Vector Search: The Hard way

Chicago Search Meetup
Sept, 2023

A series of educational mistakes

© Doug Turnbull (http://softwaredoug.com), all opinions my own, not my employer

Obligatory Bio Slide

👋 Hi I'm Doug (@softwaredoug everywhere)

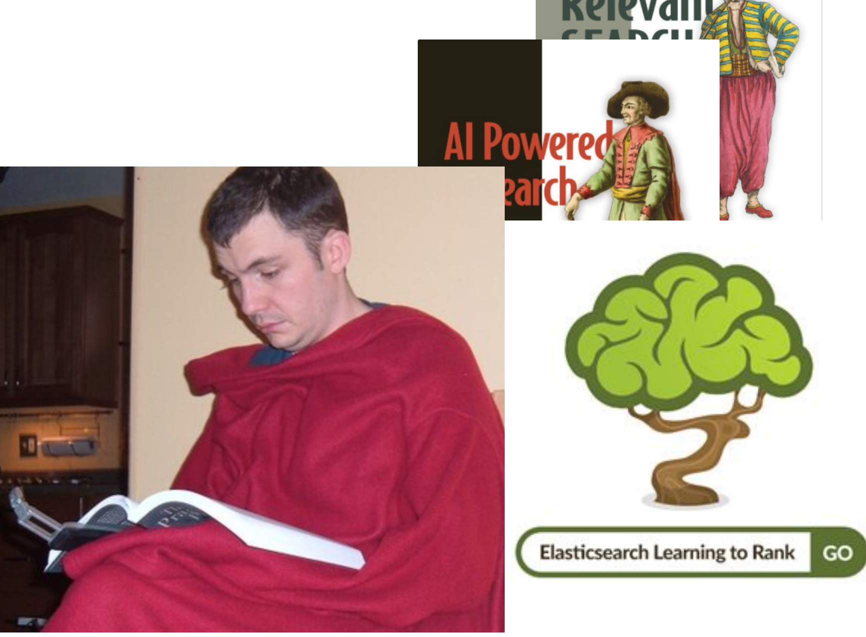
Long-time search enthusiast... Not yet (never?) an expert

I wrote some search books, did some open source

I work at Reddit

I worked at Shopify & OpenSource Connections in search

I blog here: http://softwaredoug.com

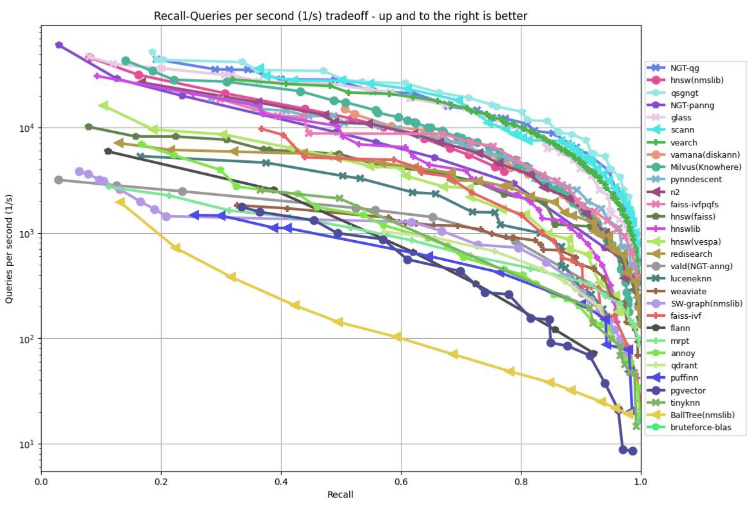


:itme:

The Problem

My Strawman(?)

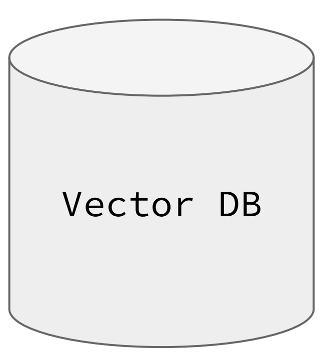
Build Time → Search Time



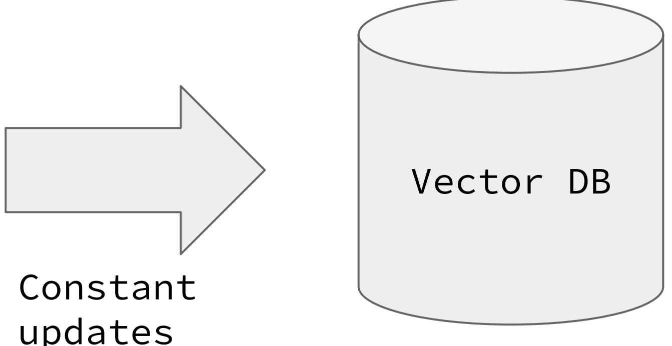
(Searching static index)

