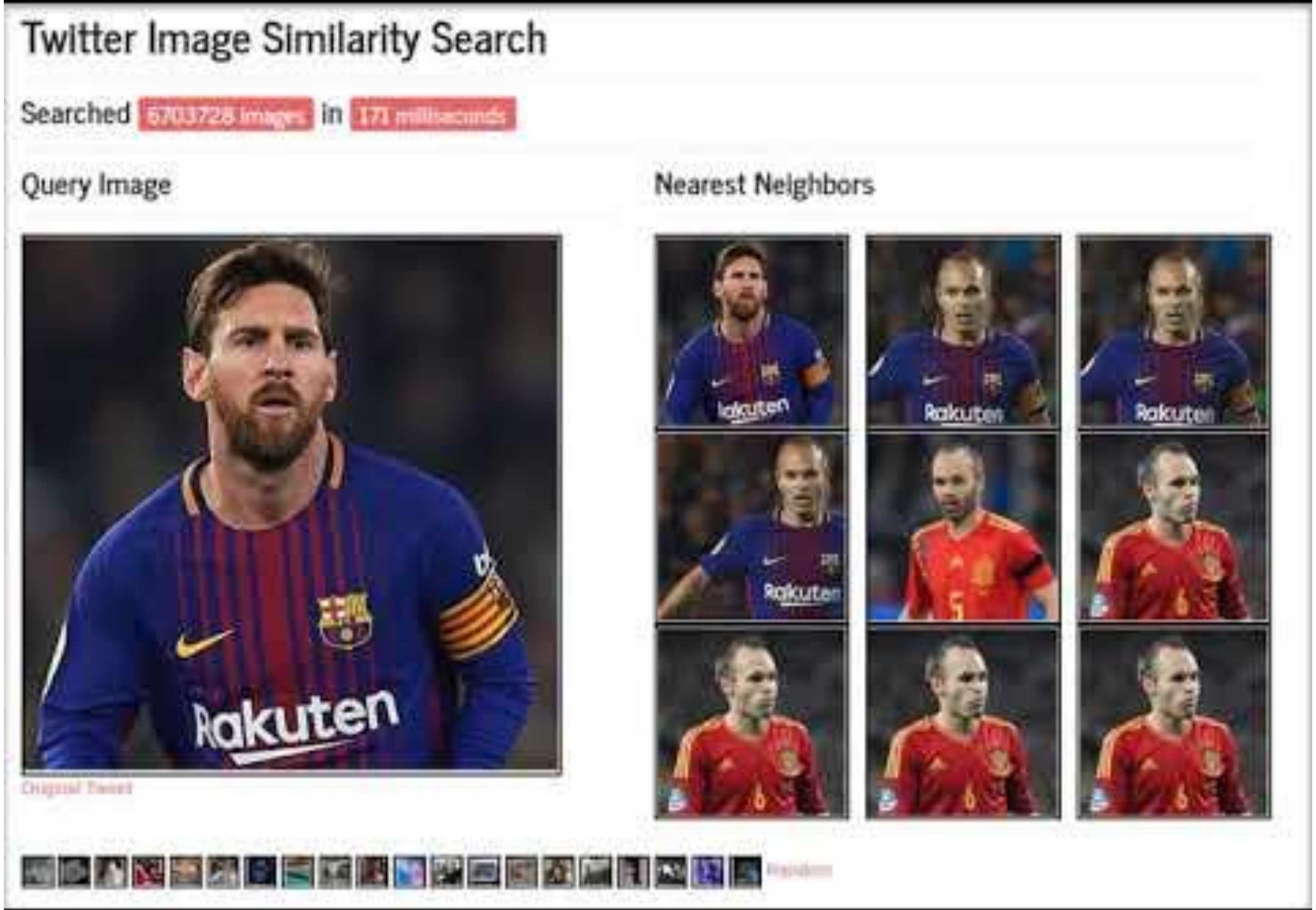
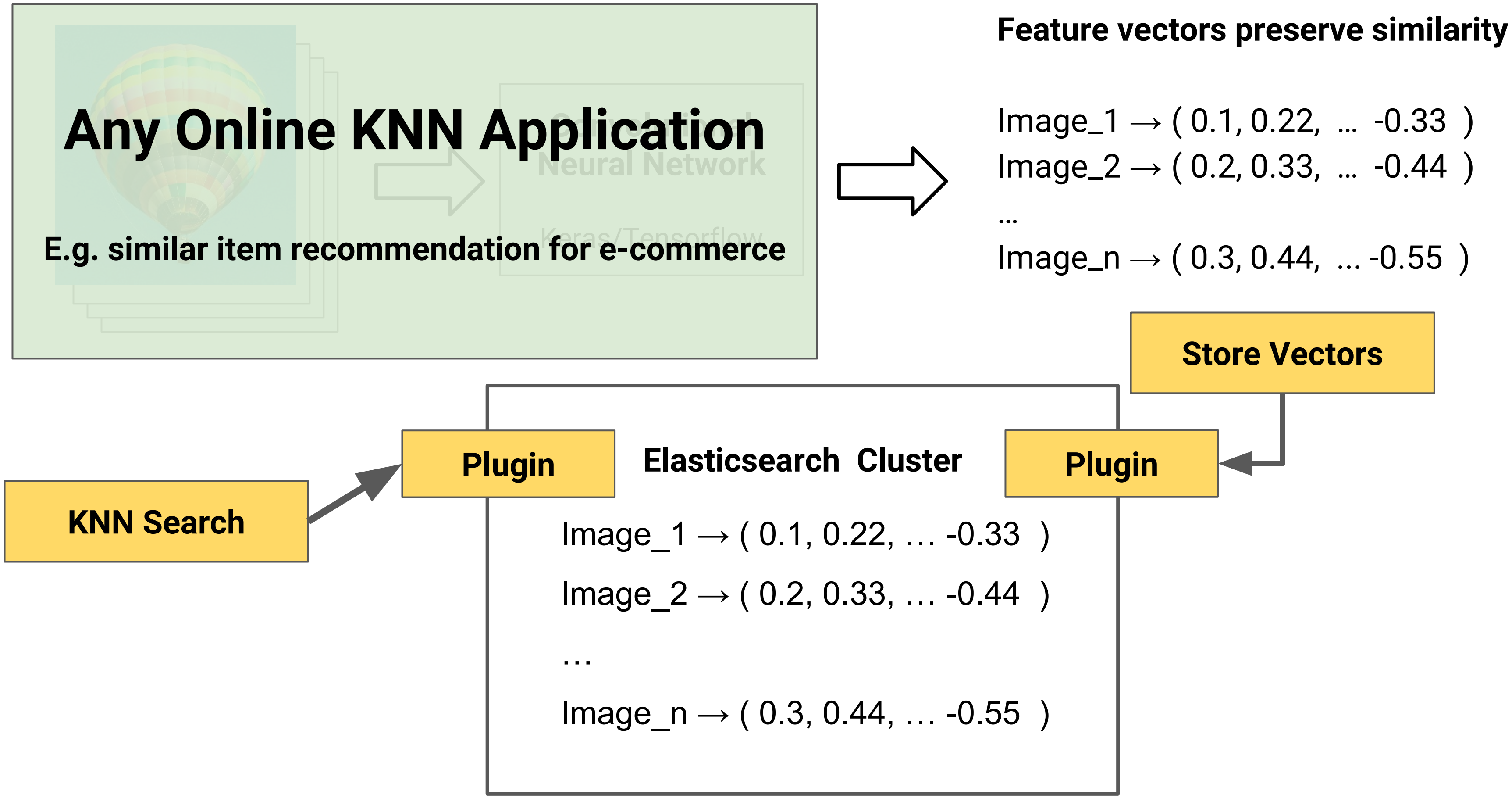
ElastiK Nearest Neighbors

