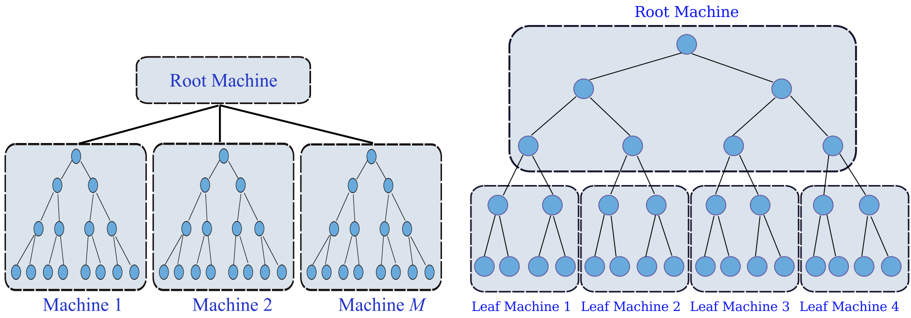


Figure 1: **Kd-Tree Parallelizations**. (*Left*) ***Independent Kd-Tree (IKdt)***. The image database is partitioned into $M$ equal-sized subsets, each of which is assigned to one of the $M$ machines that builds an independent Kd-Tree. At query time, all the machines search in parallel for the closest match. (*Right*) ***Distributed Kd-Tree (DKdt)***. A global Kd-Tree is built across $M+1$ machines. The *root* machine stores the top of the tree, while the *leaf* machines store the leaves of the tree. At query time, the root machine directs features to a subset of the leaf machines, which leads to higher throughput. See Sec. 3.

Most past research on large scale image search [1, 2, 5, 8, 9, 10, 14, 15, 16, 17] did not scale past a few million images on one machine because of RAM limits. However, if the goal is to scale up to billions of images, we have to think beyond one machine. Therefore, we focus on ways to distribute the image database on an arbitrary number of machines to be able to scale up the recognition problem to hundreds of millions of images. We implement and compare two approaches for Kd-Tree parallelizations using the MapReduce paradigm [7]: Independent Kd-Trees (IKdt), which is the baseline brute force approach, and Distributed Kd-Trees (DKdt) [2], see Figure 1. We run extensive experiments to assess the different system parameters and compare the two approaches with datasets of up to 100 million images on a computer cluster with over 2000 machines.

Our contributions are:

1. We provide practical implementations of two approaches to parallelize Kd-Trees. We build upon our previous work in [2], where we introduced the idea of distributed Kd-Tree parallelization. We provide the intricate details of backtracking within the DKdt as well as a novel way to build the tree, see Sec. 3.
2. We introduce the use of the MapReduce [7] paradigm in large scale image retrieval. Up to our knowledge, this is the first work using MapReduce in this application.
3. We go beyond any published work (up to our knowledge) with respect to the size of the database in an object recognition application. We run experiments up to 100 million images on over 2000 machines.
4. Our experiments show the superiority of DKdt which, for a database of 100M images, has over 30% more precision than IKdt and about 100 times more throughput, and processes a query image in a fraction of a second.

**Algorithm 1** Full Representation Image Search with Kd-Tree.

**Image Index Construction**

1. Extract local features $\mathbf{f}_{ij}$ from every database image $i$, where $j = 1 \ldots F_i$ and $F_i$ is the #features in image $i$.
2. Build a Kd-Tree $T(\mathbf{f}_{ij})$ from these image features.

**Image Index Search**

1. Given a probe image $q$, extract its local features $\mathbf{f}_{qj}$. For each feature $\mathbf{f}_{qj}$, search the Kd-Tree $T$ to find the nearest neighbors $n_{qjk}$.
2. Every nearest neighbor votes for one (or more) database image(s) $i$ that it is associated to.
3. Sort the database images based on their score $s_i$.
4. Post-process the sorted list of images to enforce some geometric consistency and obtain a final list of sorted images $s'_i$.