https://ann-benchmarks.com/

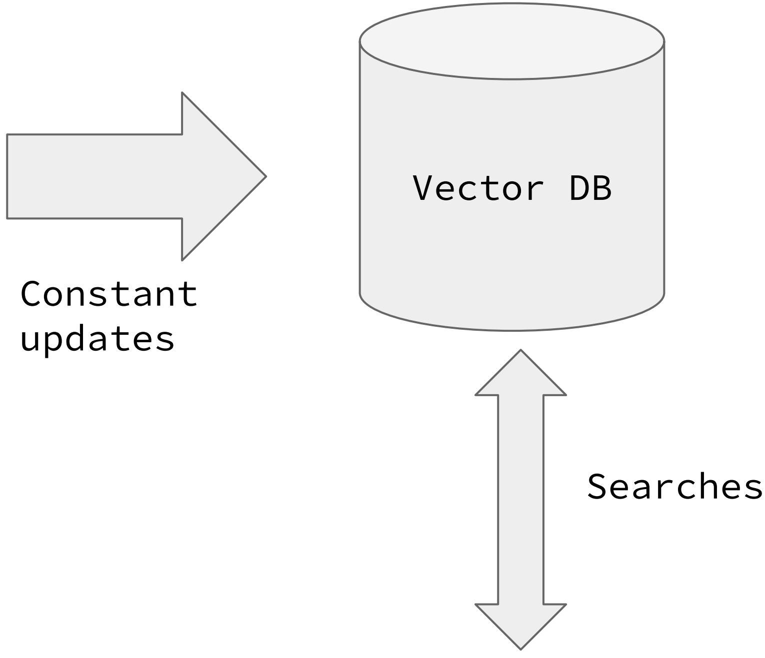
Real Life systems…



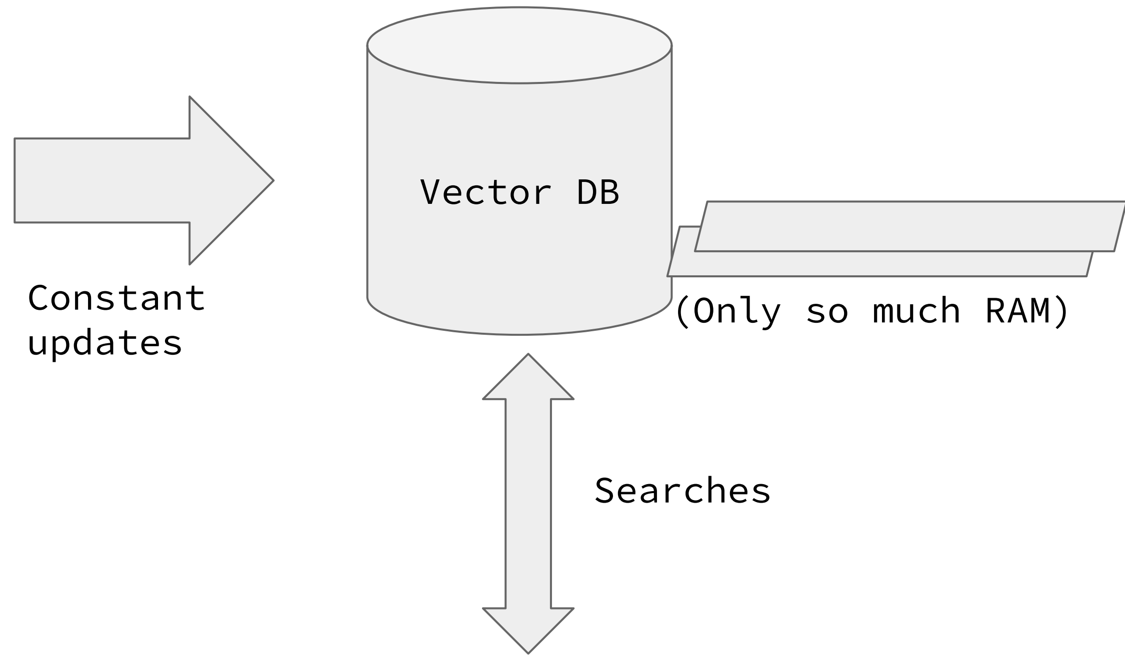
Updates are constant



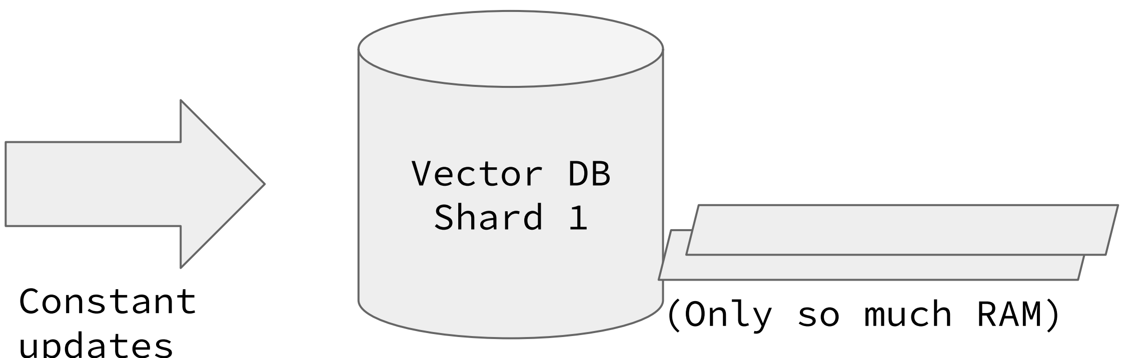
Searches are constant

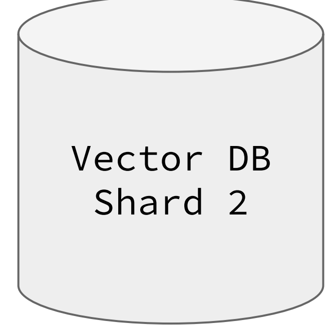


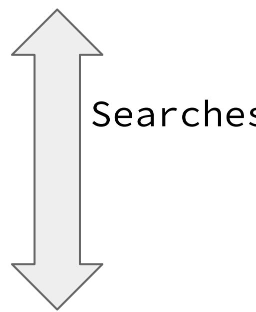
Limited memory



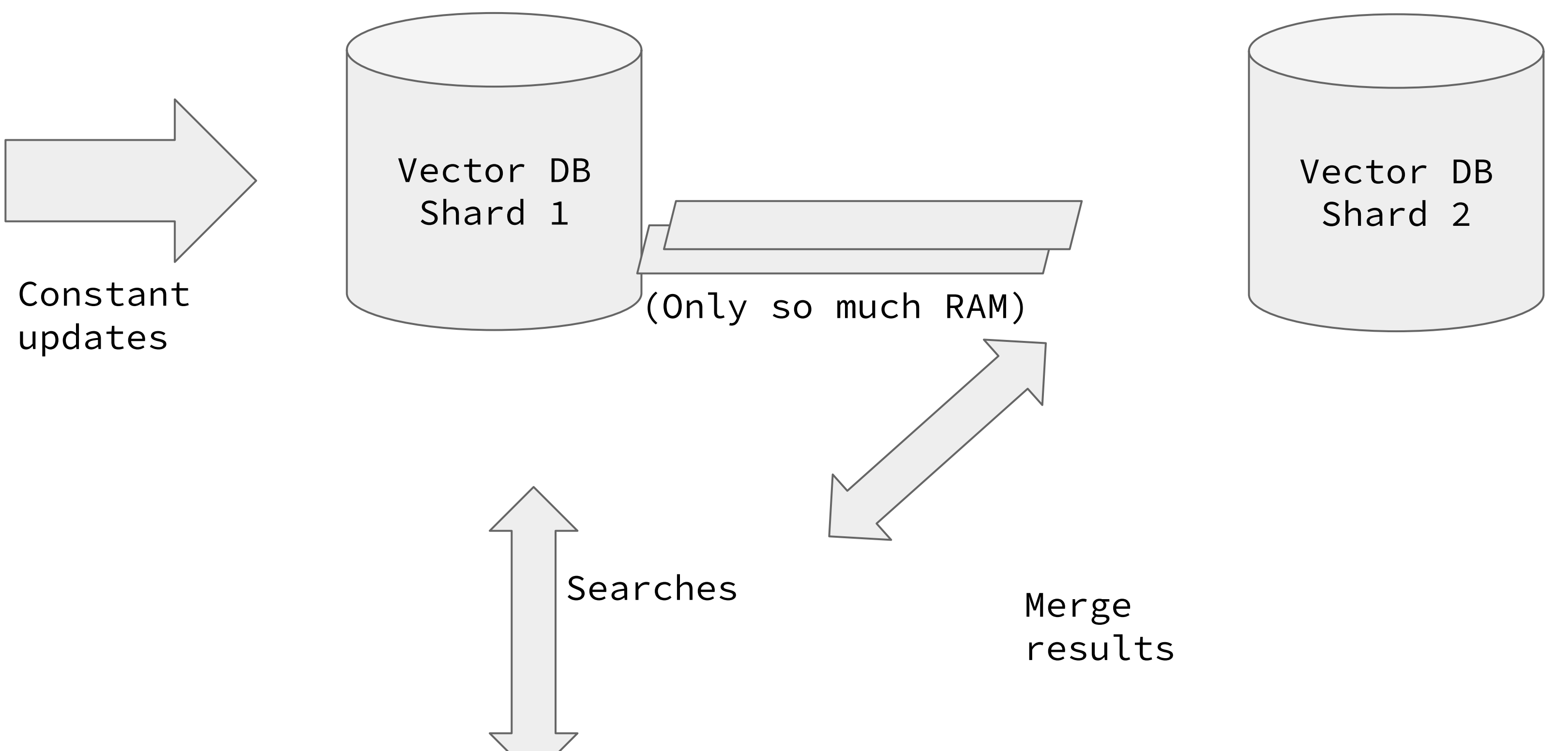
Sharded







Sharded



Real life constraints

Current Vector DB systems

- High recall
- Low latency

Real life constraints

Current Vector DB systems

- High recall
- Low latency

“Benchmark” regime

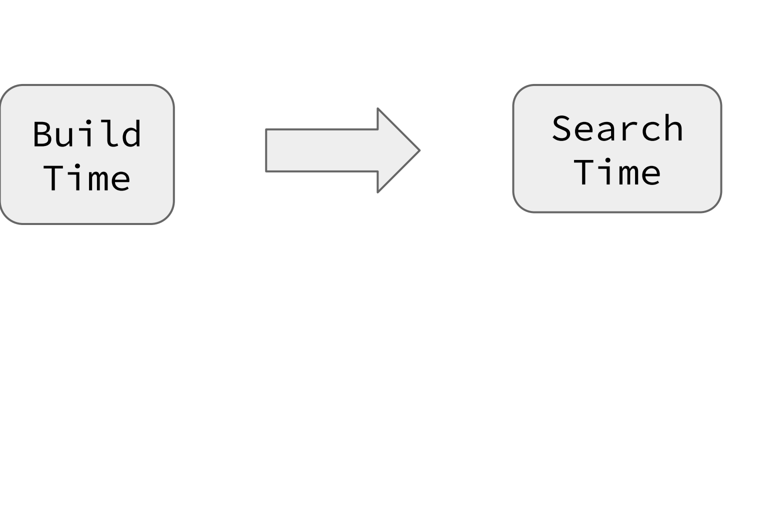
Real Life

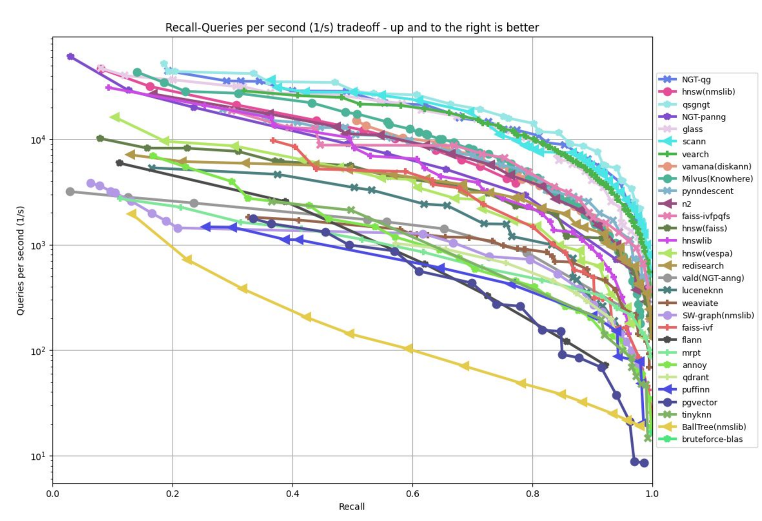
- Updates need to happen constantly
  - Index not built up-front
- RAM can’t be absorbed by millions of floating point values
- We need to shard, merge indices etc

IRL

Start at the end

What results from just these incentives?

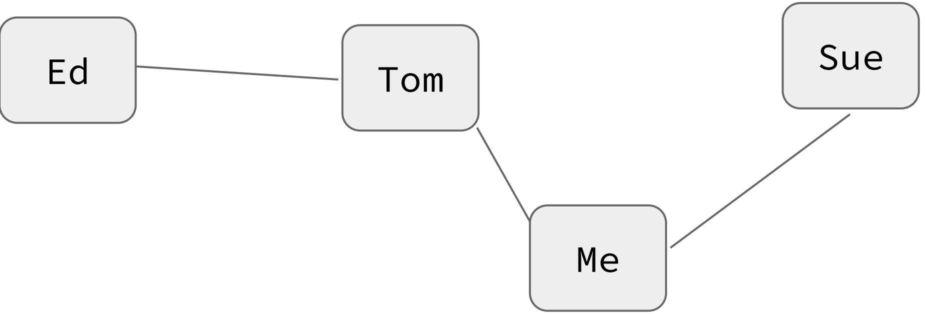




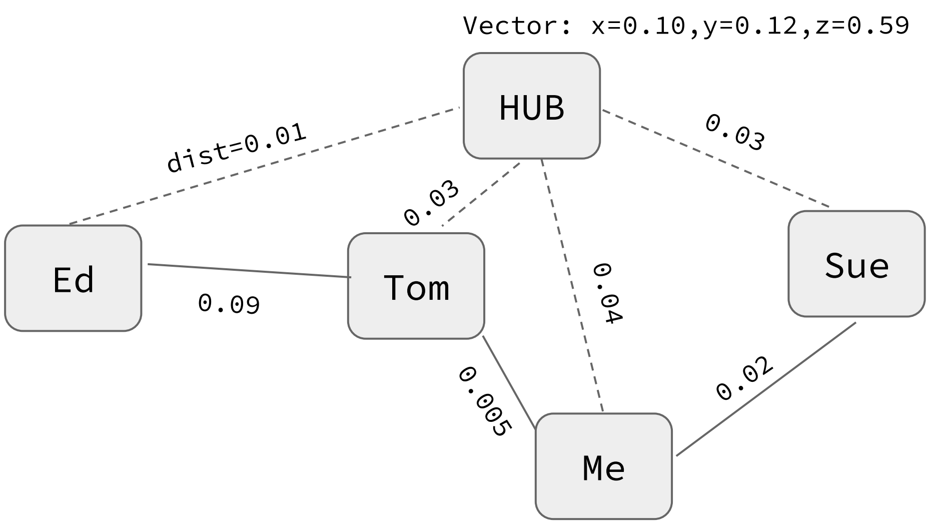
(Searching static index)