An Elasticsearch Plugin to Simplify Online K-Nearest-Neighbors Search

Alex Klibisz - Insight Data Engineering - Boston


Twitter Image Similarity Search
Searched 6703728 images in 171 milliseconds
Query Image
Nearest Neighbors

Application: Image Similarity Search Engine



KNN on Elasticsearch - Why is it useful?

KNN in **offline** setting → standard batch infrastructure:

- Static corpus of vectors
- Batch job computes and caches neighbors

KNN in **online** setting → complicated infrastructure:

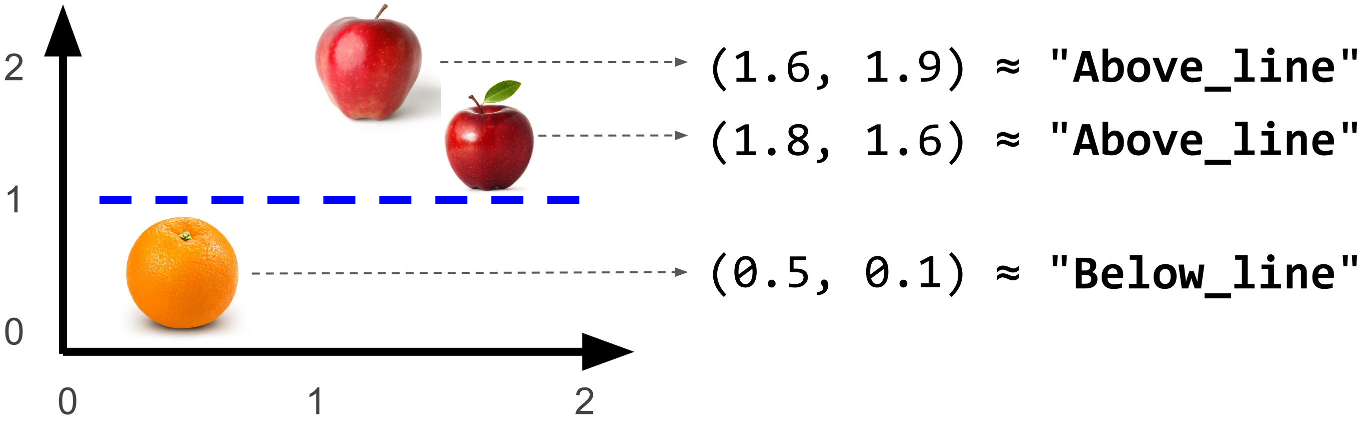
- Store and randomly access millions of vectors
- Constantly add more items to corpus
- Scale horizontally for many concurrent searches