**Algorithm 2 Kd-Trees Construction and Search**

**Kd-Tree Construction**

**Input**: A set of vectors $\{x_i\} \in \mathbb{R}^N$

**Output**: A set of binary Kd-Trees $\{T_t\}$. Each internal node has a split $(dim, val)$ pair where $dim$ is the dimension to split on and $val$ is the threshold such that all points with $x_i[dim] \leq val$ belong to the left child and the rest belong to the right child. The leaf nodes have a list of indices to the features that ended up in that node.

**Operation**: For each tree $T_t$:

1. Assign all the points $\{x_i\}$ to the root node
2. For very *node* in the tree visited in Breadth-First order, compute the split as follows:
   (a) For each dimension $d = 1 \ldots N$, compute its mean $mean(d)$ and variance $var(d)$ from the points in that node
   (b) Choose a dimension $d_r$ at random from the variances within 80% of the maximum variance
   (c) Choose the split value as the median of that dimension
   (d) For all points that belong to this node: if $x[d_r] \leq mean(d_r)$ assign $x$ to $left[node]$ , otherwise assign $x$ to $right[node]$

**Kd-Tree Search**

**Input**: A set of Kd-Trees $\{T_t\}$, a set of vectors $\{x_i\} \in \mathbb{R}^N$ used to build the trees, a query vector $q \in \mathbb{R}^N$, maximum number of backtracking steps $B$

**Output**: A set of $k$ nearest neighbors $\{n_k\}$ with their distances $\{d_k\}$ to the query vector $q$.

**Operation**:

1. Initialize a priority queue $Q$ with the root nodes of the $t$ trees by adding $branch = (t, node, val)$ with $val = 0$. The queue is indexed by $val[branch]$ i.e it returns the branch with smallest $val$.
2. $count = 0$. $list = []$
3. While $count \leq B$
   (a) Retrieve the top *branch* from $Q$.
   (b) Descend the tree defined by *branch* till *leaf*, adding unexplored branches on the way to $Q$.
   (c) Add the points in *leaf* to *list*.
4. Find the $k$ nearest neighbors to $q$ in *list* and return the sorted list $\{n_k\}$ and their distances $\{d_k\}$.

# 2 Background

The basic Full Representation image search approach with Kd-Trees is outlined in Alg. 1, while the Kd-Tree construction and search are outlined in Alg. 2. We use the Best-Bin-First variant of Kd-Trees [3, 11] which utilizes a priority queue [6] to search through all the trees simultaneously.

MapReduce [7] is a software framework introduced by Google to support distributed processing of large data sets on large clusters of computers. The software model takes in a set of *input* key/value pairs and produces a set of *output* key/value pairs. The user needs to supply two functions: **(1)** ***Map***: takes an input pair and produces a set of *intermediate* key/value pairs. The library collects together all intermediate pairs with the same key $I$ and feeds these to the Reduce function. **(2)** ***Reduce***: takes an intermediate key $I$ and a set of values associated with it and "merges" these values together and outputs zero or more output pairs.