https://ann-benchmarks.com/

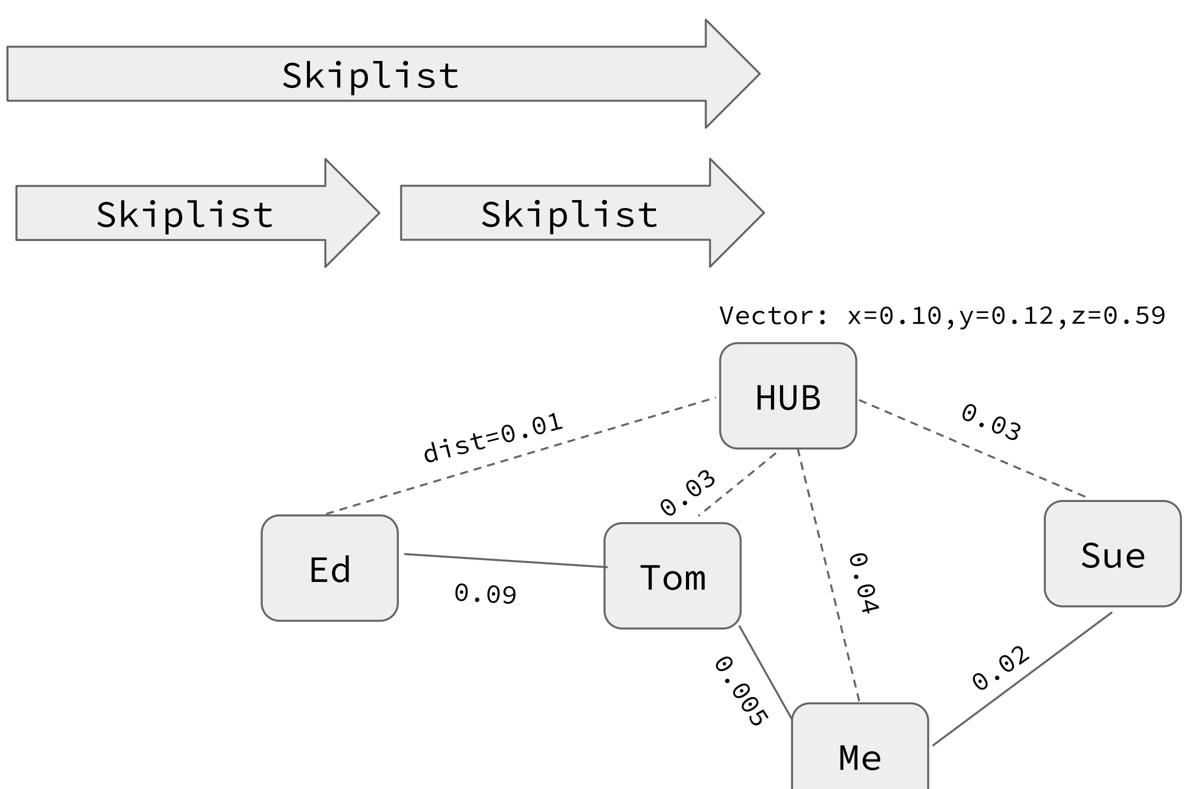
… you precompute the right answer



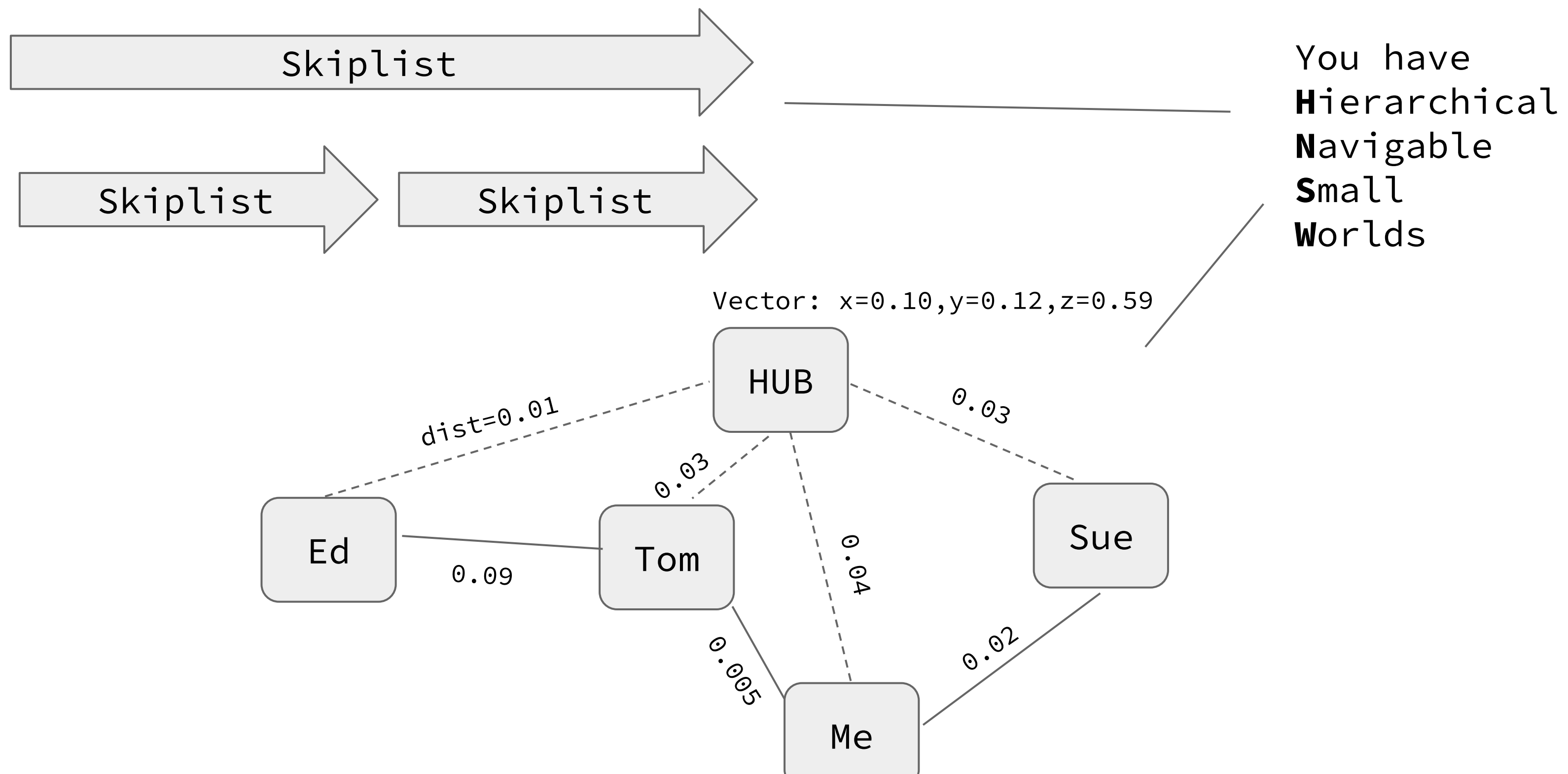
… you create a ‘hub’ for that area



… you create a way to get there fast



… you create a way to get there fast



You have
Hierarchical
Navigable
Small
Worlds

“Graph” regime of today

Skiplist

Skiplist

Vector: x=0.10,y=0.12,z=0.59

HUB

dist=0.01

0.03

0.03

0.04

Ed

0.09

Tom

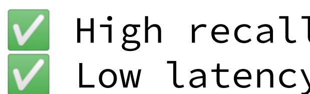
Sue

0.005

0.02

Me

<u>Current Vector DB systems</u>



✅ High recall
✅ Low latency

“Benchmark” regime

“Zip code” metaphor

Nat’l Distribution

Regional

Vector: x=0.10,y=0.12,z=0.59

Post Office

dist=0.01

0.03

0.03

0.04

Ed

0.09

Tom

Sue

0.005

0.02

Me

Current Vector DB systems

“Benchmark” regime

“Graph” regime

Skiplist

Skiplist

Vector: x=0.10,y=0.12,z=0.59

HUB

dist=0.01

0.03

0.03

0.04

Ed

0.09

Tom

Sue

0.005

0.02

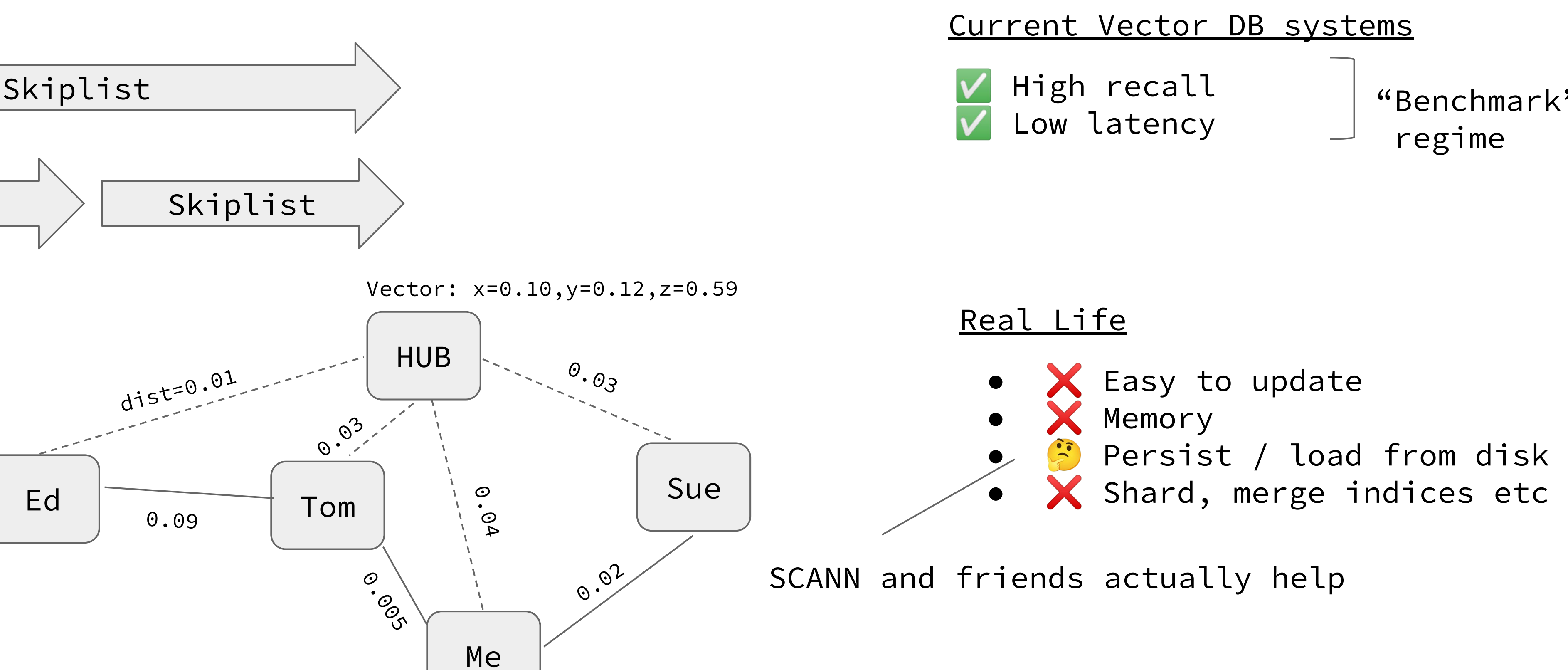
Me

Current Vector DB systems

Real Life

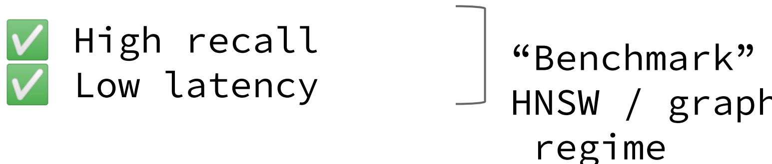
- ❌ Easy to update
- ❌ Memory
- ❌ Persist / load from disk
- ❌ Shard, merge indices etc

“Graph” regime



Are we really solving this first principles?

Current Vector DB systems



Real Life

- ❌ Easy to update
- ❌ Memory
- ❌ Disk
- ❌ Shard, merge indices etc

Lipstick on a pig?

Current Vector DB systems

Other requirements

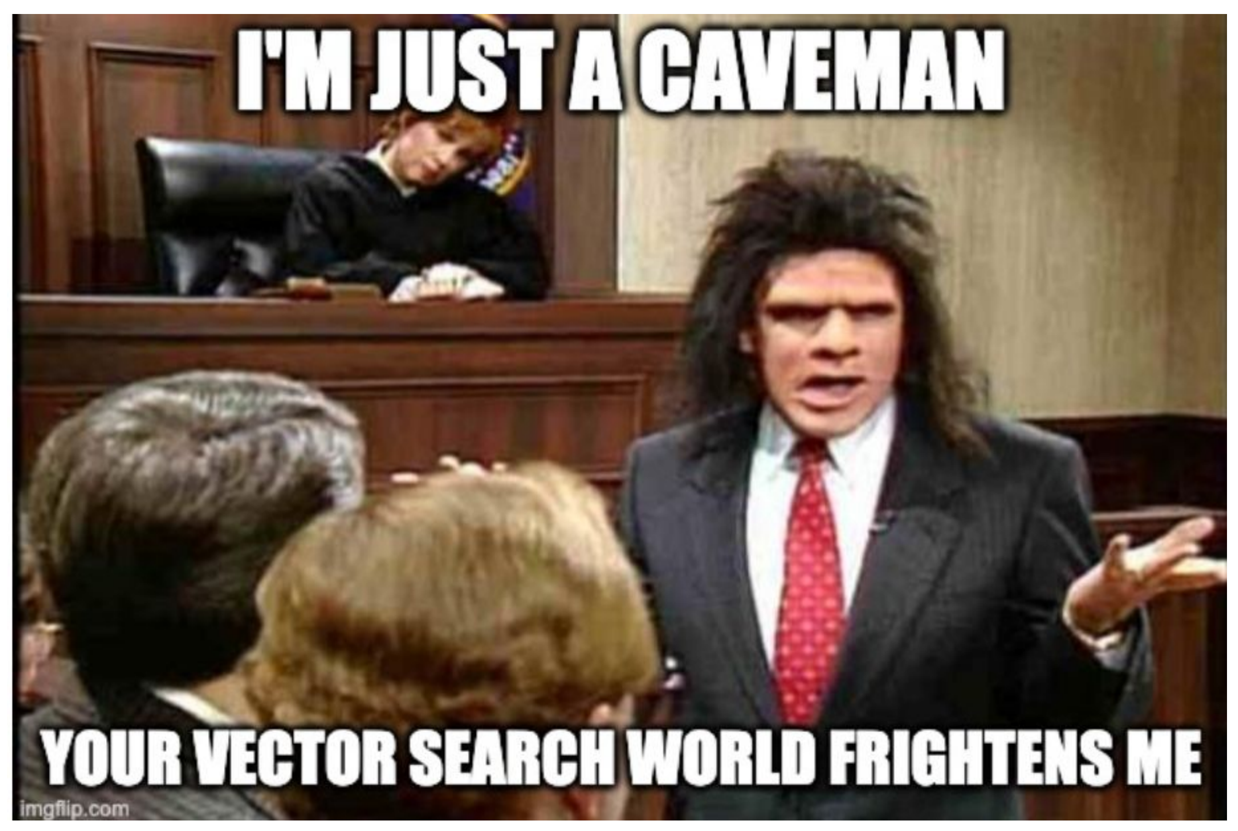
- ❌Recover cosine similarity
- ❌Integration with traditional search
- ❌Beyond just “recall” over top N
- ...?

Real Life

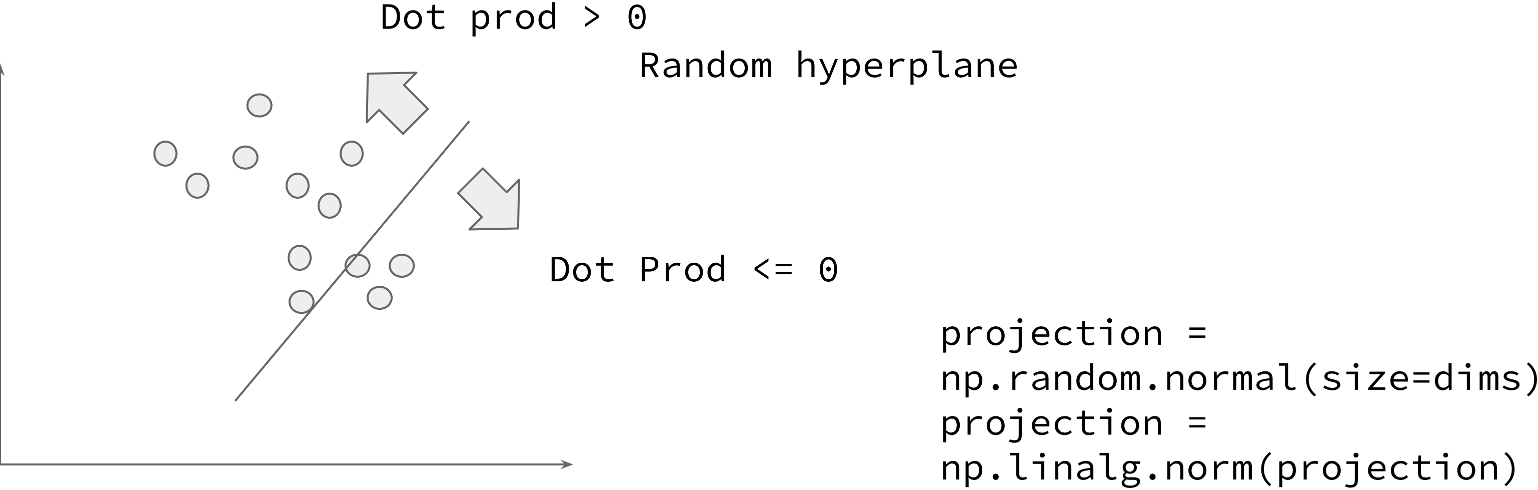
- ❌ Easy to update
- ❌ Memory
- ❌ Disk
- ❌ Shard, merge indices etc

Hashing

I’m just a caveman

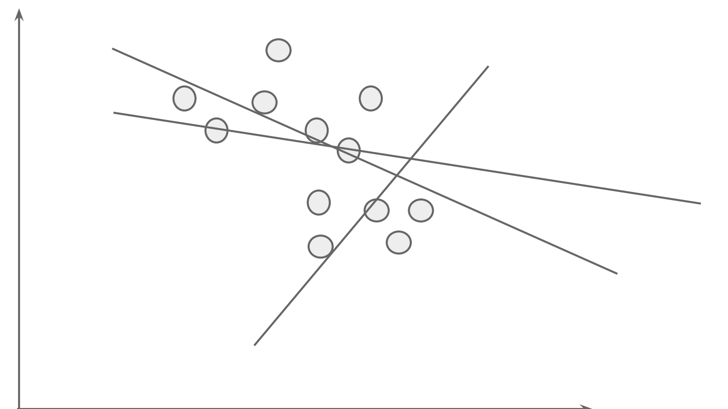


Dumb hashes




```
projection =
np.random.normal(size=dims)
projection =
np.linalg.norm(projection)
```

Times many many more



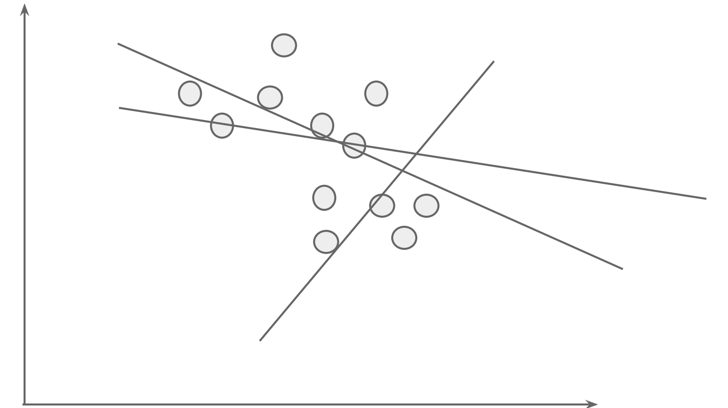
Caveman Lawyer Nearest Neighbors

Insert:

```
bit_mask = “”
for proj in projections:
    dotted = np.dot(proj, new_vector)

    if dotted > 0:
       bit_mask += “1”
    else:
       bit_mask += “0”

hashed_vectors.append(bit_mask)
```

Dumb list

Query

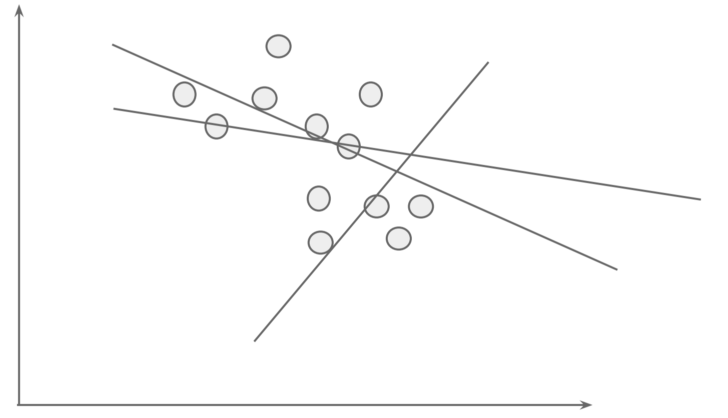


Caveman Lawyer Nearest Neighbors

```
Query Hashed        : 11001111
Index Vector Hashed: 11000100
```

Same side of projection 5 / 8 times
Hamming similarity 0.625

Times many many more



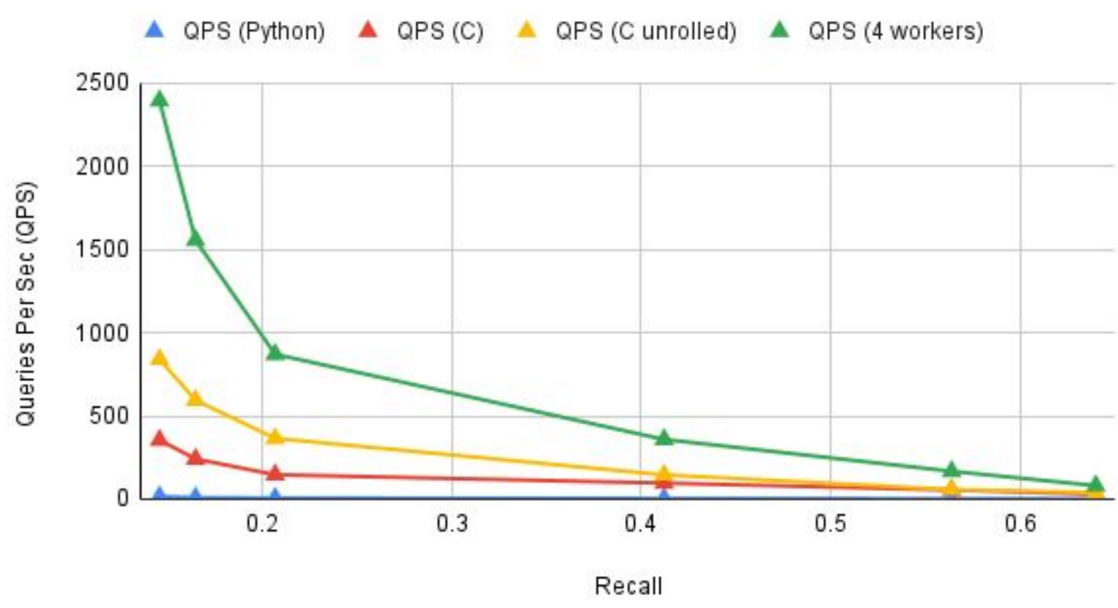
Caveman Lawyer Nearest Neighbors

<u>Query:</u>

```
query_bit_mask = /*same as last slide*/

for hashed in projections:

      # Saves some operations by just counting
OPPOSITE, lower here is more similar
      xord = hashed ^ query_bitmask
      num_bits_set = popcount(xord)

      if num_bits_set < min_so_far:
         … append to top N…
         min_so_far = num_bits_set
```

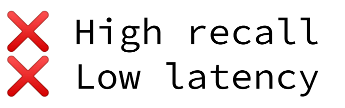
Results

"Benchmark"
HNSW / graph
regime



From: https://softwaredoug.com/blog/2023/08/21/implementing-random-projections

Results



"Benchmark" HNSW / graph regime

QPS (Python) QPS (C) QPS (C unrolled) QPS (4 workers)

Queries Per Sec (QPS)

2500
2000
1500
1000
500
0

0.2 0.3 0.4 0.5 0.6

Recall

✅ Easy to update (append!)
✅ Very little RAM
✅ Dumb as 💩 to merge / shard

IRL Caveman engineer concerns

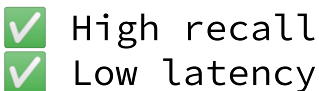
From: https://softwaredoug.com/blog/2023/08/21/implementing-random-projections

Highly Parametric

Pre-compute the teeniest structures (ie graphs)

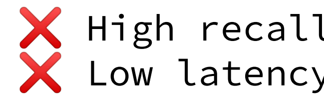
Highly Nonparametric

Assume nothing about vector space



✅ High recall
✅ Low latency

❌ Easy to update (append!)
❌ RAM
❌ Disk
❌ Dumb as 💩 to merge / shard



❌ High recall
❌ Low latency

✅ Easy to update (append!)
? Very little RAM
✅ Dumb as 💩 to merge / shard

Highly Parametric
Pre-compute the teeniest structures (ie graphs)

Highly Nonparametric
Assume nothing about vector space

✅ High recall
✅ Low latency

❌ Easy to update (append!)
❌ RAM
❌ Disk
❌ Dumb as 💩 to merge / shard

Global structure to maintain, update, merge…

❌ High recall
❌ Low latency

✅ Easy to update (append!)
? Very little RAM
✅ Dumb as 💩 to merge / shard

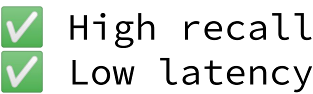
Little global structure to maintain, update, merge..