Put the nearest neighbors search on a proven infrastructure: Elasticsearch

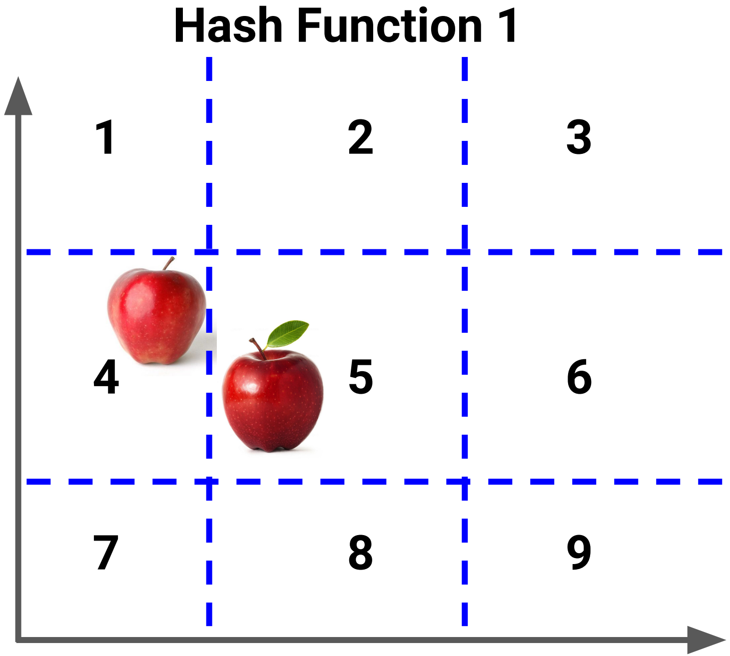
Indexing and searching floating-point vectors?

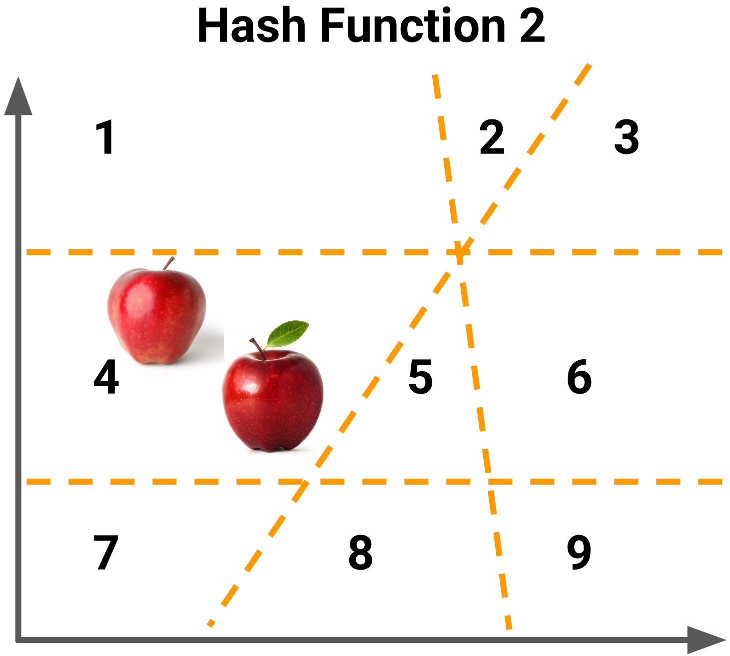
Locality Sensitive Hashing

- Represent vectors as discrete tokens while preserving similarity



Mapping LSH to Elasticsearch





More hash functions → More likely similar vectors share regions

How to store, query for vectors that share regions?