In addition, the user supplies the *specifications* for the MapReduce job to be run, for example specifying the required memory, disk, the number of machines, etc. The canonical example for MapReduce is counting how many times each word appears in a document, see Fig. 2. The *Map* function is called for every line of the input document, where the key might be the line number and the value is a string containing that line of text. For every

```
Map(String key, String value):
 for each word w in value:
  EmitIntermediate(w, "1");
```

```
Reduce(String key, Iterator values):
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

Figure 2: Canonical MapReduce Example. See Sec. 2.

word in the line, it emits an intermediate pair, with the word as the key and a value of "1". The MapReduce takes care of grouping all intermediate pairs with the same key (word), and presents them to the *Reduce* function, which just sums up the values for every key (word) and outputs a pair with the word as the key and number of times that word appeared as the value.

We use MapReduce because it has been proven successful and used in a lot of companies, most notably Google, Yahoo!, and Facebook. It is simple, reliable, and scalable [7]. The user provides two functions, and the infrastructure takes care of all intermediate processing, key sorting, inter-machine communications, machine failures, job restarts, etc. It can easily scale up to tens of thousands of machines. Furthermore, there is already a strong open source implementation, called Hadoop, that is widely used.

## 3 Distributed Kd-Trees

When the number of images exceed the memory capacity of one machine, we need to parallelize the Kd-Tree on a number of machines. We explore two ways to parallelize Kd-Trees, see Fig. 1:

1. **Independent Kd-Trees (IKdt)**: The simplest way of parallelization is to divide the image database into subsets, where each subset can fit in the memory of one machine. Then each machine builds an independent Kdt for its subset of images. A single root machine accepts the query image, and passes the query features to all the machines, which then query their own Kdt. The root machine then collects the results, counts the candidate matches, and outputs the final sorted list of images.
2. **Distributed Kd-Trees (DKdt)**: Build a single Kdt for the entire database, where the top of the tree resides on a single machine, the root machine. The bottom part of the tree is divided among a number of leaf machines, which also store the features that end up in leaves in these parts. At query time, the root machine directs the search to a subset of the leaf machines depending on where features exit the tree on the root machine. The leaf machines compute the nearest neighbors within their subtree and send them back to the root machine, which performs the counting and outputs the final sorted list of images.

The most obvious advantage of DKdt is that a single feature will only go to a small subset of the leaf machines, and thus the ensemble of leaf machines may search simultaneously for the matches of multiple features at the same time. This is justified by the fact that most of the computations is performed in the leaf machines [2]. The root machine might become a bottleneck when the number of leaf machines increases, and this can be resolved by having multiple copies of the root, see Sec. 5 and Fig. 8. The two main challenges with DKdt are: (a) How to build the Kdt that contains billions of features since it does not fit on one machine? (b) How to perform backtracking in this distributed Kdt?

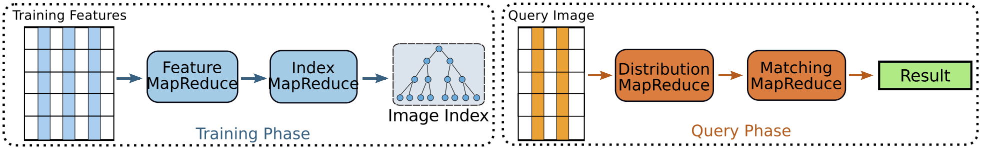


Figure 3: **Parallel Kd-Tree Schematic**. In the *training phase*, the Training MapReduce distributes the features among the different machines, and builds the different Kdts. In the *query phase*, the query image is first routed through the Distribution MapReduce, which routes the query features into the appropriate machines, whose results are then picked up by the Matching MapReduce, that queries the respective Kdts and outputs the results. See Sec. 3.

We solve these two problems by noticing the properties of Kd-Trees: (a) We do not build the Kdt on one machine, we rather build a feature "distributor", that represents the top part of the tree, on the root machine. Since can not fit all the features in the database in one machine, we simply subsample the features and use as many as the memory of one machine can take. This does not affect the performance of the resulting Kdt since computing the means in the Kdt construction algorithm subsamples the points anyway. (b) We only perform backtracking in the leaf machines, and not in the root. To decide which leaf machines to go to, we test the distance to the split value, and if it is below some threshold $S_t$, we include the corresponding leaf machine in the process.

**Algorithm 3 Parallel Kd-Trees**

**Independent Kd-Tree (IKdt)**

```
Feature Map(key, val)
// nothing
Feature Reduce(key, vals)
// nothing
Index Map(key, val)
Emit(val.imageid mod M, val.feat);
Index Reduce(key, vals)
index = BuildIndex(vals);
Emit(key, index);
Distribution Map(key, val)
for (i = 0; i < M; ++i)
 Emit(i, val);
Distribution Reduce(key, vals)
// nothing
Matching Map(key, val)
nn = SearchNN(val.feat);
Emit(val.imageid, nn);
Matching Reduce(key, vals)
matches = Match(vals);
Emit(key, matches);
```

**Distributed Kd-Tree (DKdt)**