Partitioning

Most try to mitigate HNSW / graphs

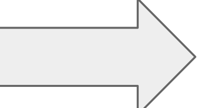
<u>Highly Parametric</u>
Pre-compute the teeniest structures (ie graphs)

<u>Highly Nonparametric</u>
Assume nothing about vector space



✅ High recall
✅ Low latency

❌ Easy to update (append!)
? Very little RAM
❌ Dumb as 💩 to merge / shard



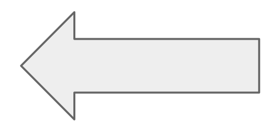
Make them easier to update, etc

What about the other way?

<u>Highly Nonparametric</u>

Assume nothing about vector space

❌ High recall
❌ Low latency

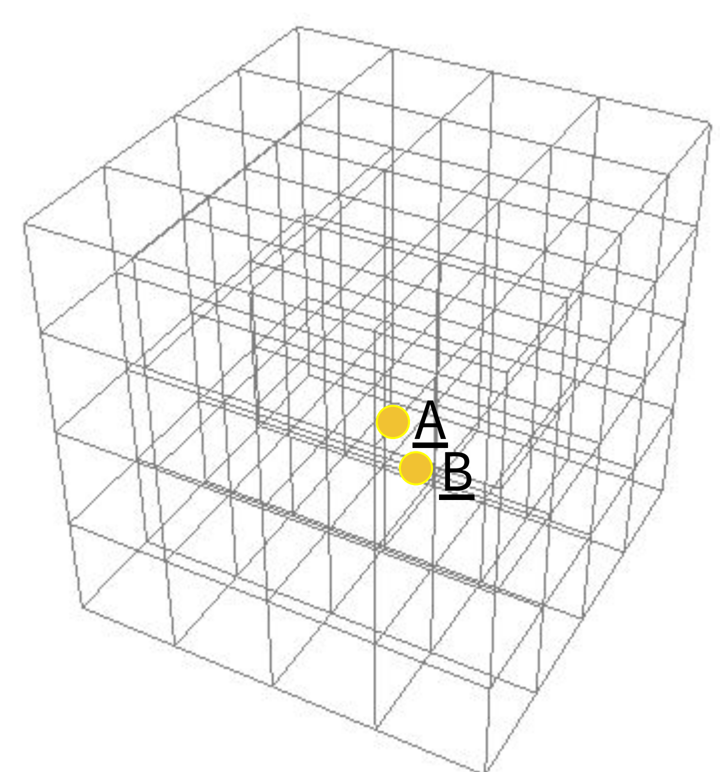


Improve speed & recall but preserve operational dumb caveman concerns

✅ Easy to update (append!)
? Very little RAM
✅ Dumb as 💩 to merge / shard

Little global structure to maintain, update, merge..

Why can’t we just…

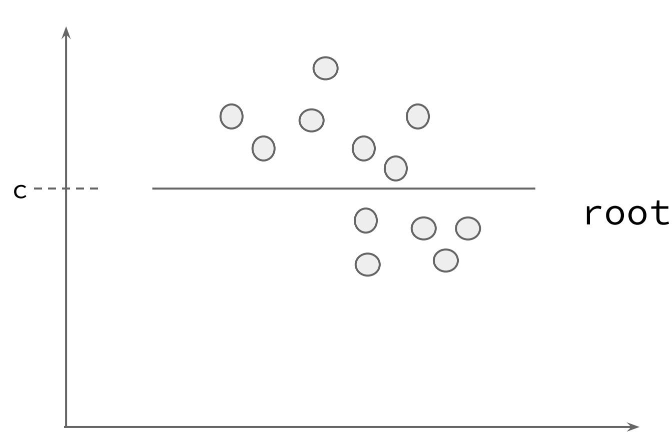


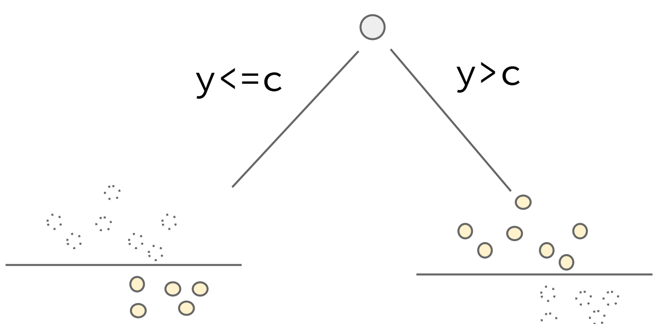
Like assign “subcubes” or somesuch that A and B share?

Index
(1, 2, 0) -> A, B
(0, 1, 3) -> C, D

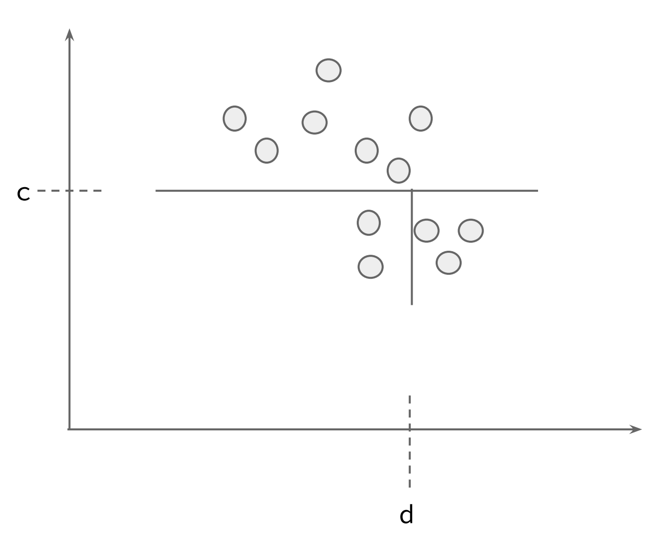
Img from https://stackoverflow.com/questions/69172349/r-project-plot-a-3d-mesh-grid-using-plot3d-package

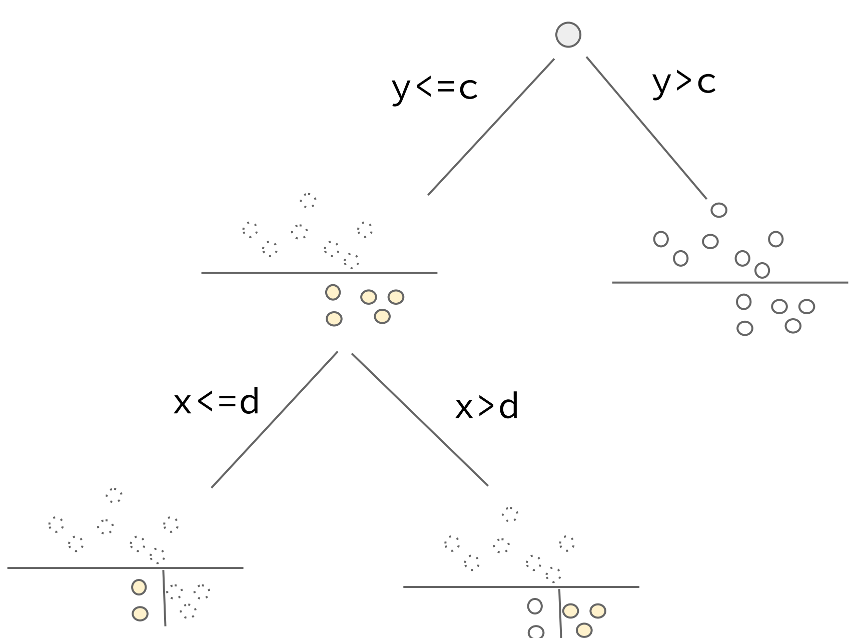
KD Trees



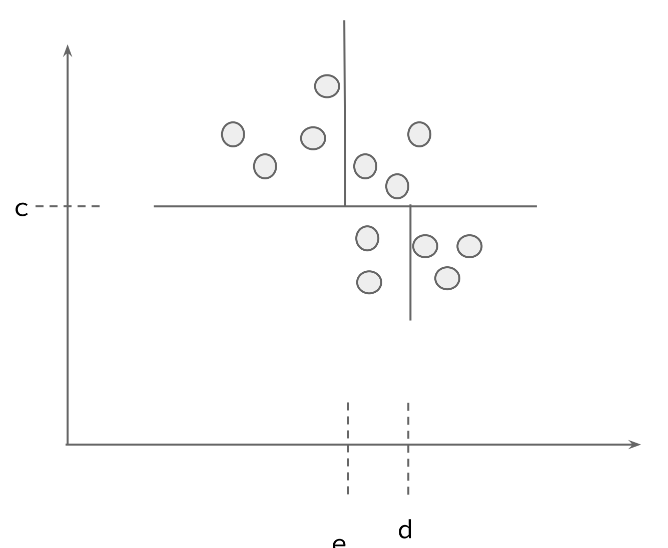


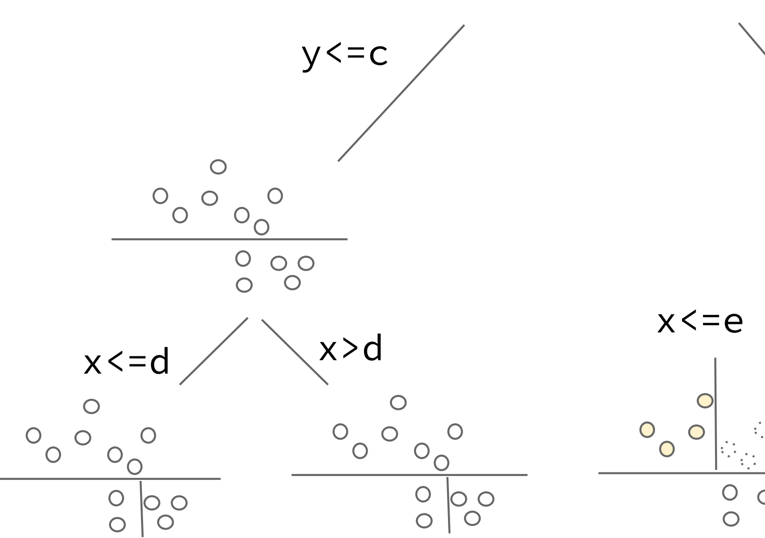
KD Trees



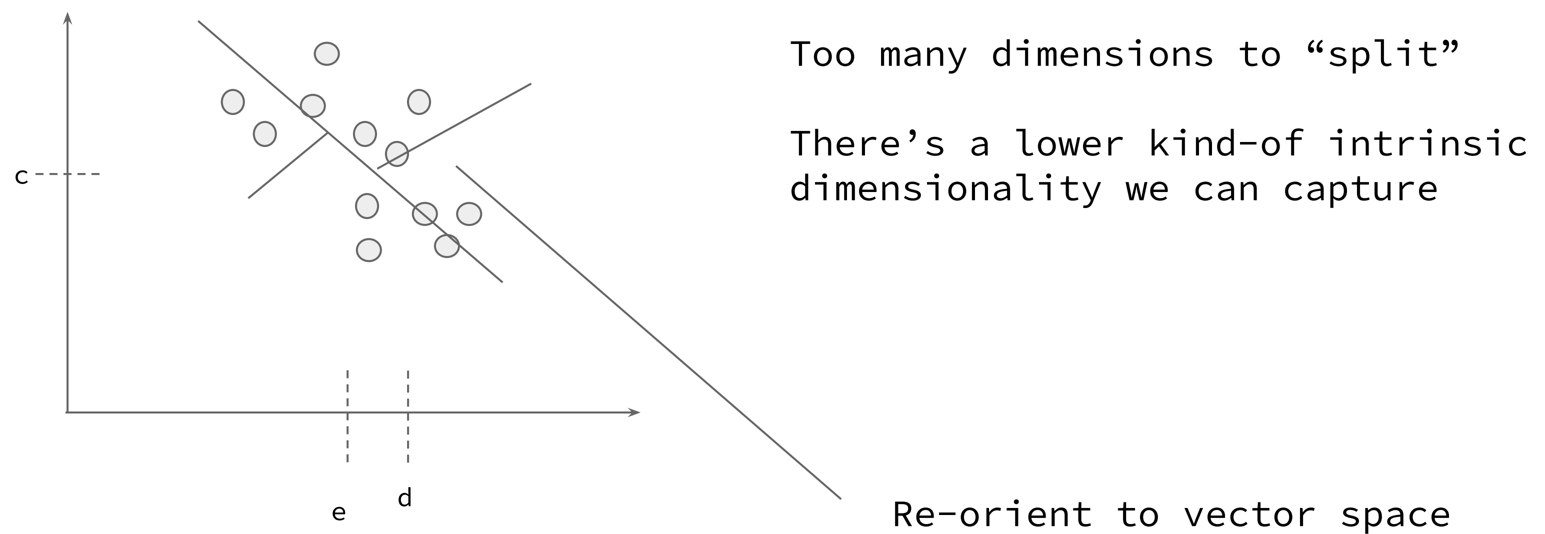


KD Trees





KD Trees break down...