Mapping LSH to Elasticsearch

Query vector

```
"hashes": {
    "h1": 8
    "h2": 31
    "h3": 44
}
```

$Hash_1(vec) = 8$

$Hash_2(vec) = 31$

$Hash_3(vec) = 44$

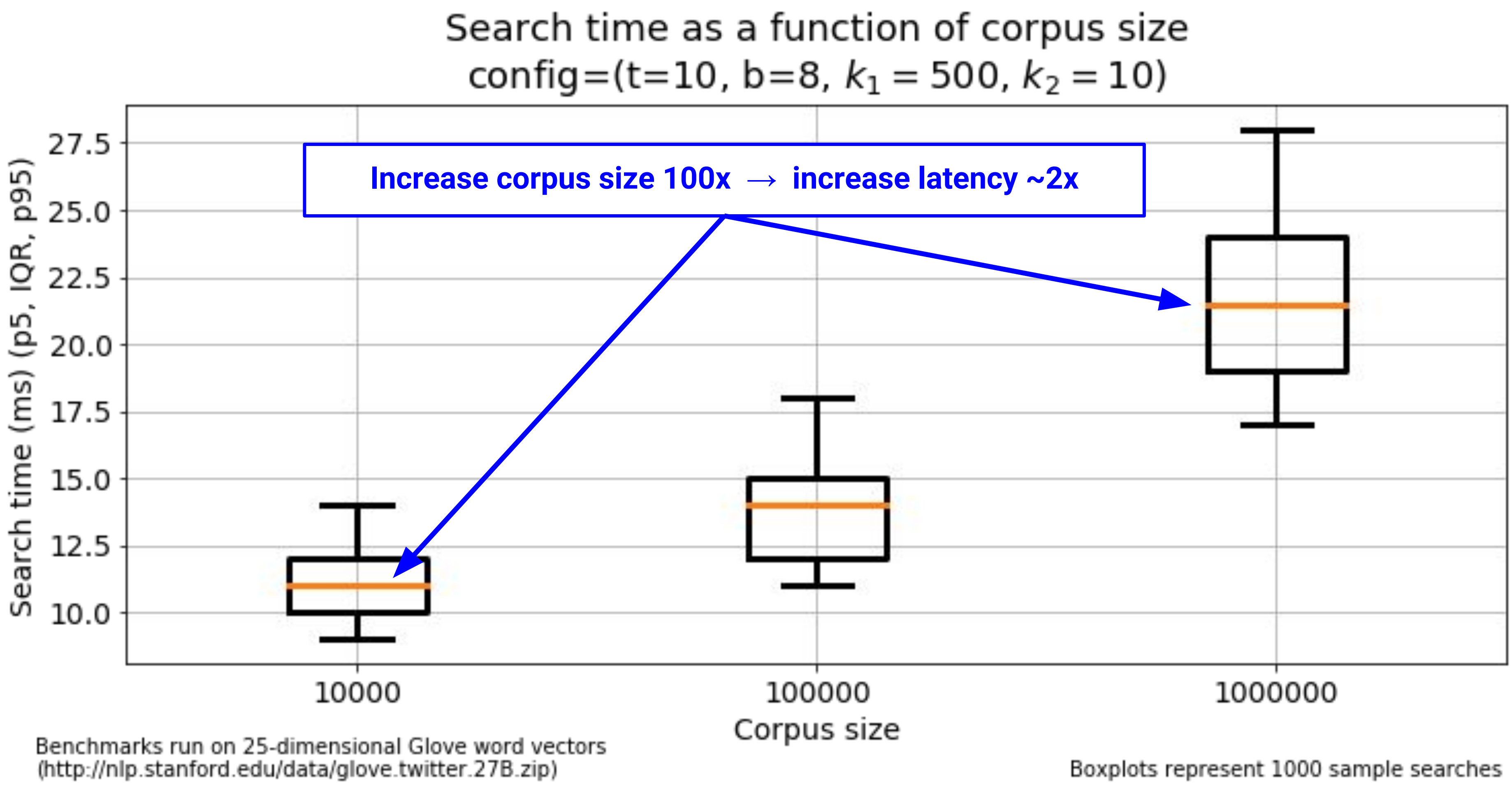
Candidate vector

```
"hashes": {
    "h1": 8
    "h2": 31
    "h3": 33
}
```

BooleanQuery

```
"_similarity": 2/3
```

- Return vectors with most matching key-value pairs
- Search time scales sub-linearly with respect to corpus size
  - Critical property for online setting


Search time as a function of corpus size
config=(t=10, b=8, $k_1 = 500$, $k_2 = 10$)
Increase corpus size 100x → increase latency ~2x
Search time (ms) (p5, IQR, p95)
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
10000
100000
1000000
Corpus size
Benchmarks run on 25-dimensional Glove word vectors
(http://nlp.stanford.edu/data/glove.twitter.27B.zip)
Boxplots represent 1000 sample searches

More details available

Implementation

Performance (speed and recall)

Image processing pipeline

… saved for Q and A, see supplementary slides

Alex Klibisz

Computer science, University of Tennessee

Enjoy productionizing ML Systems

Enjoy tennis, mountain-biking, traveling, and podcasts

Kayaking in Seward Alaska, May '17

Supplementary Material

Nearest Neighbors Plugin

- Implementation
- Parameters
- Benchmark - search latency
- Benchmark - recall
- Distributed performance
- Time complexity
- Locality sensitive hashing details
- Boolean query details

Full Pipeline

Image processing pipeline

ES-Aknn Implementation

Java with Gradle build system

Operates as a middleware

- All logic runs on Elasticsearch nodes
- Register endpoint handlers on start
- Parse HTTP request parameters and JSON requests when endpoints are hit
- Query and store documents using ElasticSearch Java API
- Construct and return JSON responses

ES-Aknn Parameters

`nb_tables`

- Number of hash functions applied to each vector
- Increasing this increases recall, decreases speed

`nb_bits_per_table`

- Each hash function partitions the space into $2^{\text{nb\_bits\_per\_table}}$ buckets
- Changing this affects recall, shouldn’t affect speed

