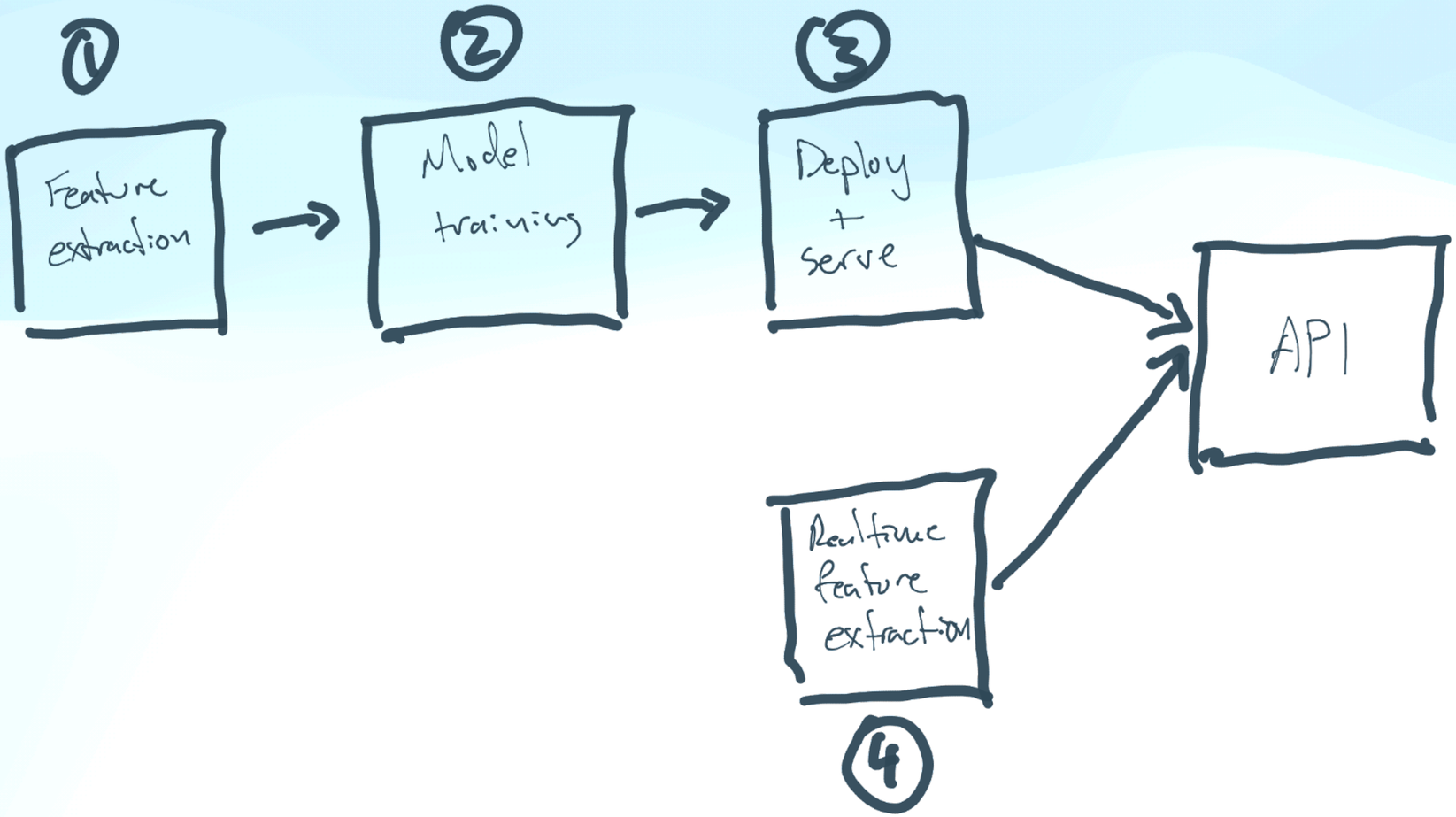
Vector search in Apache Cassandra

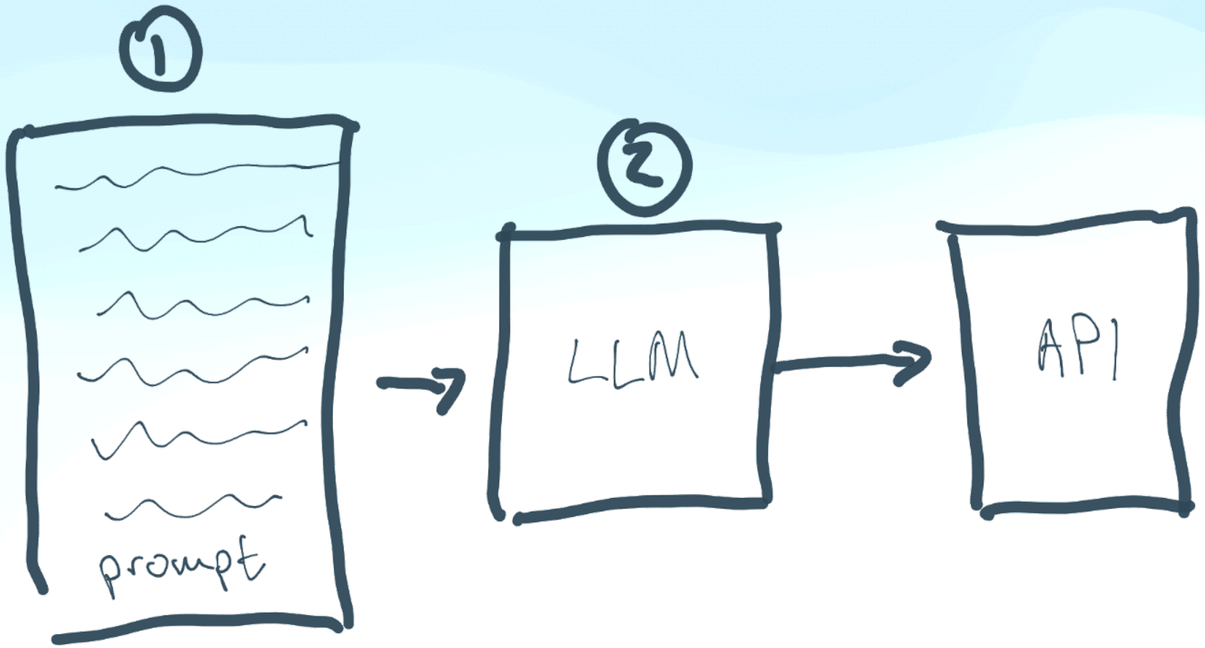
Jonathan Ellis

How LLMs + vector search make AI tractable for everyone

Old school AI



AI with LLMs



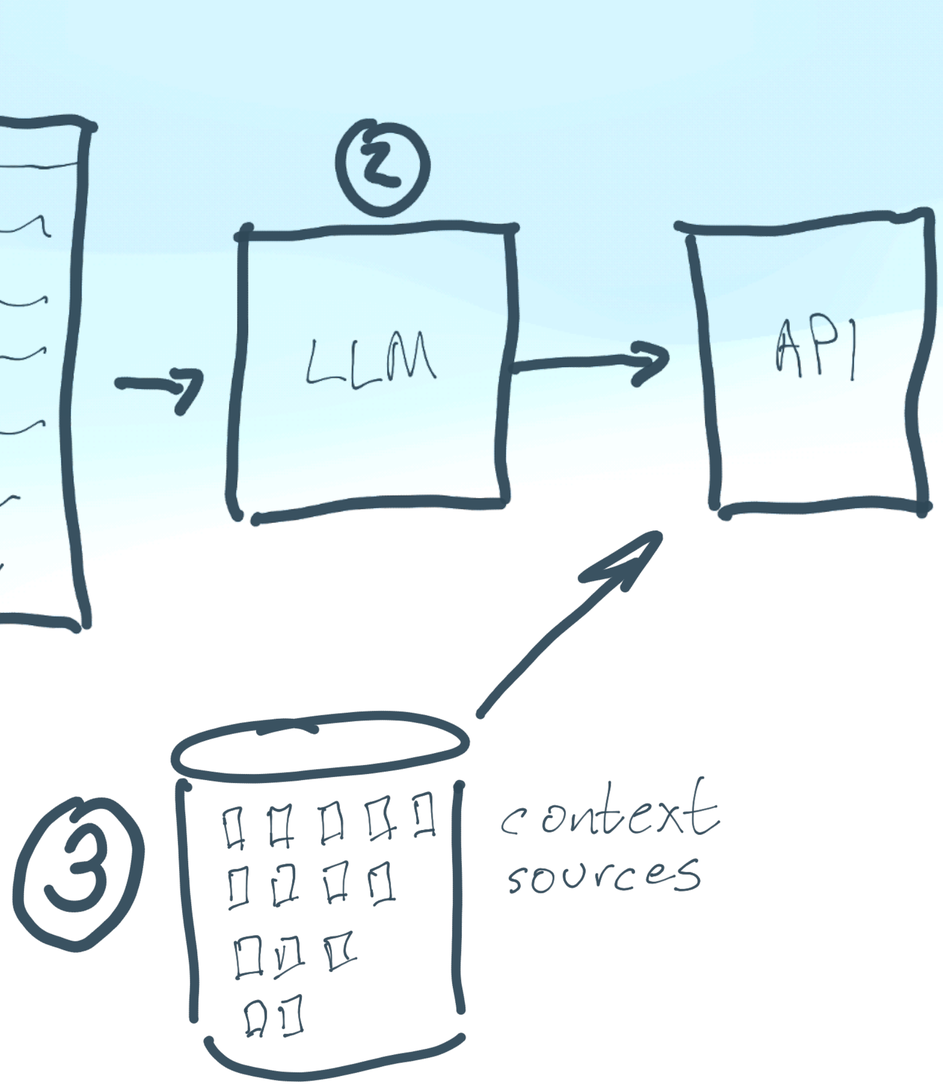
Cassandra's unified compaction strategy, introduced in version 4.0, aims to simplify the compaction process and improve overall database performance. It combines several compaction strategies into a single, more flexible approach. Here's a brief summary of Cassandra's unified compaction strategy:

1. **Background**:
   - In earlier versions of Cassandra, multiple compaction strategies existed, each with its own trade-offs and configuration parameters.
   - Unified compaction was introduced to streamline compaction and make it easier to manage.
2. **Single Compaction Strategy**:
   - Unified compaction merges multiple compaction strategies into a single, unified approach.
   - It combines Size-Tiered, Time Window, and Leveled compaction into a more versatile strategy.
3. **Dynamic Configuration**:
   - Unified compaction allows for dynamic configuration based on workload and performance requirements.
   - Users can adjust compaction parameters such as the min and max compaction throughput, compaction window size, and others to fine-tune performance.

Cassandra's Unified Compaction Strategy (UCS) is a new approach to compaction in Cassandra that combines aspects of both tiered and leveled compaction strategies. It introduces sharding, adaptability, and aims to optimize read and write performance based on user-defined parameters. Here's a summary of how UCS works:

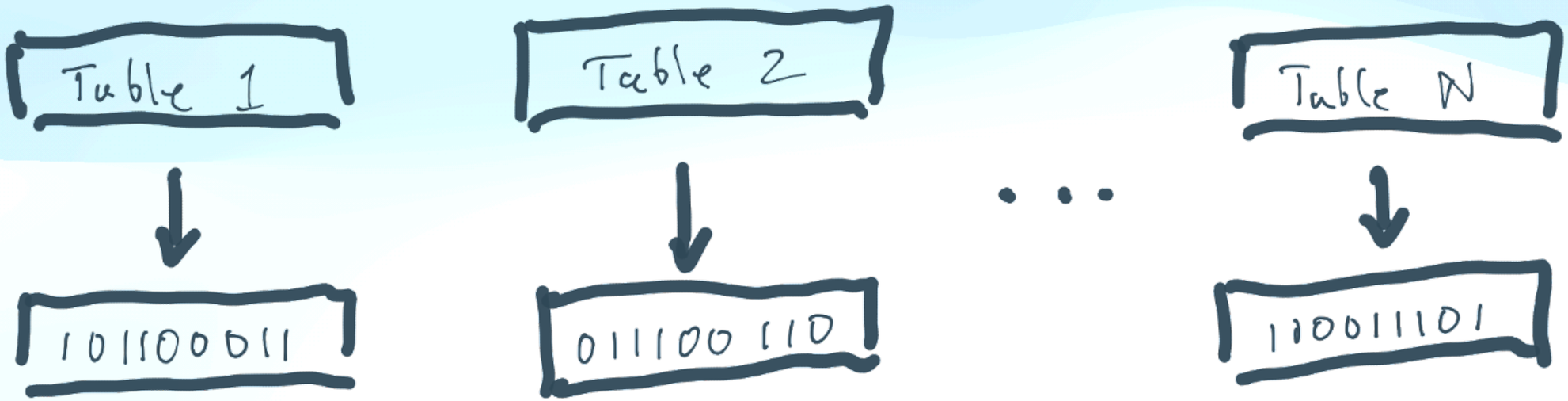
1. **Generalization of Compaction**: UCS generalizes tiered and leveled compaction strategies, treating them as the same by observing that both create levels based on the size of sstables and trigger compactions when a certain number of sstables are present on one level.
2. **Density-Based Levels**: UCS categorizes sstables into levels based on the logarithm of sstable density, with a user-defined fanout factor (f) as the base of the logarithm. Each level triggers a compaction when it contains a specified number (t) of overlapping sstables. The values of f and t determine the strategy's behavior, allowing users to choose between leveled and tiered compaction.
3. **Sharding**: UCS splits sstables at specific shard boundaries based on the density of an sstable. These shard boundaries increase with the density of the sstable, enabling concurrent compactions.
4. **Size-Based Levels**: If we ignore density and splitting, UCS groups sstables into levels based on their size relative to a memtable flush size. This allows UCS to determine the level of an sstable based on its size compared to a predefined flush size.