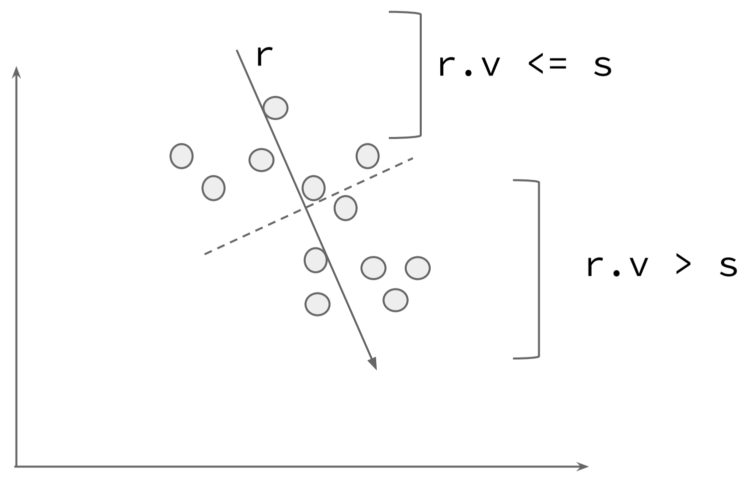


Too many dimensions to “split”

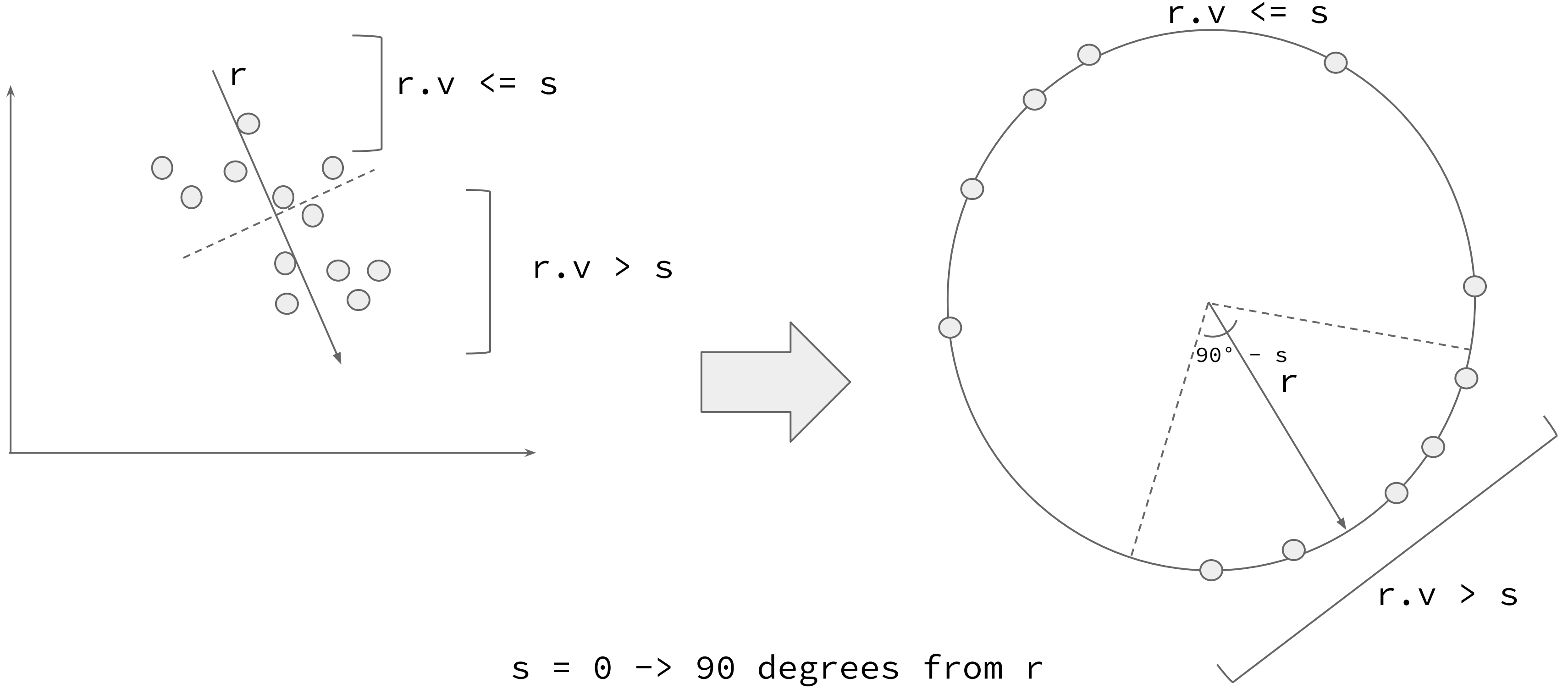
There’s a lower kind-of intrinsic dimensionality we can capture

Re-orient to vector space

Random Projection Trees

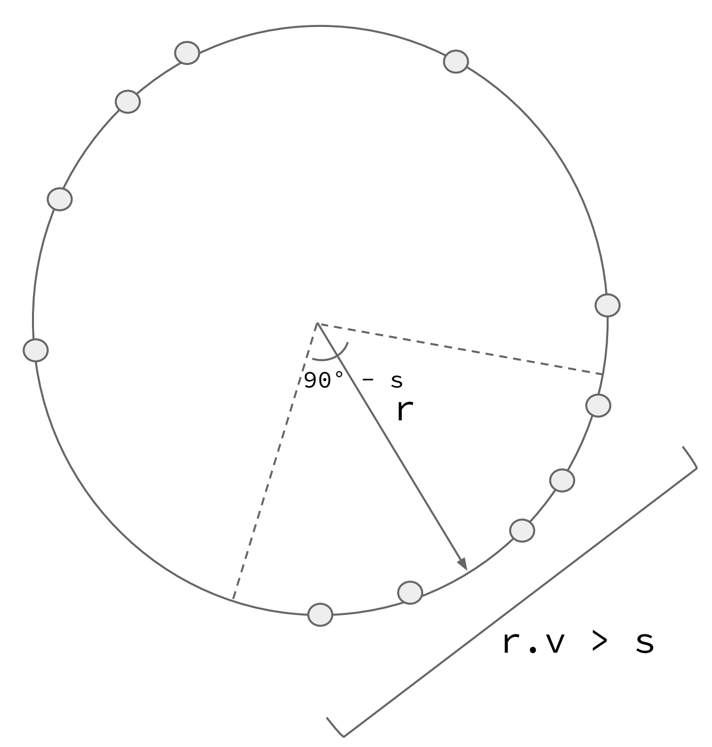


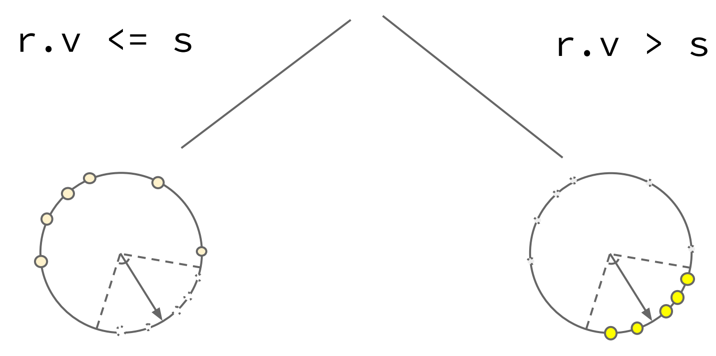
Radial Projection on unit sphere



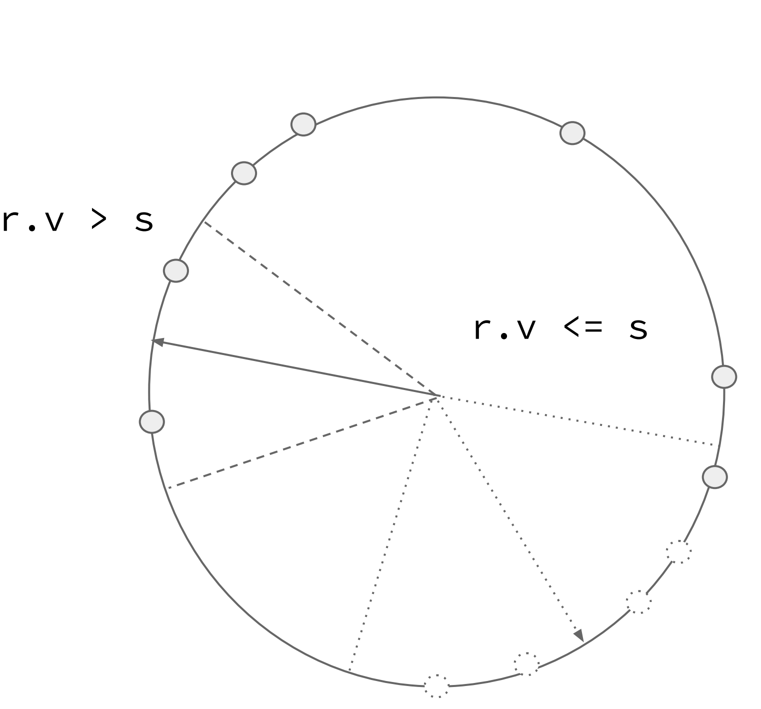
As a tree…

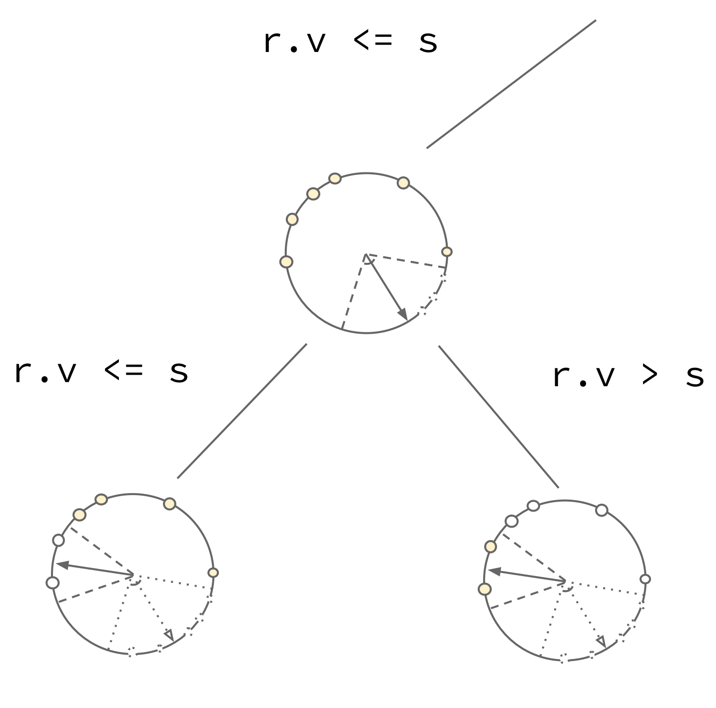
r.v <= s





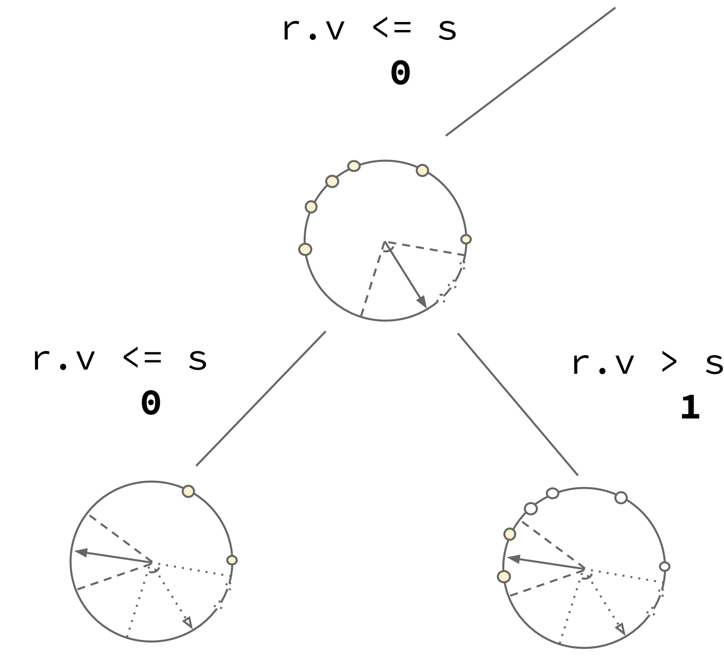
Nested – left hand side shown





Represent tree path as binary hash

```
--------------------------------
             king: 00100
            kings: 01110
           prince: 00110
            queen: 01110
             King: 01110
           throne: 01000
          kingdom: 00110
             lord: 00110
            royal: 01110
            reign: 01000
--------------------------------
         Fernvale: 00100
              IBG: 01010
     ReachedSorry: 10100
           MapsUV: 11110
ScoresAndOdds.com: 10010
             BRSC: 11000
      Lifestreams: 01010
      IMMOLATION1: 10100
            Purga: 01100
     Miniaturezed: 01000
```



First split... Pretty good!

```
king: 00100
kings: 01110
prince: 00110
queen: 01110
King: 01110
throne: 01000
kingdom: 00110
lord: 00110
royal: 01110
reign: 01000
```

```
python benchmark/glove.py --algorithm rp_tree --workers 1 --verbose --seed 0
```

First split… Pretty good!

```
   king: 00100
  kings: 01110
 prince: 00110
  queen: 01110
   King: 01110
 throne: 01000
kingdom: 00110
   lord: 00110
  royal: 01110
  reign: 01000
```

🤔- some neighbors sliced off…

```
python benchmark/glove.py --algorithm rp_tree --workers 1 --verbose --seed 0
```

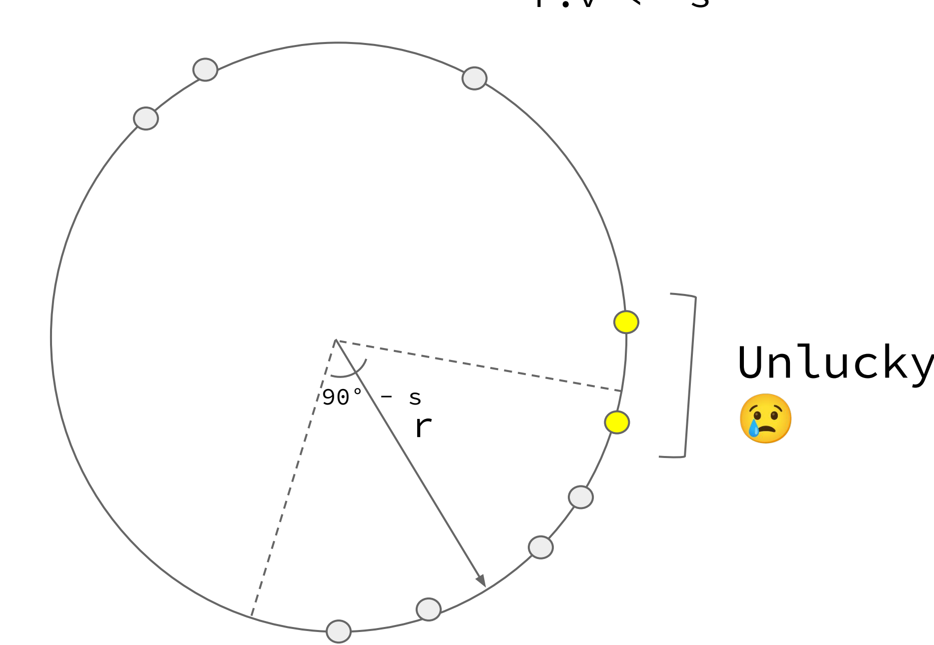
First split… Pretty good!

```
king: 00100
kings: 01110
prince: 00110
queen: 01110
King: 01110
throne: 01000
kingdom: 00110
lord: 00110
royal: 01110
reign: 01000
```

🤔- some neighbors sliced off…

```
python benchmark/glove.py --algorithm rp_tree --workers 1 --verbose --seed 0
```

OK, we expect this




```
python benchmark/glove.py --algorithm rp_tree --workers 1 --verbose --seed 0
```

... But how often?