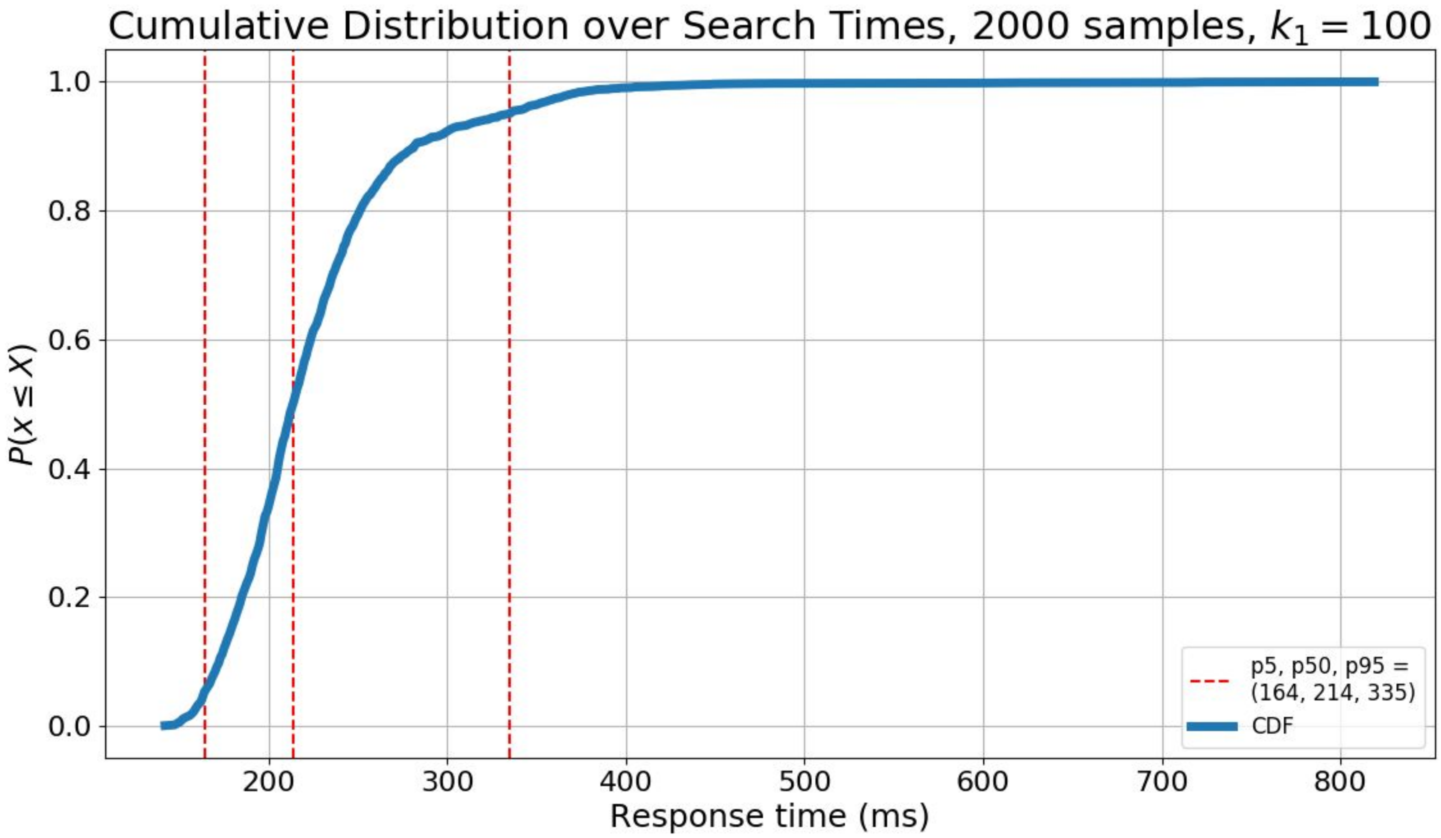
`k1`

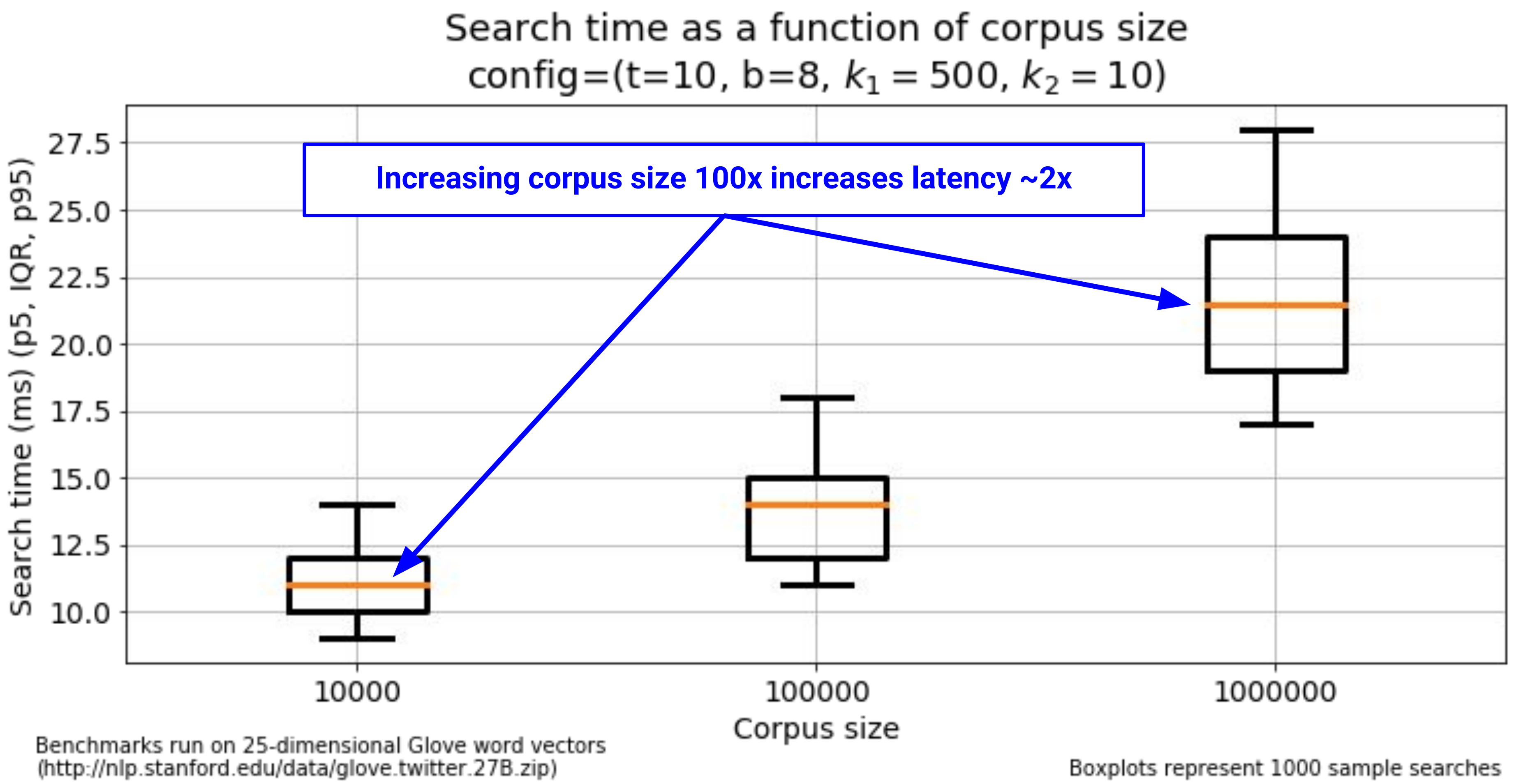
- Number of approximate neighbors considered for exact computation
- Increasing this increases recall, decreases speed