AI with LLMs + context



Finding the right context

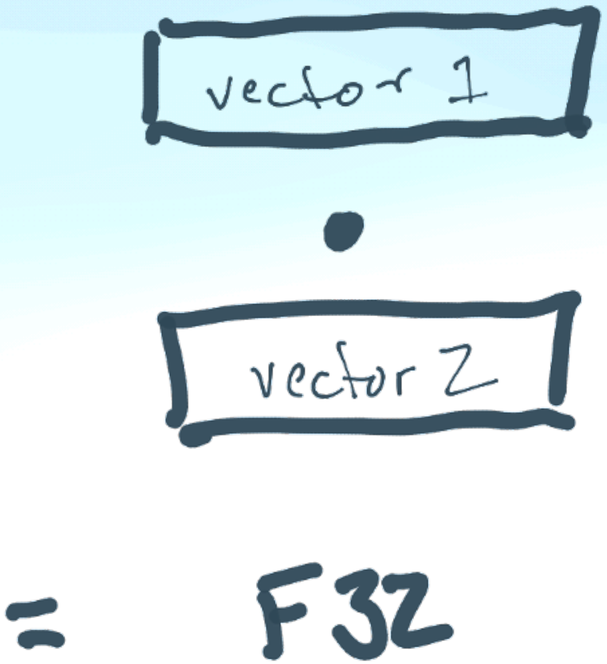
Embedding turns any text into vectors



Embeddings capture semantics

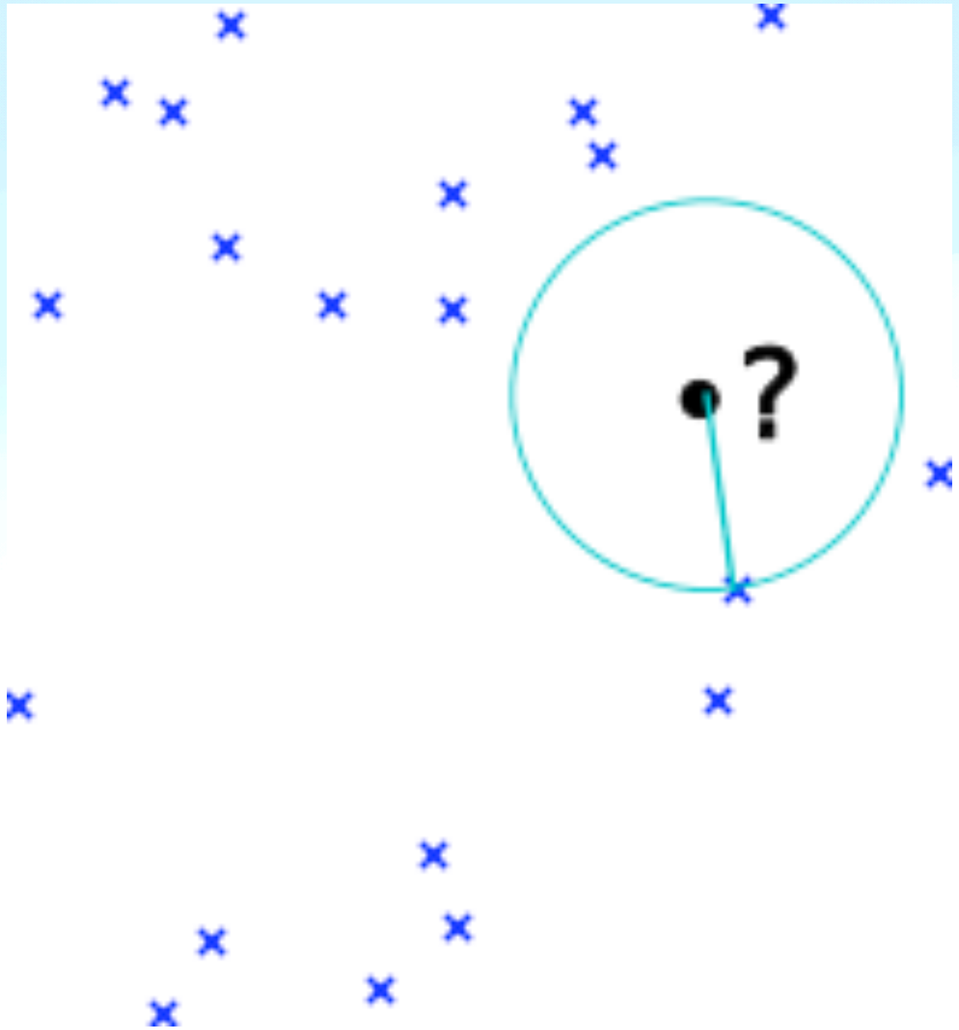
- “Man bites dog” vs “dog bites man”
- “Tourism numbers are collapsing” vs “Travel industry fears Covid-19 crisis will cause more companies to enter bankruptcy”
- “I need a new phone” vs “My old device is broken”