```
Feature Map(key, val)
Emit(val.id mod skip, val.feat);
Feature Reduce(key, vals)
top = BuildTree(vals);
Emit(key, top);
Index Map(key, val)
indexId = SearchTop(val.feat);
Emit(indexId, val.feat);
Index Reduce(key, vals)
index = BuildIndex(vals);
Emit(key, index);
Distribution Map(key, val)
indexIds = SearchTop(val.feat, S_t);
for id in indexIds:
 Emit(id, val.feat);
Distribution Reduce(key, vals)
// nothing
Matching Map(key, val)
nn = SearchNN(val.feat);
Emit(val.image id, N);
Matching Reduce(key, vals)
matches = Match(vals);
Emit(key, matches);
```

The MapReduce architecture for implementing both IKdt and DKdt is shown in Fig. 3. It proceeds in two phases:

1. **Training Phase**: the Training MapReduce directs the training features into the different machines and then each machine builds the Kdts with the features assigned to it.
2. **Query Phase**: the Distribution MapReduce directs the query features into the appropriate machines

The implementation of IKdt with MapReduce is straightforward, see Alg. 3. At training time, the Feature MapReduce is empty, while the Index MapReduce builds the independent Kd-Trees from groups of images, where the Map distributes features according to the image id, and the Reduce builds the Kdt with the features assigned to every machine. At query time, the Distribution MapReduce dispatches the features to all the $M$ Kdts (machines). The Matching MapReduce searches the Kdts on each machine in the Map and performs the counting and sorting in the Reduce.

The implementation of DKdt is outlined in Alg. 3. The notable difference from IKdt is the Feature MapReduce, which builds the top of the Kd-Tree. Given $M$ machines, the top part of the Kdt should have $\lceil \log_2 M \rceil$ levels, so that it has at least $M$ leaves. The Feature Map subsamples the input features by emitting one out of every input *skip* features, and the Feature Reduce builds the Kdt with those features. The Index MapReduce builds the $M$ bottom parts of the tree, where the Index Map directs the database features to the Kdt that will own it, which is the first leaf of the top part that the feature reaches with depth first search. The Index Reduce then builds the respective leaf Kdts with the features it owns. At query time, the Distribution MapReduce dispatches the query features to zero or more leaf machines, depending on whether the distance to the split value is below the threshold $S_t$. The Matching MapReduce then performs the search in the leaf Kdts and the counting and sorting of images, as in IKdt.

# 4 Experimental Setup

We use the same experimental setup from Aly *et al.* [2]. In particular, for the query set, we use the *Pasadena Buildings* dataset, and for the distractor set we use *Flickr Buildings*.Since that distractor set goes only up to 1M images, we downloaded from the Internet a set of ~100 million images searching for landmarks. So in total we have over 100M images in the database, with 625 query images (see [2]). The first 1M images have on average 1800 features each, while the rest of the 100M images have 500 features on average. The total number of features for all the images is ~46 billion. We report the performance as precision@$k$ i.e. the percentage of the queries that had the ground truth matching image in the top $k$ returned images. Specifically, precision@$k = \frac{\sum_q \{r_q \leq k\}}{\#queries}$ where $r_q$ is the resulting rank of the ground truth image for query image $q$ and $\{x\} = 1$ if $\{x\}$ is *true*. We wish to emphasize at this point that the Pasadena Buildings dataset is very challenging and that no method scores even near 100% correct on this dataset even when tested within a small database of $10^3 - 10^4$ images [2].