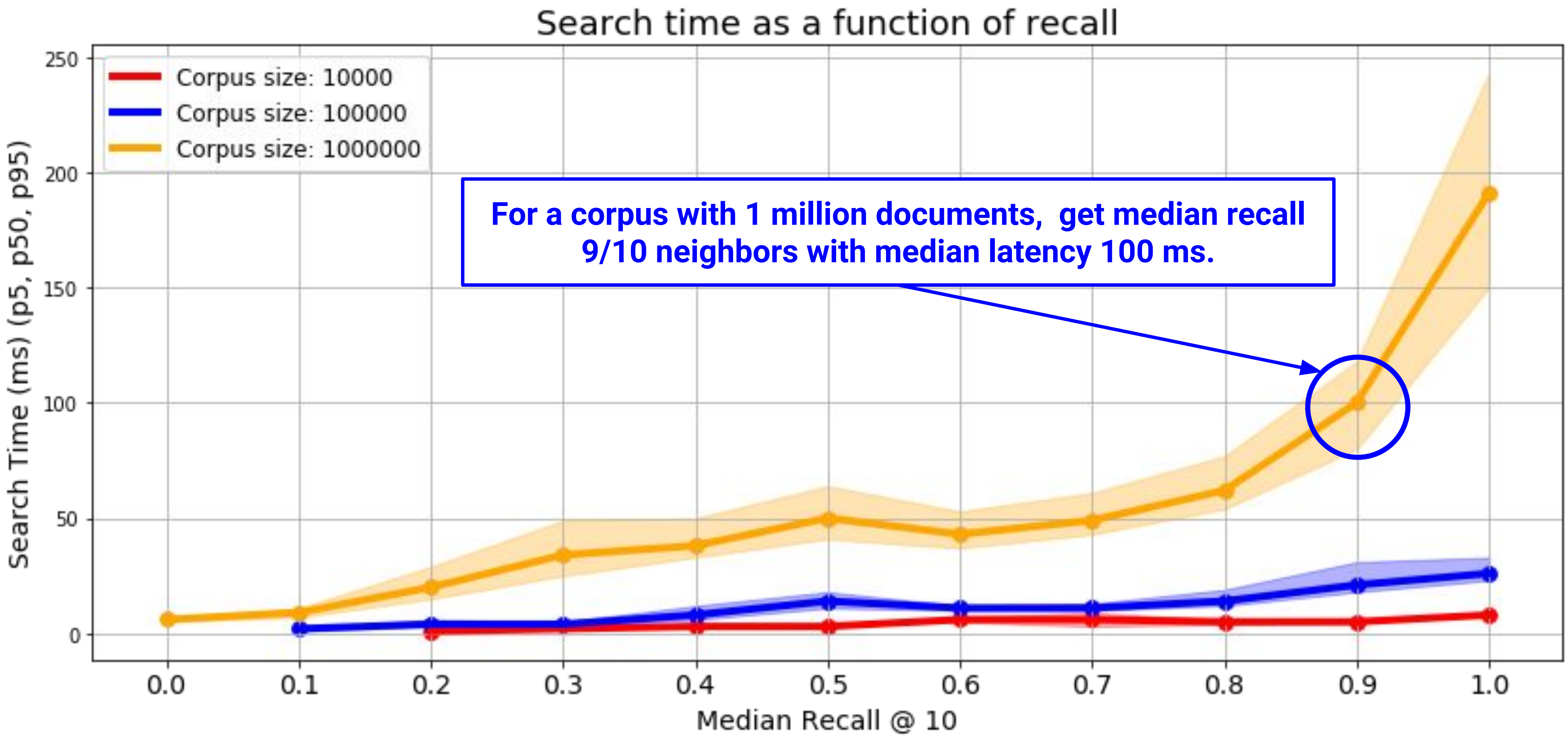
`k2`

- Number of exact neighbors returned
- Changing this has generally negligible effect on speed

Image Similarity Search Times




Search time as a function of corpus size
config=(t=10, b=8, $k_1 = 500$, $k_2 = 10$)
Increasing corpus size 100x increases latency ~2x
Search time (ms) (p5, IQR, p95)
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
10000
100000
1000000
Corpus size
Benchmarks run on 25-dimensional Glove word vectors
(http://nlp.stanford.edu/data/glove.twitter.27B.zip)
Boxplots represent 1000 sample searches



Benchmarks run on 25-dimensional Glove word vectors
(http://nlp.stanford.edu/data/glove.twitter.27B.zip)

Each point represents 1000 sample searches

ES-Aknn Distributed Performance

Setup

- 4x M5.xLarge (4-core, 16GB, SSD), 5x ES shards
- 300-dimensional Glove vectors

Indexing new documents

- 1600 new docs / second using round-robin posts containing 10K vectors

Concurrent searches

- 100 concurrent queries, LSH with 32 tables, 32 bits
- Latency for corpus of 1M vectors: mean ≅ 300 ms, stdv ≅ 80 ms

ES-Aknn Complexity

Search

```
O(BooleanQuery(N, nb_tables, k1) + (k1 × nb_dimensions) + partial-sort(k1, k2))
```

Find k1 candidate vectors with matching hashes | compute exact similarity | find the top k2

```
≅ O(BooleanQuery(N, nb_tables, k1)) << O(N)
```

In practice the Lucene BooleanQuery dominates runtime. It’s typically logarithmic or better but depends on configurations.

Indexing

```
O((nb_tables × nb_bits_per_table × nb_dimensions) + Index(nb_tables))
```

Hash the floating-point vector via dot products
The time spent on hashing vs. indexing depends on the batch size

Index as a standard ES document

LSH Implementation

Build LSH Model from sample of vectors

```
def make_lsh_model(nb_tables, nb_bits, nb_dimensions, vector_sample):
    # vector_sample: np arr w/ shape (2 * nb_tables * nb_tables, nb_dimensions).
    # normals, midpoints: np arrs w/ shape (nb_bits, nb_dimensions)