Vector similarity

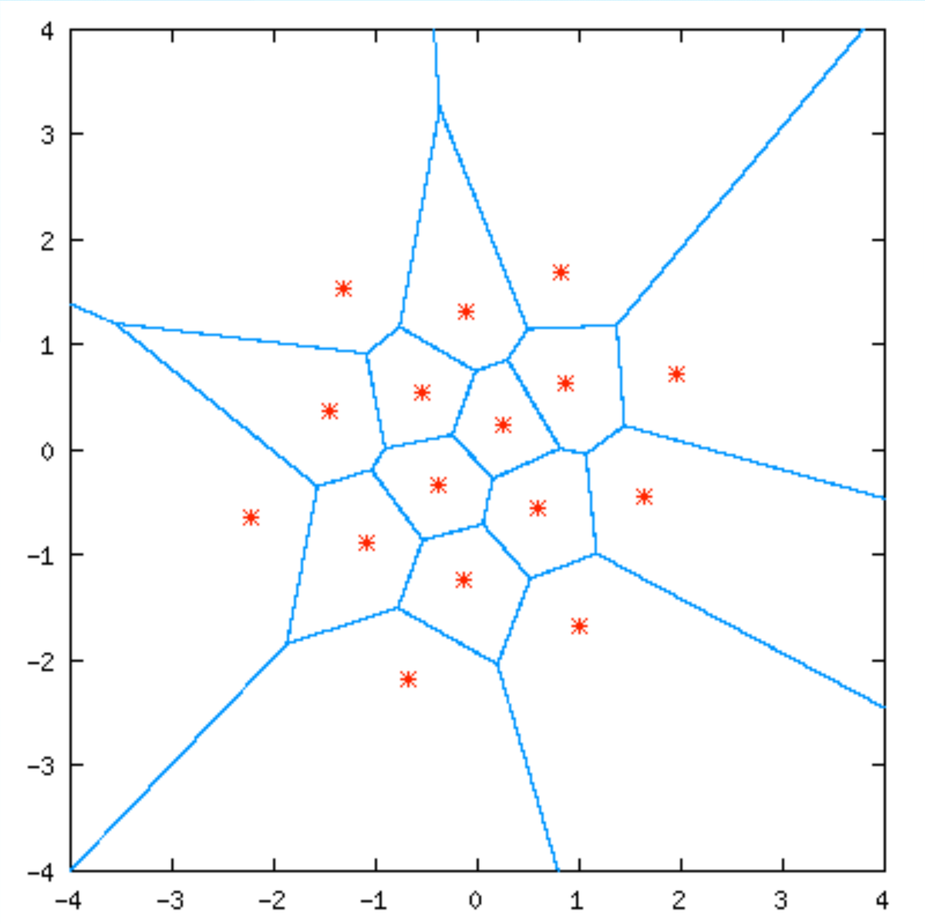