For both IKdt and DKdt, we fix the budget for doing backtracking for every feature, and this is shared among all the Kd-Trees searched for that feature. So for example, in the case of DKdt with a budget of 30K backtracking steps, if a feature goes to two leaf machines, each will use $B$ =15K, while if ten machines are accessed each will use $B$ =3K. For IKdt with

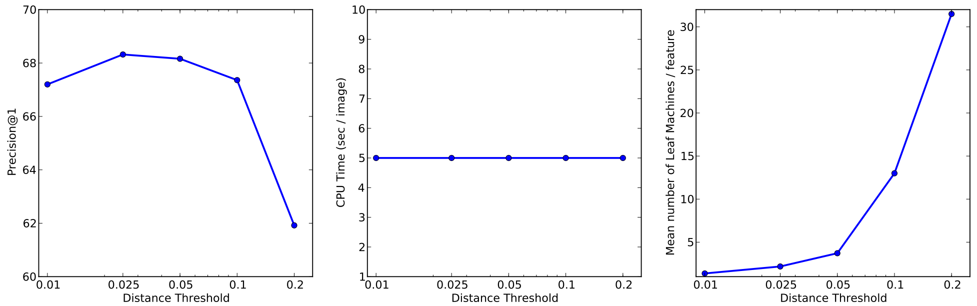


Figure 4: **Effect of Distance Threshold** $S_t$. The X-axis depicts the distance threshold $S_t$ which controls how many leaf machines are queried in DKdt (see Sec. 3). The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Mean number of leaf machines accessed per feature (*right*), using 1M images. See Sec. 5.

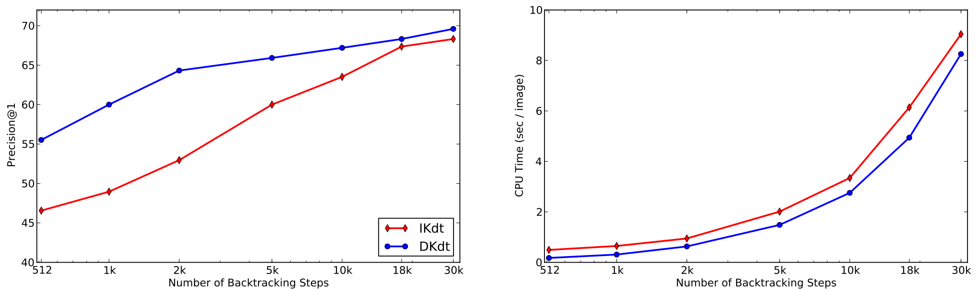


Figure 5: **Effect of Backtracking Steps** $B$. The X-axis depicts the number of backtracking steps $B$ which controls how deep the Kd-Trees are searched, and consequently the CPU time it takes (see Sec. 3). The Y-axis depicts precision@1 (*left*), and total CPU time (*right*), using 1M images. $S_t$ was set to 0.025 for DKdt. See Sec. 5.

$M$ machines, each machine will get $B/M$ backtracking steps. This allows a fair comparison between IKdt and DKdt by allocating the same number CPU cycles used in searching the Kd-Trees in both of them. The CPU time measurements in Sec. 5 count the matching time excluding the time for feature generation for the query images.

We use SIFT [11] feature descriptors (128 dimensions) with hessian affine [12] feature detectors. We used the binary available from tinyurl.com/vgg123. We implemented the two algorithms using Google's proprietary MapReduce infrastructure, however it can be easily implemented using the open-source Hadoop software. The number of machines ranged from 8 (for 100K images) up to 2048 (for 100M images). The memory per machine was limited to 8GB.

# 5 Experimental Results

We first explore, using 1M distractor images from *Flick Buildings*, the different parameters that affect the performance of the distributed Kd-Trees: distance threshold $S_t$, the number of backtracking steps $B$, and the number of machines $M$. Fig. 4 shows the effect of using

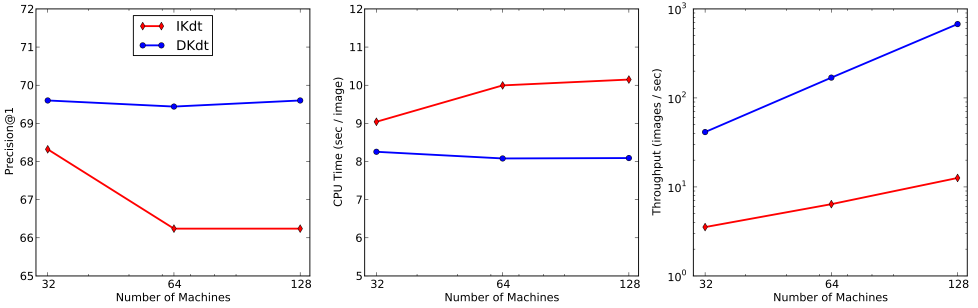


Figure 6: **Effect of Number of Machines** $M$. The X-axis depicts the number of machines used to build the system $M$. The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Throughput (*right*), using 1M images. See Sec. 5.

different values for the distance threshold $S_t$, which affects how many leaf machines are searched at query time when using DKdt, see Sec. 3. The CPU time counts the sum of the computational cost on all the machines, and stays almost constant with increasing the number of machines since we have a fixed budget for backtracking steps $B$ (see Sec. 4). The best tradeoff is with $S_t = 0.025$, which gives ~3 leaf machines per feature, while using a bigger $S_t$ means more leaf machines are queried and each will not be explored deep enough.