    # thresholds: np arrs w/ shape (nb_bits)
    # all_normals, all_thresholds: lists w/ one normal, one threshold per table.
    all_normals, all_thresholds = [], []
    for i in range(0, len(vector_sample), 2 * nb_bits):
        vector_sample_a = vector_sample[i:i + nb_bits]
        vector_sample_b = vector_sample[i + nb_bits: i + 2 * nb_bits]
        midpoints = (vector_sample_a + vector_sample_b) / 2
        normals = vector_sample_a - midpoints
        thresholds = np.zeros(nb_bits)
        for j in range(nb_bits):
            thresholds[j] = normals[j].dot(midpoints[j])
        all_normals.append(normals)
        all_thresholds.append(thresholds)
    return all_normals, all_thresholds
```

Github Gist, Linear Algebra Explanation

LSH Implementation

Compute the hashes for a vector

```python
def get_lsh_hashes(vec, all_normals, all_thresholds):
    # vec: np arr w/ shape (nb_dimensions, )
    # hashes: one hash per table.
    hashes = dict()
    for normal, thresholds in zip(all_normals, all_thresholds):
        hsh = 0
        dot = vec.dot(normal.T)  # shape (nb_bits,)
        for i, (d, t) in enumerate(zip(dot, thresholds)):
            if d > t:
                hsh += i ** 2
        hashes[len(hashes)] = hsh
    return hashes
```

[Github Gist](), [Linear Algebra Explanation]()

LSH Linear Algebra

Given two sample vectors v_1 and v_2, compute an equidistant hyperplane

- Hyperplane defined by midpoint `m`, normal vector `n`
  `m = (m₁, m₂) = (v₁ + v₂) / 2`
  `n = (n₁, n₂) = v₁ - m`

Given a new vector v, compute its hash relative a hyperplane defined by `m` and `n`

- The hash function should return 1 if the vector is “above” the hyperplane, 0 if the vector is “below” the hyperplane
  `h(v, m, n) = 1[n·v > n·m]` (indicator function)

- There are `nb_tables` * `nb_bits_per_table` such hash functions applied, each using an `m` an `n` computed from a random sample pair of vectors.
- Applying the hash functions for a single table also yields a binary tree.