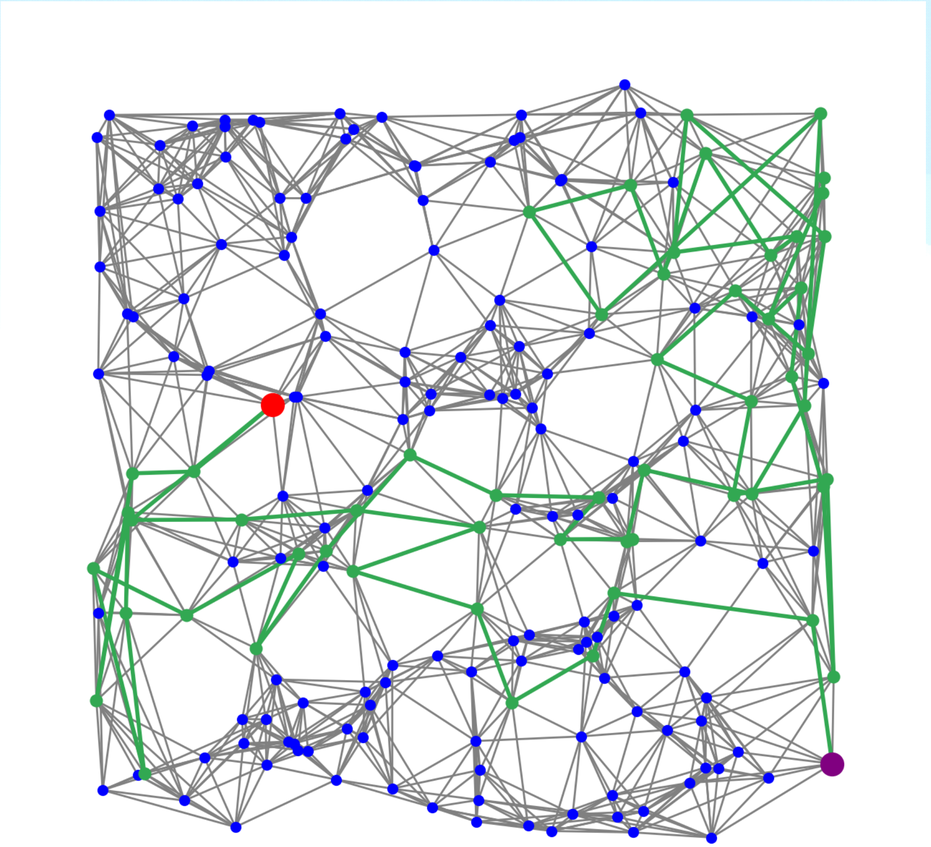
Vector search 101