Fig. 5 shows the effect of the number of the total backtracking steps $B$. DKdt is clearly better than IKdt for the same $B$, since it explores less Kdts but goes deeper into them, unlike IKdt which explores all Kdts but with lower $B$. Fig. 6 shows the effect of the number of machines $M$ used to build the system. For DKdt, this defines the number of levels in the top of the tree, which is trained in the Feature MapReduce, Sec. 3. For IKdt, this defines the number of groups the images are divided into. For the same number of machines, DKdt is clearly superior in terms of precision, CPU time, and throughput. In particular, with increasing the number of machines, the CPU time of DKdt stays almost constant while that of IKdt grows, because despite $B$ is distributed over the all the machines, the features still need to be copied and sent to all the machines, and this memory copy consumes a lot of CPU cycles, see Fig. 7. We also note that the throughput increases with the number of machines, and that DKdt has almost 100 times that of IKdt.

Fig. 7 shows the effect of the number of images indexed in the database. We used 8 machines for 100K images, 32 for 1M images, 256 for 10M images, and 2048 for 100M images. DKdt clearly provides superior precision to IKdt, with lower CPU cost and much higher throughput. For 100M images, DKdt has precision about 32% higher than IKdt (53% vs 40%), with throughput that's about 30 times that of IKdt (~ 12 images/sec vs. ~ 0.4) i.e. processes images in a fraction of a second. Fig. 8(*left*) shows another view of the precision vs. the throughput. It is clear that by increasing the number of images, the precision goes down. Paradoxically, the throughput goes up with larger databases, and this is because we use more machines, and in the case of DKdt, this allows more interleaving of computation among the leaf machines and thus more images processed per second.

We note a drop in the throughput after some point with adding more machines, this is because the computations can not all be parallelized. The *root* machine, accepts the query image, computes the features, and dispatches them to the *leaf* machines that hold the Kd-Trees. It then gets back the results and performs the final scoring. While the Kdt search is

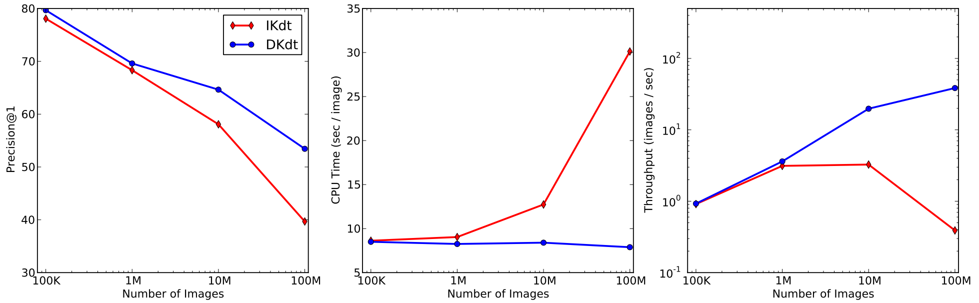


Figure 7: **Effect of Number Images**. The X-axis depicts the number of images in the database. The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Throughput (*right*). See Sec. 5.

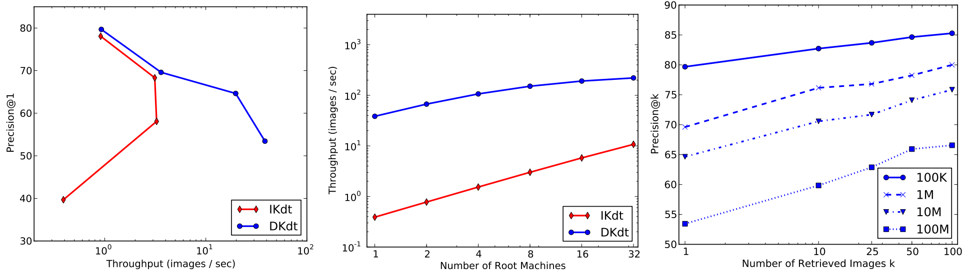


Figure 8: ***(Left)*** Precision@1 Vs Throughput. Every point represents one database size, from 100K up to 100M, going from the top left to the bottom right. ***(Center)*** Throughput Vs the Number of Root Machines for 100M images. ***(Right)*** Precision@$k$ for different values of $k$ for DKdt. See Sec. 5.

parallelized, the other computations are not, and by adding more leaf machines, the bottleneck of the root machines starts decreasing the throughput. Fig. 8(*center*) shows how the throughput increases when adding replicas of the root machine (using 100M images), which provides multiple entry points for the system and allows more images to be processed at the same time. The throughput grows with the number of root machines added, for example growing to ~ 200 images/sec for DKdt with 32 machines vs. ~ 10 images/sec with 1 root machine.