300D Glove embeddings,
“king”, its neighbors, and 1000 random points

```
vectors = np.load("test/glove_sample.npy")
vector_idx = 774 # idx being searched 774 king


for seed in range(0, 400):

     np.random.seed(seed)
     splitter = rptree_proj_maxvar_chooserule(vectors)

     left, right = splitter.split(vectors)
     assert len(left) != 0
     assert len(right) != 0

     if nn_on_correct_side(vectors, vector_idx, left, right):
         pass_count += 1
         print("✅")
     else:
         failed_seeds.add(seed)
         print("❌")

     runs += 1
```

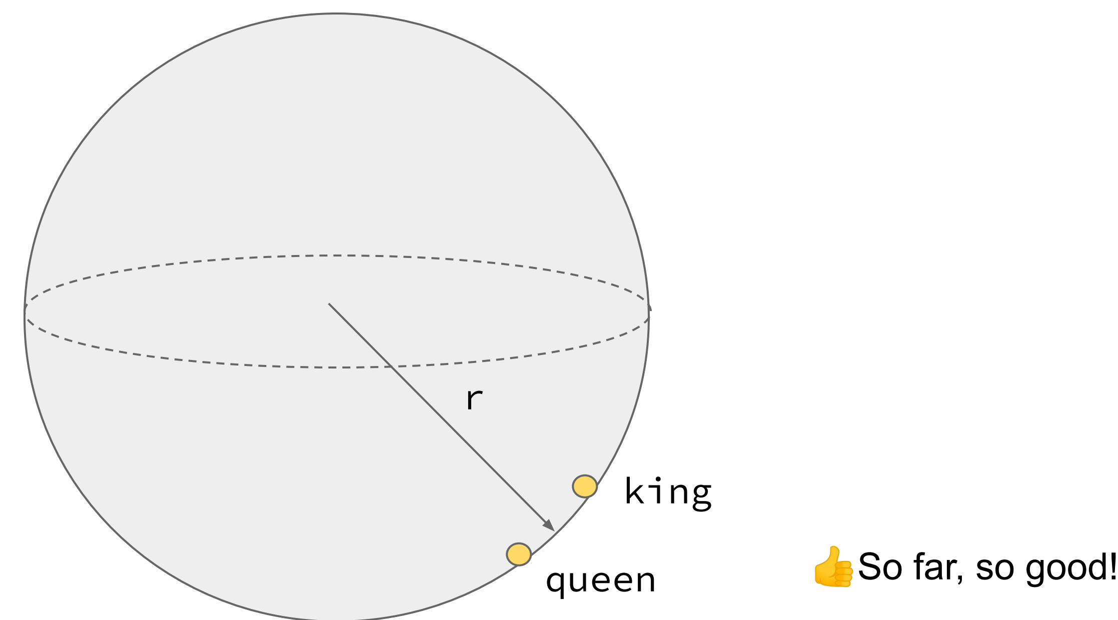
Neighbors stay together:

Random Point: ~60% of time
Outlier (king): ~80% of the time

Maybe slight improvements:
Choosing projection with
most variance?

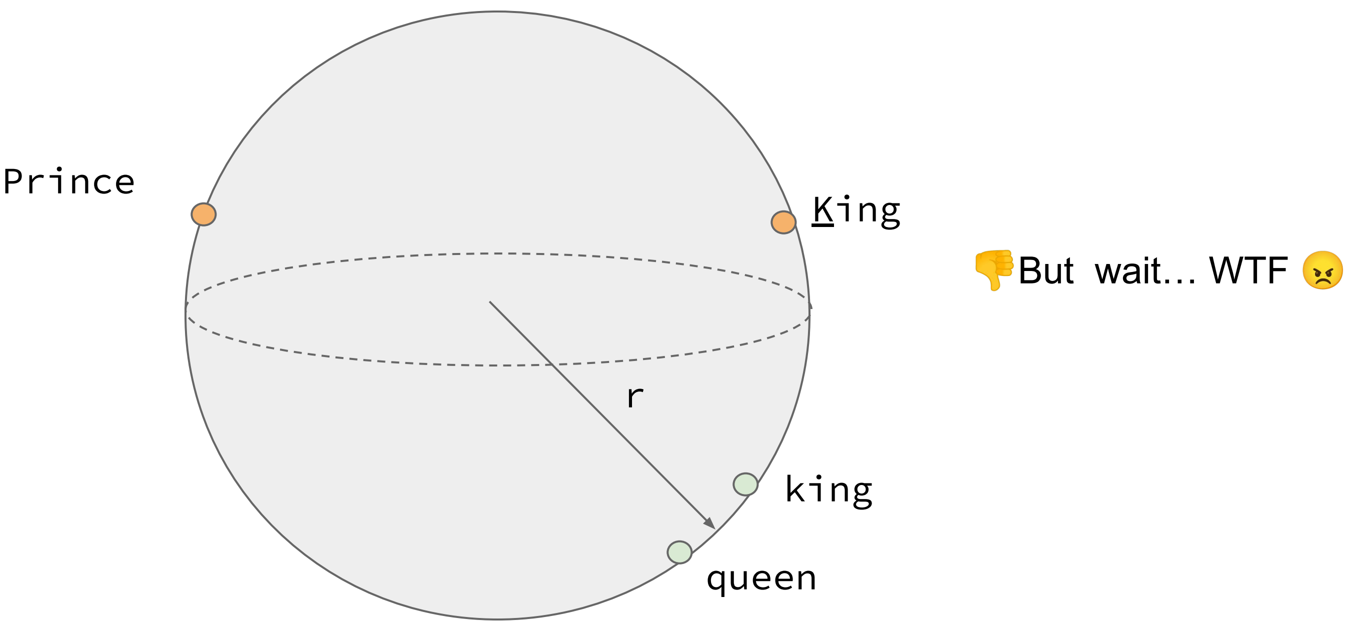
Source code:
https://github.com/softwaredoug/np-sims/blob/7e9b95580f36bbc4c57545e66c27d5802041949e/test/test_partition.py#L128