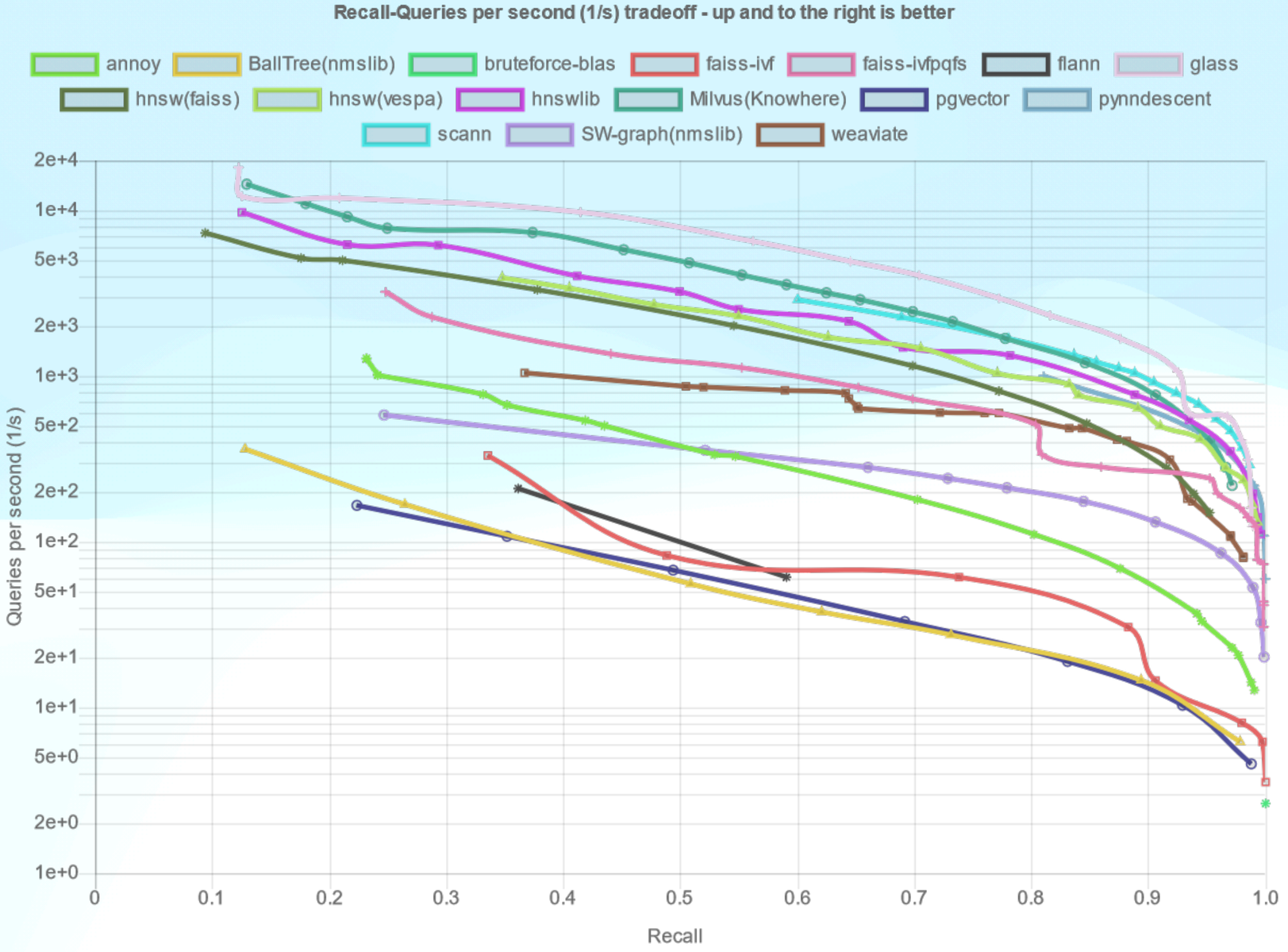
KNN

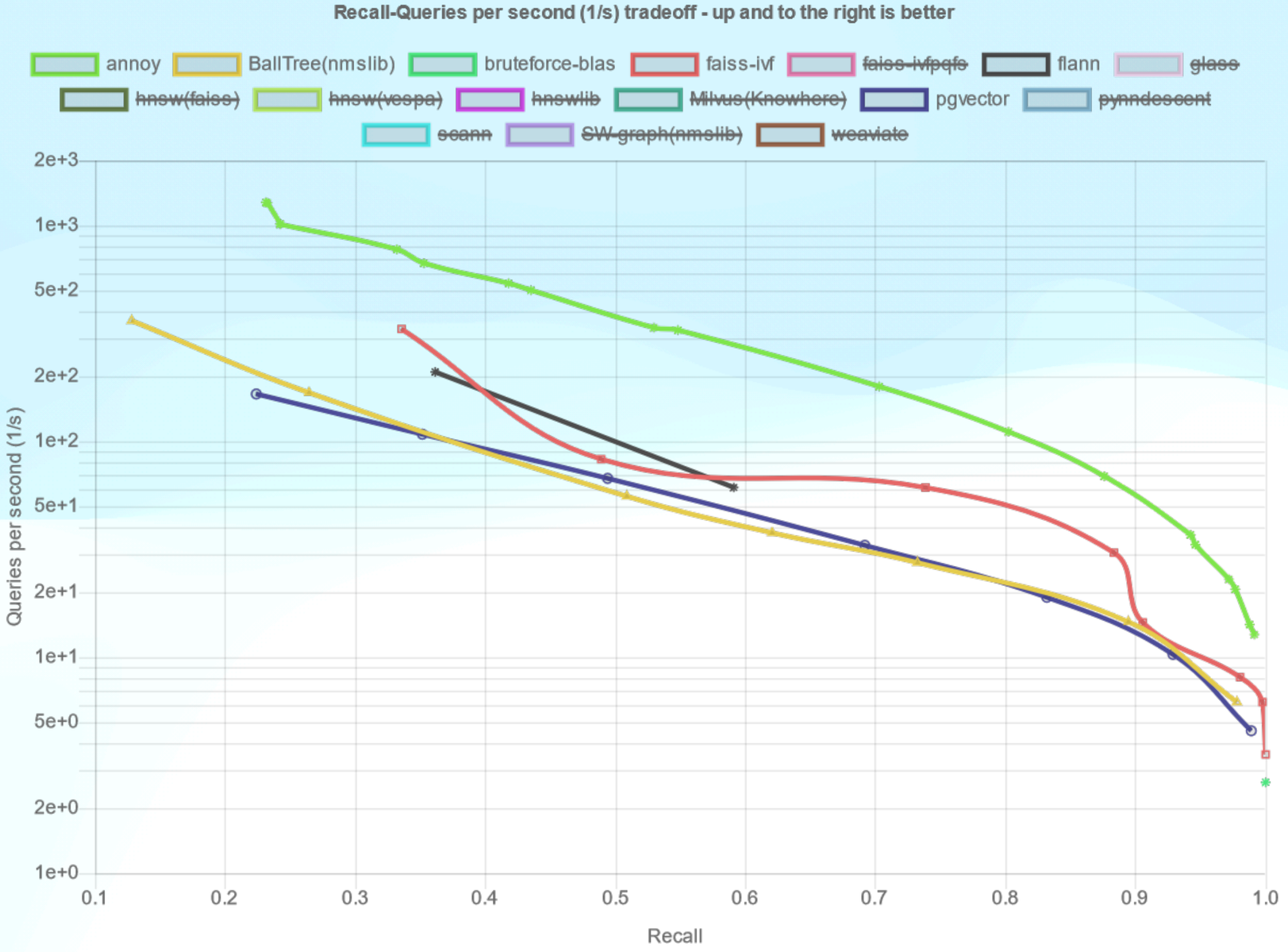