ES-Aknn Boolean Query

```
GET demo/doc/1
{
  "_index": "demo",
  "_type": "doc",
  "_id": "1",
  "_source": {
     "h0": 10,
     "h1": 20,
     "h2": 30
  }
}
```

```
GET demo/doc/_search
{
 "query": {
  "bool": {
   "should": [
    {"term": {"h0": 10}},
    {"term": {"h1": 20}},
    ...
   ]
  }
 }
}
```

ES-Aknn Applications

General

- K-nearest-neighbors vector storage
- Low-latency K-nearest-neighbors search with growing corpus

Specific

- Image similarity search (a.k.a reverse image search)
- Audio similarity search
- Recommendation engines (especially when frequently adding/updating user/item vectors)

Full Pipeline

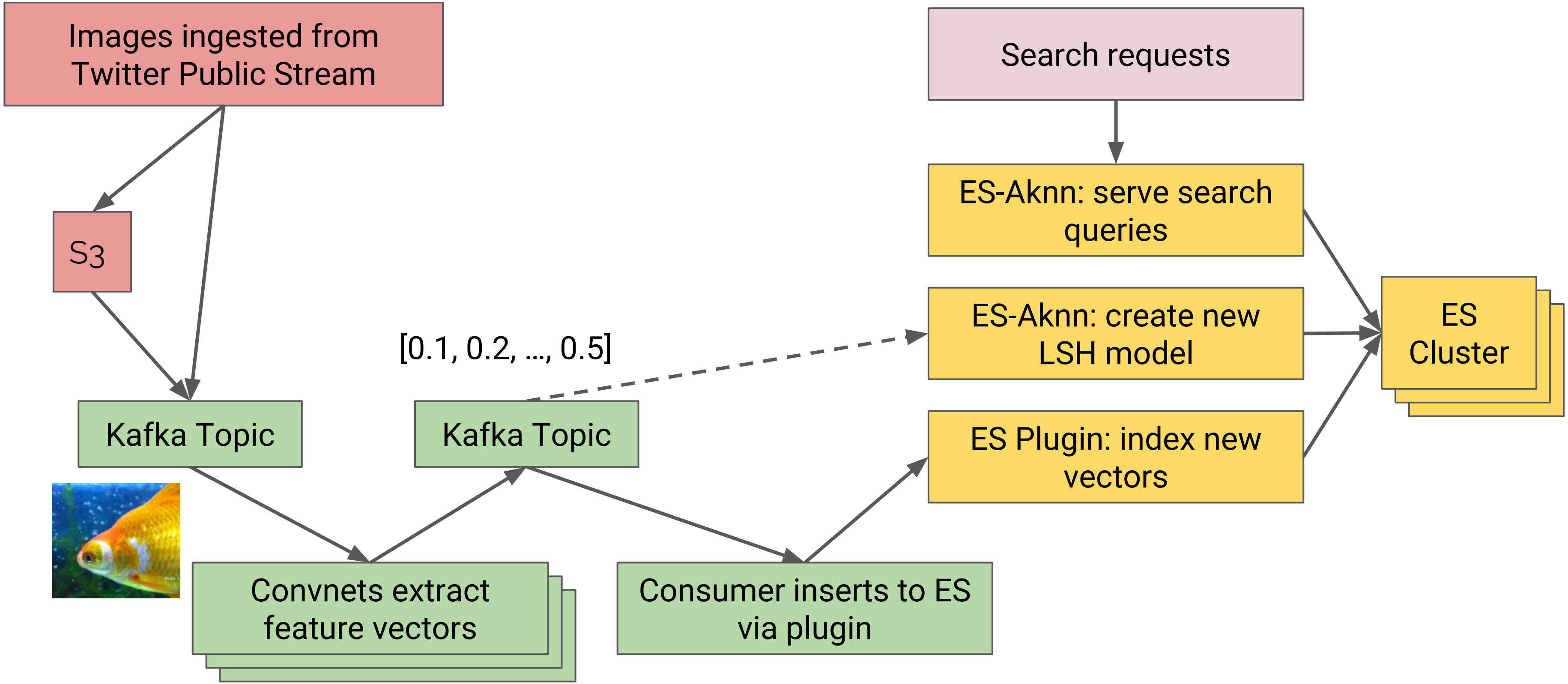


Image Processing Pipeline

Implementation

- Kafka as a distributed job queue with 10 EC2 instance workers
- Python Kafka consumers computing feature vectors
- Keras (MobileNet) pre-trained convnet - use `conv_preds` layer for output
- Consumers pull images from S3 (faster than funneling through Kafka)
- Parallelized S3 requests (ThreadPoolExecutor), image preprocessing (Multiprocessing Pool)

Performance

- 40 images / node / second with K80 GPU ($0.3/hr spot instance)
- 33 images / node / second with 36-core CPU ($0.6/hr spot instance)

Helpful Resources

LSH lectures by Victor Lavrenko

ES plugin cookiecutter template

ES Ingest-OpenNLP plugin

Presentation about ANNOY by Erik Berhnardsson