Curse of Dimensionality



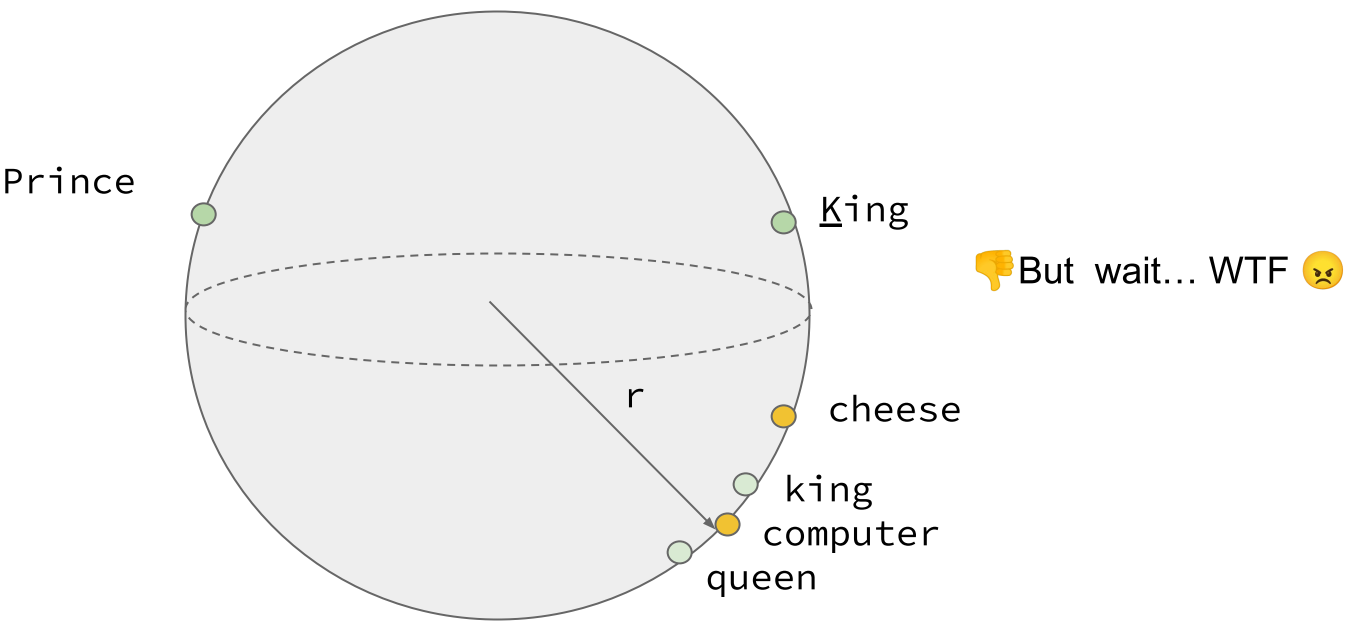
300 Dimensional Sphere

Curse of Dimensionality



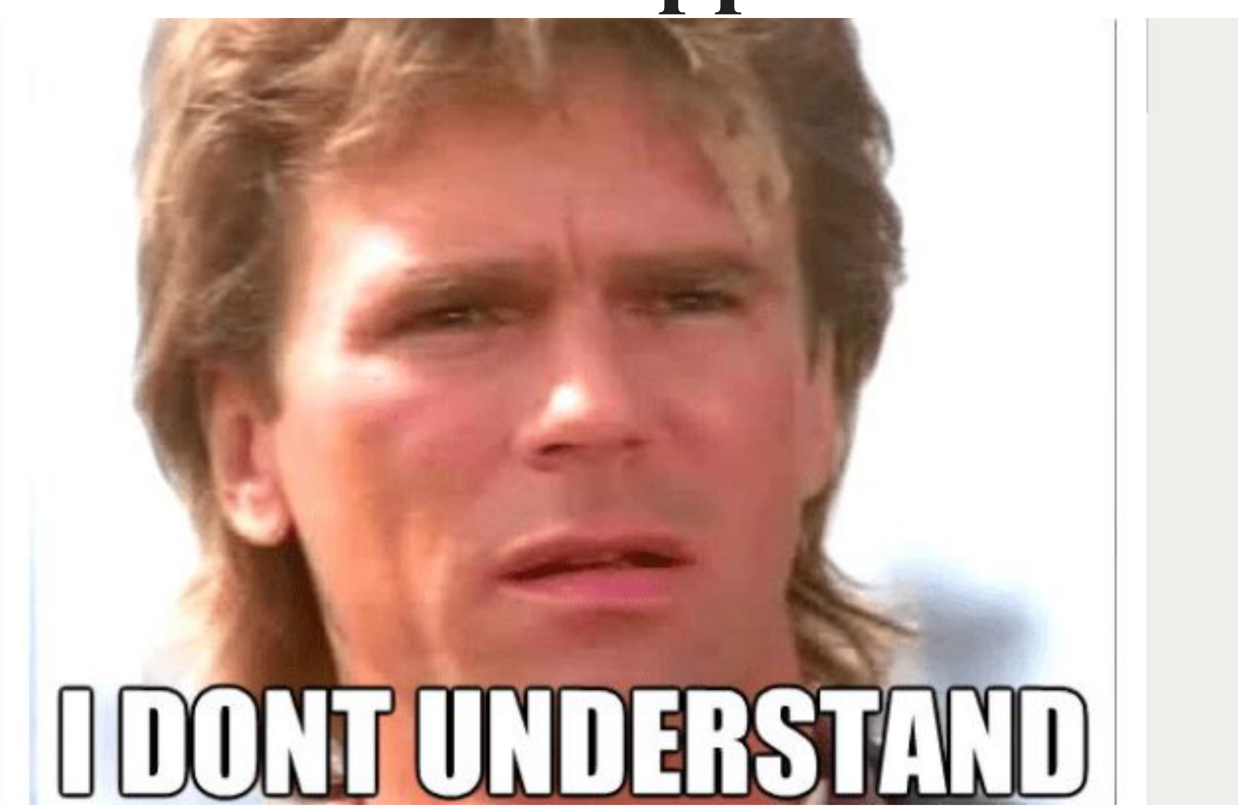
👎But wait… WTF 😠

Curse of Dimensionality

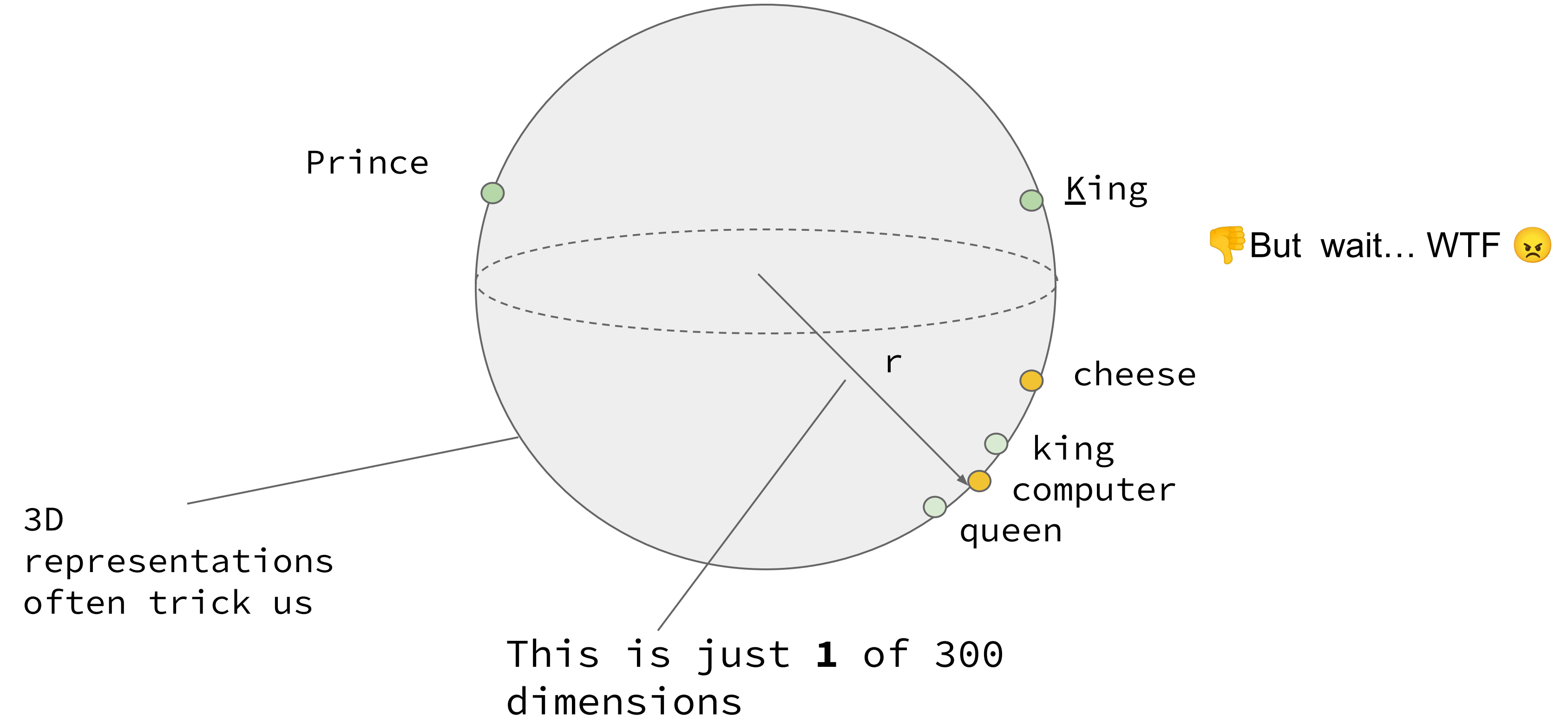


300 Dimensional Sphere

Like how does this happen



Curse of Dimensionality



Shared dot product tells us **so Little**

	0	…	298	299
king	r.king	?	…	?
queen	r.queen	?	…	?

299 ways it can be different!

https://softwaredoug.com/blog/2023/03/02/shared-dot-product

More Concretely

	lat	long	altitude	…
You	41.87	87.63	100 m	?
Bob	41.87	87.62	95 m	?

Neighbors, right?

https://softwaredoug.com/blog/2023/03/02/shared-dot-product

More Concretely

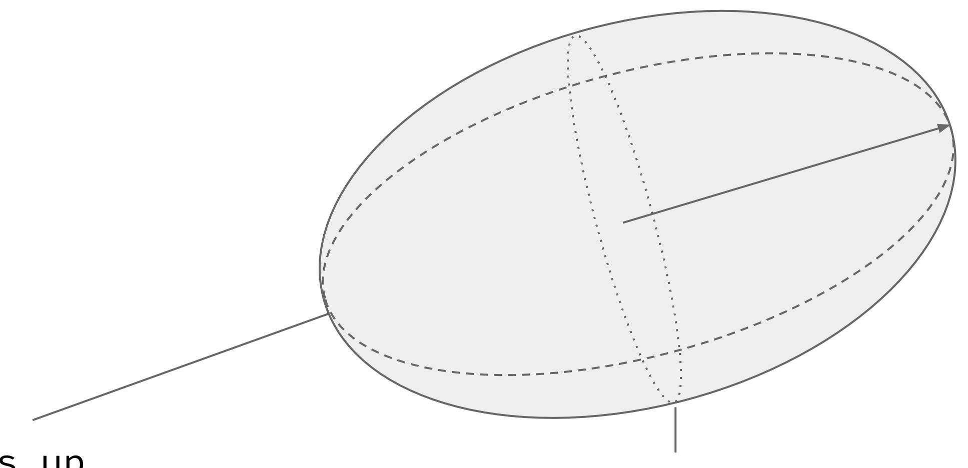
	lat	long	altitude	age	Birthplace
You	41.87	87.63	100 m	5	Chicago
Bob	41.87	87.62	95 m	95	Ukraine

Actually dramatically different just with these **2** dimensions

https://softwaredoug.com/blog/2023/03/02/shared-dot-product

Split efficiency

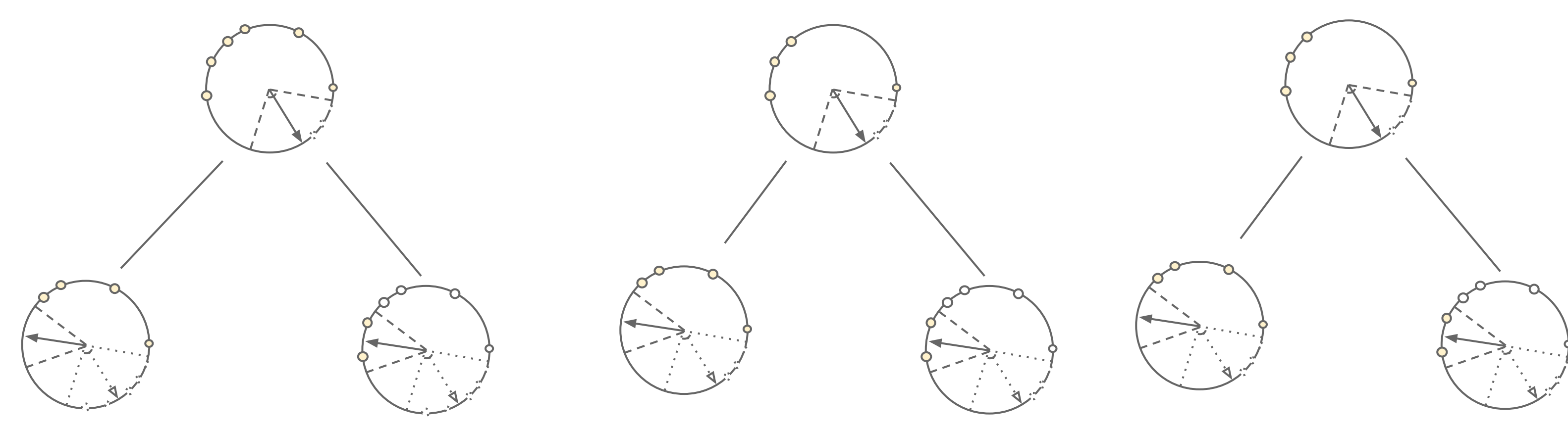
Can improve split efficiency by choosing PCAs