Finally, the precision@1 for 100M images for DKdt might seem disappointing, standing at about 53%. However, users usually do not just look at the top image, they might also examine the top 25 results, which might constitute the first page of image results. Fig. 8(*right*) shows the precision@$k$ for different values of $k$, ranging from 1 to 100. For 100M images, the precision jumps from 53% @1 to 63% @25 and up to 67% @100 retrieved images. This is remarkable, considering that we are looking for 1 image out of 100M images i.e. the probability of hitting the correct image by chance is $10^{-8}$. One more thing to note is that all the experiments were run with one Kd-Tree, and we anticipate that using more trees will give higher precision values at the expense of more storage per tree, see [2, 13].

# 6 Conclusions

In this paper, we explored parallel Kd-Trees for ultra large scale image retrieval. We presented implementations of two ways, Independent Kd-Tree and Distributed Kd-Tree, to parallelize Kd-Trees using the MapReduce architecture. We compared the two methods and ran experiments on databases with up to 100M images. We showed the superiority of DKdt which, for 100M images, has over 30% more precision than IKdt and at the same time over 30 times more throughput, and processes a query image in a fraction of a second. We find that the overall retrieval performance on what is believed to be a challenging dataset holds up well for very large image collections, although there is some space for improvement.

# Acknowledgements

This work was supported by ONR grant #N00173-09-C-4005 and was implemented during an internship at Google Inc. The implementation of distributed Kd-Trees is pending a US patent GP-2478-00-US [4]. We would like to thank Ulrich Buddemeier and Alessandro Bissacco for allowing us to use their implementation. We would also like to thank James Philbin, Hartwig Adam, and Hartmut Neven for their valuable help.

# References

[1] Mohamed Aly, Mario Munich, and Pietro Perona. Bag of words for large scale object recognition: Properties and benchmark. In *International Conference on Computer Vision Theory and Applications (VISAPP)*, March 2011.

[2] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *WACV*, 2011.

[3] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[4] Ulrich Buddemeier and Alessandro Bissaccoo. Distributed kd-tree for efficient approximate nearest neighbor search, 2009.

[5] O. Chum, J. Philbin, M. Isard, and A. Zisserman. Scalable near identical image and shot detection. In *CIVR*, pages 549–556, 2007.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[8] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV*, 2008.

[9] H. Jégou, M. Douze, and C. Schmid. Packing bag-of-features. In *ICCV*, sep 2009.

[10] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *CVPR*, 2010.

[11] David Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[12] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *IJCV*, 2004.

[13] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, 2009.

[14] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. *CVPR*, 2006.

[15] F. Perronnin, Y. Liu, J. Sánchez, and H. Poirier. Large-scale image retrieval with compressed fisher vectors. In *CVPR*, 2010.

[16] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. *CVPR*, 2007.

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

[18] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[19] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 2006. ISSN 0360-0300.