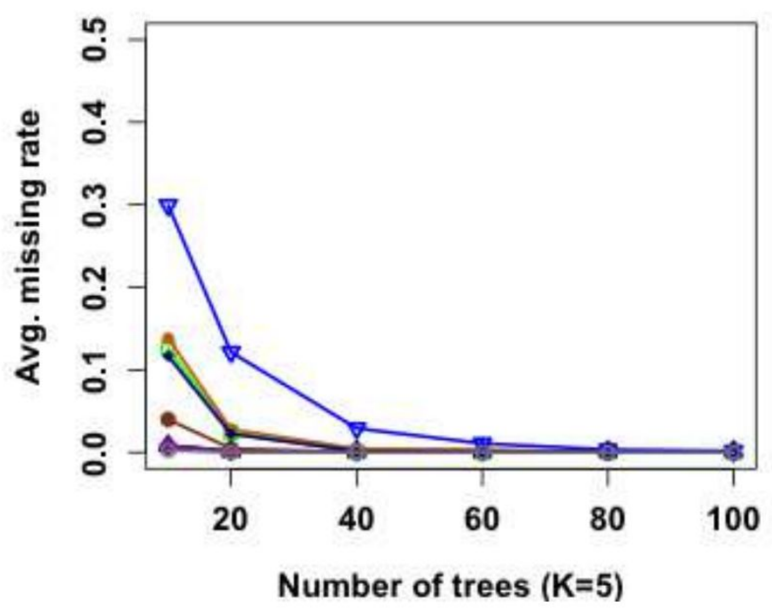
Splits up many neighbors

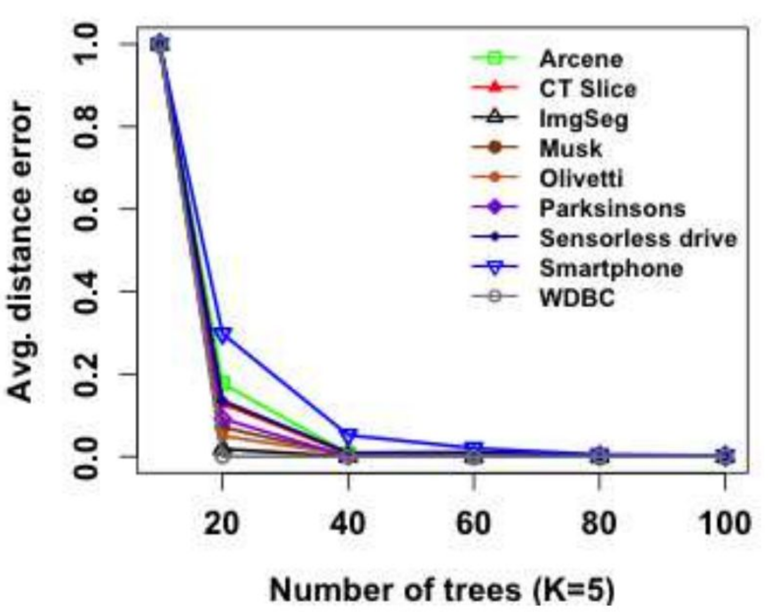
Splits up fewer neighbors

Many projections -> forest



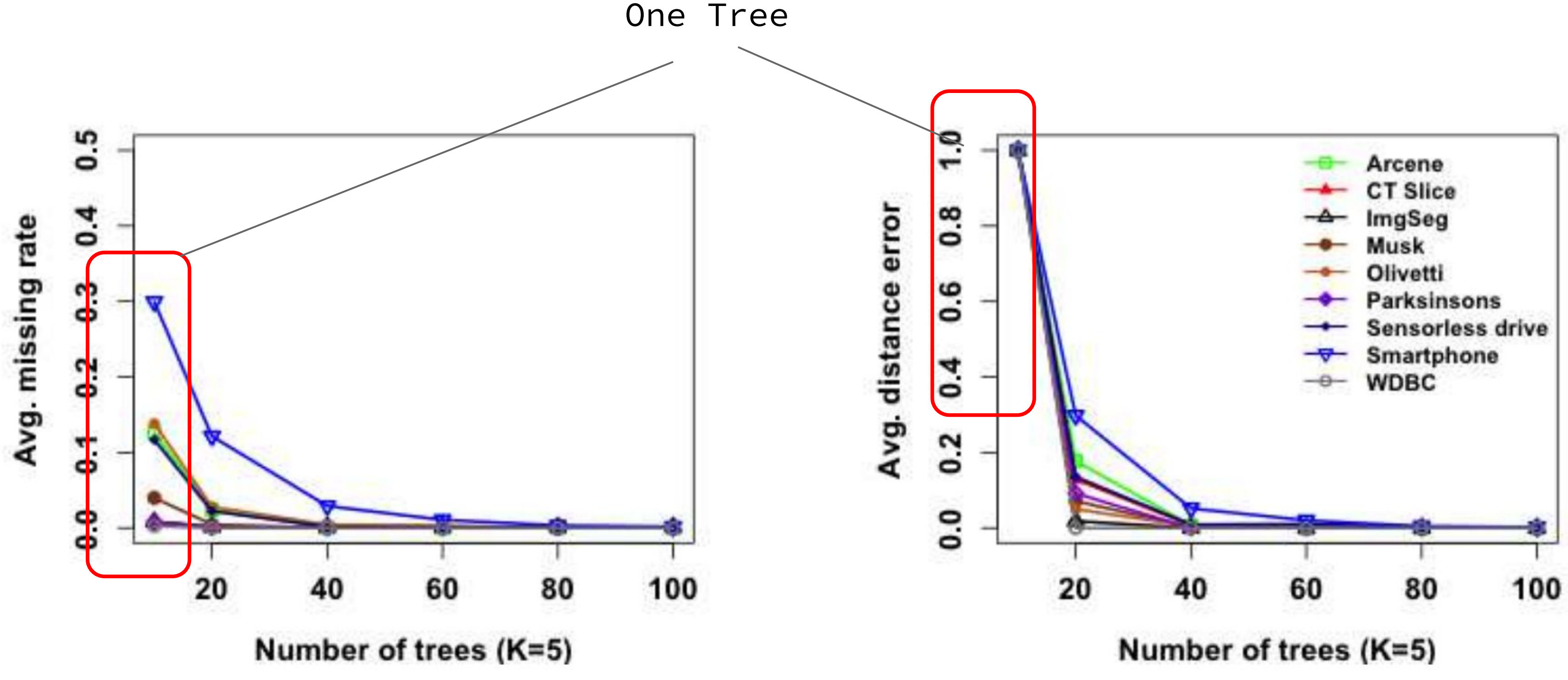
We can create an RP *forest*





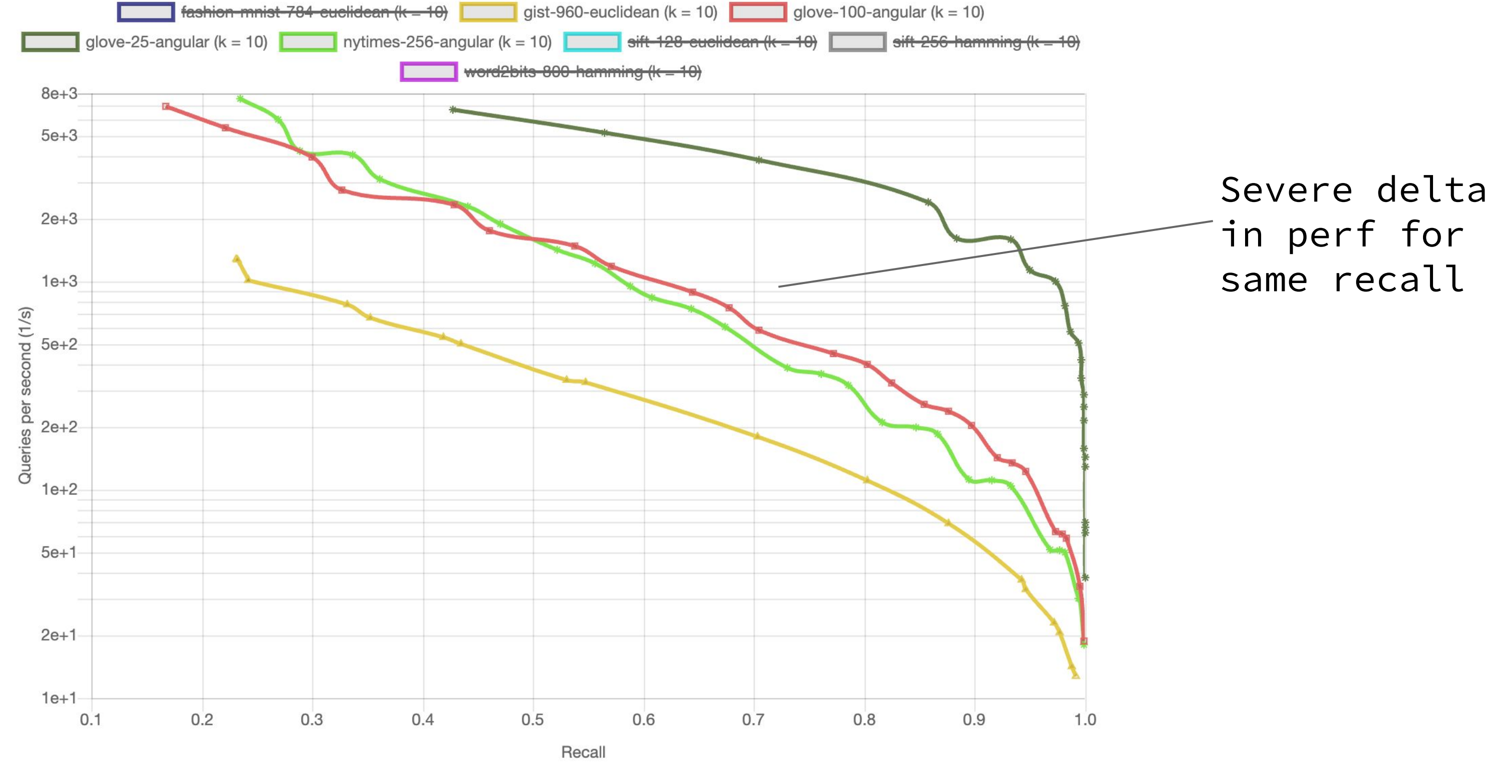
K-nearest Neighbor Search by Random Projection Forests - https://arxiv.org/pdf/1812.11689.pdf
Yan, Wang, Wang, Wang, Li

https://arxiv.org/pdf/1812.11689.pdf



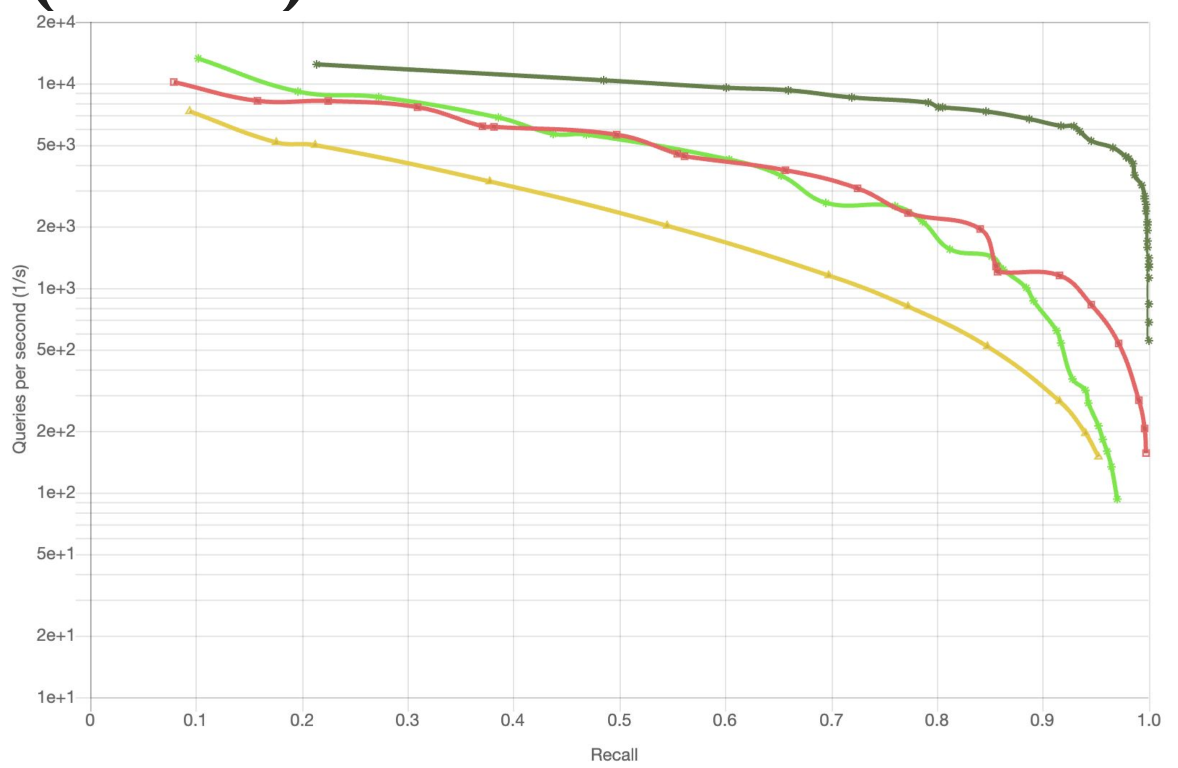
K-nearest Neighbor Search by Random Projection Forests - https://arxiv.org/pdf/1812.11689.pdf
Yan, Wang, Wang, Wang, Li

Annoy



ANNOY - https://github.com/spotify/annoy
Graph from - http://ann-benchmarks.com 30-Aug-2023

HNSW (Faiss)

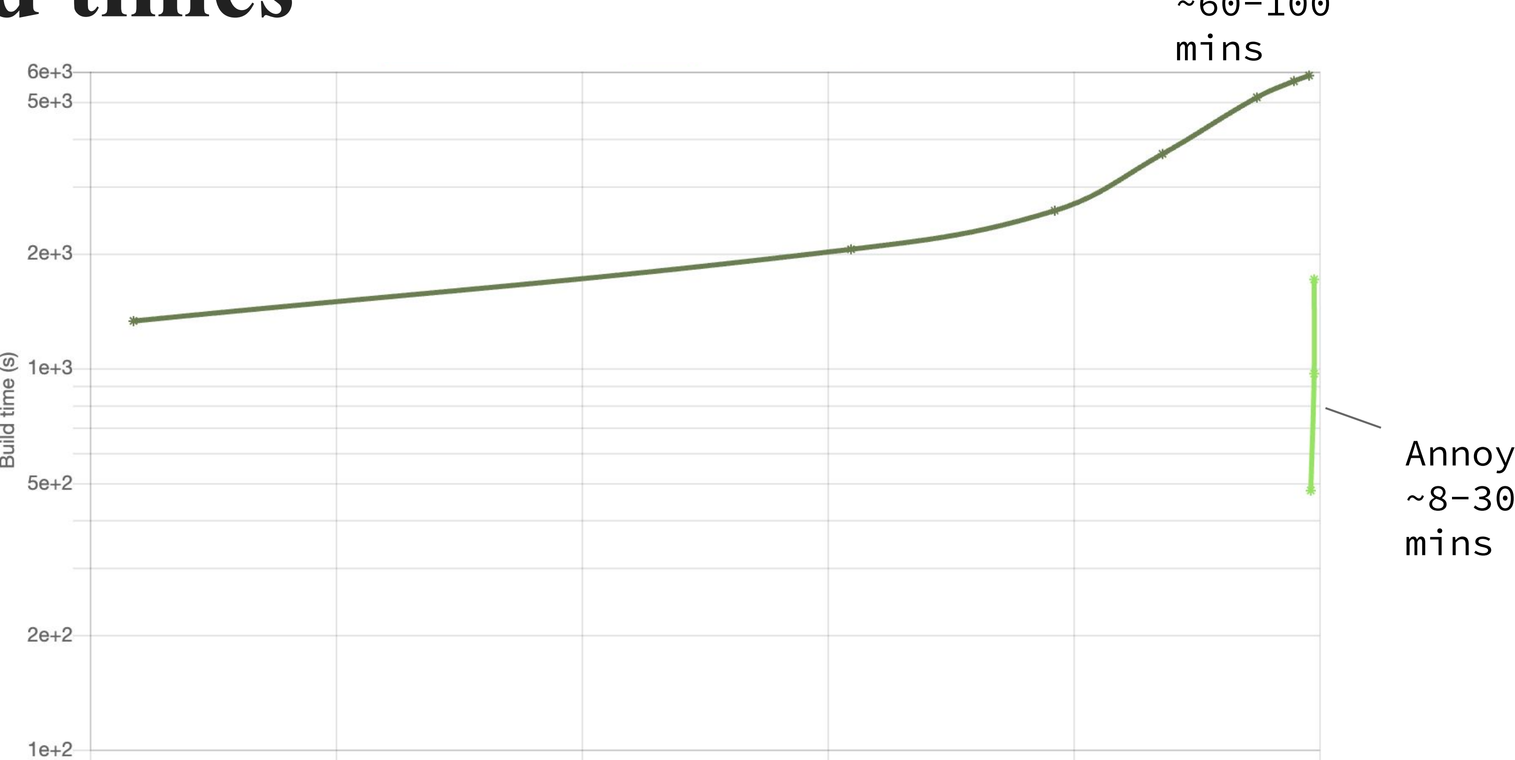


Holds a bit more steady

(also generally faster)

ANNOY - https://github.com/spotify/annoy
Graph from - http://ann-benchmarks.com 30-Aug-2023

Build times



Build times

6e+3
5e+3
2e+3
1e+3
5e+2
2e+2
1e+2

Build time (s)

0.75 0.80 0.85 0.90 1.00

Recall

Annoy
~8-30
mins

Update times &
stability vs
recall?

Trade-offs!

<u>Highly Parametric</u>
Pre-compute the teeniest structures (ie graphs)

- 😁 - Fixes curse of dimensionality...

  😭...by precomputing answer and being harder to update

<u>Highly Nonparametric</u>
Assume nothing about vector space

- 😭 - Prone to increasing curse of dimensionality problems as dims increase…

  😁 … but dumb caveman lawyer like me can maintain

More dimensions… more problems…

<u>Highly Parametric</u>
Pre-compute the teeniest structures (ie graphs)

<u>Highly Nonparametric</u>
Assume nothing about vector space

🚀 High recall
🚀 Low latency

❌ Easy to update (append!)
❌ RAM
❌ Disk
❌ Dumb as 💩 to merge / shard

High recall
Low latency

✅ Easy to update (append!)
? Very little RAM
✅ Dumb as 💩 to merge / shard

Choose right tool for job

<u>Highly Parametric</u>
Pre-compute the teeniest structures (ie graphs)

<u>Highly Nonparametric</u>
Assume nothing about vector space

Fine-grain retrieval

(ie top 10) to directly show user

Coarse-grain retrieval

(ie top 1000) to rerank

Choose right HNSW params

Highly Parametric
Pre-compute the teeniest structures (ie graphs)

Highly Nonparametric
Assume nothing about vector space

Fine-grain retrieval

Tuned to high-recall (more connections, gather more candidates)

Coarse-grain retrieval

Tune for performance (fewer connections, gather few candidates)

https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-knn-search.html

Which use-case?

Highly Parametric / Fine Grain

Pre-compute the teeniest structures (ie graphs)

Highly Nonparametric / Coarse Grain

Assume nothing about vector space

Fine-grain retrieval (ie top 10) to directly show user

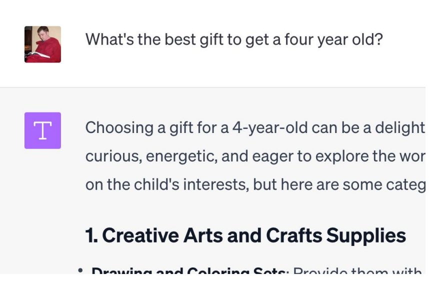
Coarse-grain retrieval (ie top 1000) to rerank w/ other factors?

Inspired by your browsing history

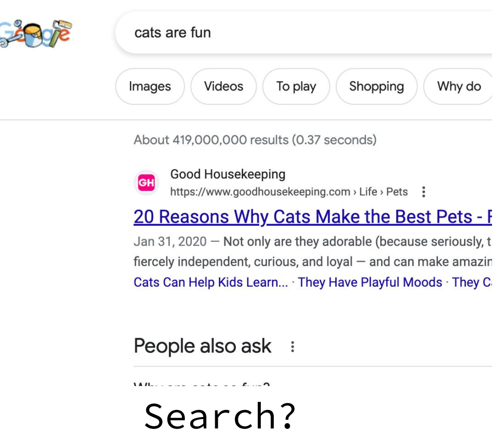
Amazon Essentials Men's Crewneck Fleece Sweatshirt
243
$15.00

Amazon Essentials Men's Slim-Fit Long-Sleeve Pocket T-Shirt
49
$10.00

Recos?

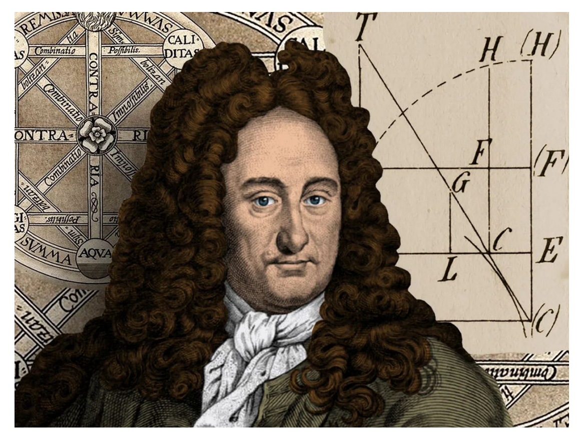


RAG?



Search?

Conclusion - who is right?



Leibniz

Newton

Relational space: space only has meaning in its relation to other objects

(ie graphs, HNSW, etc)

Absolute space: the “x,y,z” coordinates we’re used to

(ie space partitioning)

Other thought provoking talks

(Not nesc. related to vectors)

TDD Where Did It All Go Wrong (Ian Cooper)
https://www.youtube.com/watch?v=EZ05e7EMOLM
(what TDD actually means, and how we're doing it wrong)

Learning Learning to Rank (Sophie Watson)
https://www.youtube.com/watch?v=7teudGhdnqo
(Just a nice overview of LTR from ML Point of View)

The Only Unbreakable Law (Casey Muratori)
https://www.youtube.com/watch?v=5IUj1EZwpJY
(What Conway's Law actually says - how Conway's law transcends time & space on software projects)