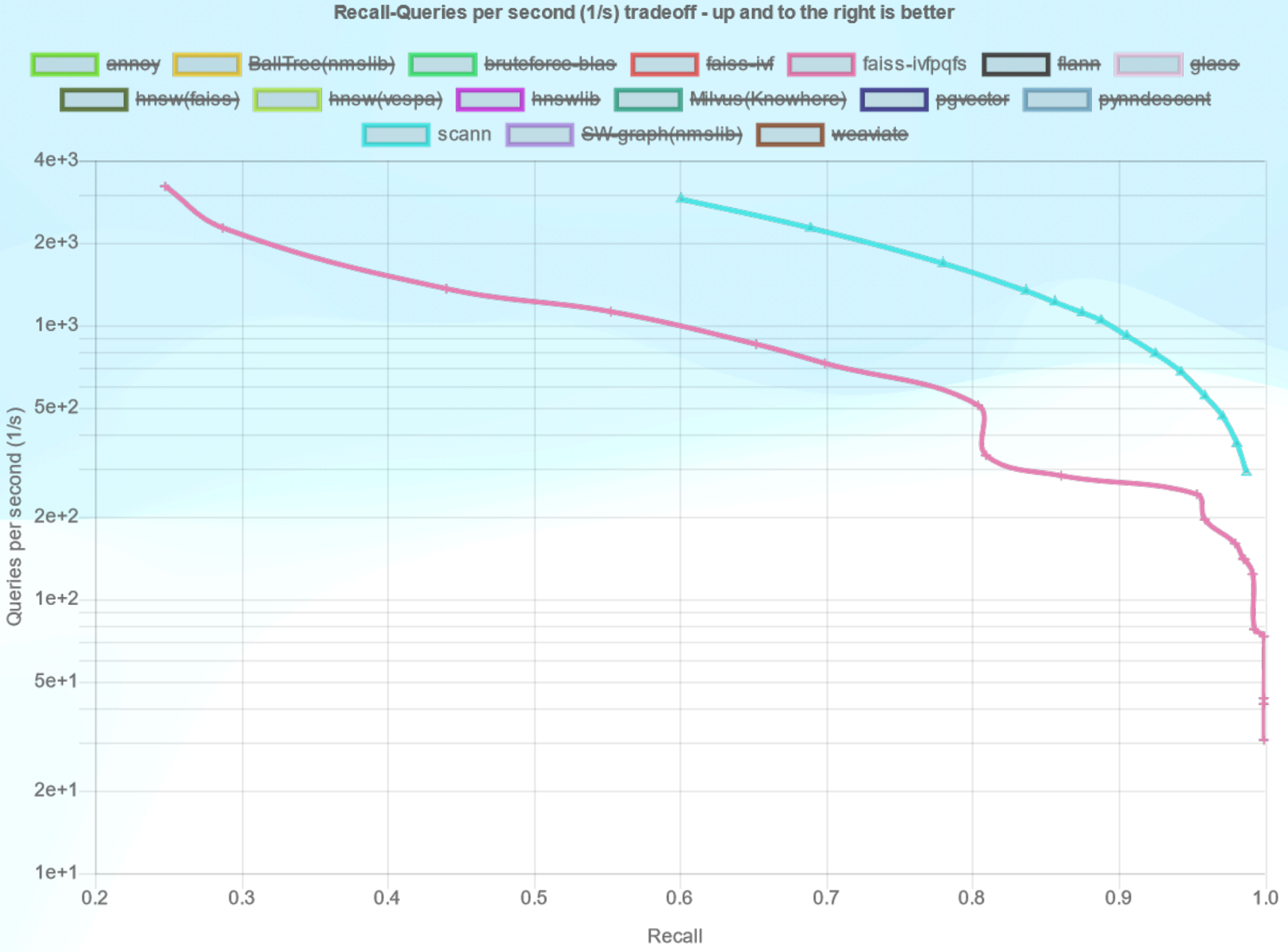
ANN

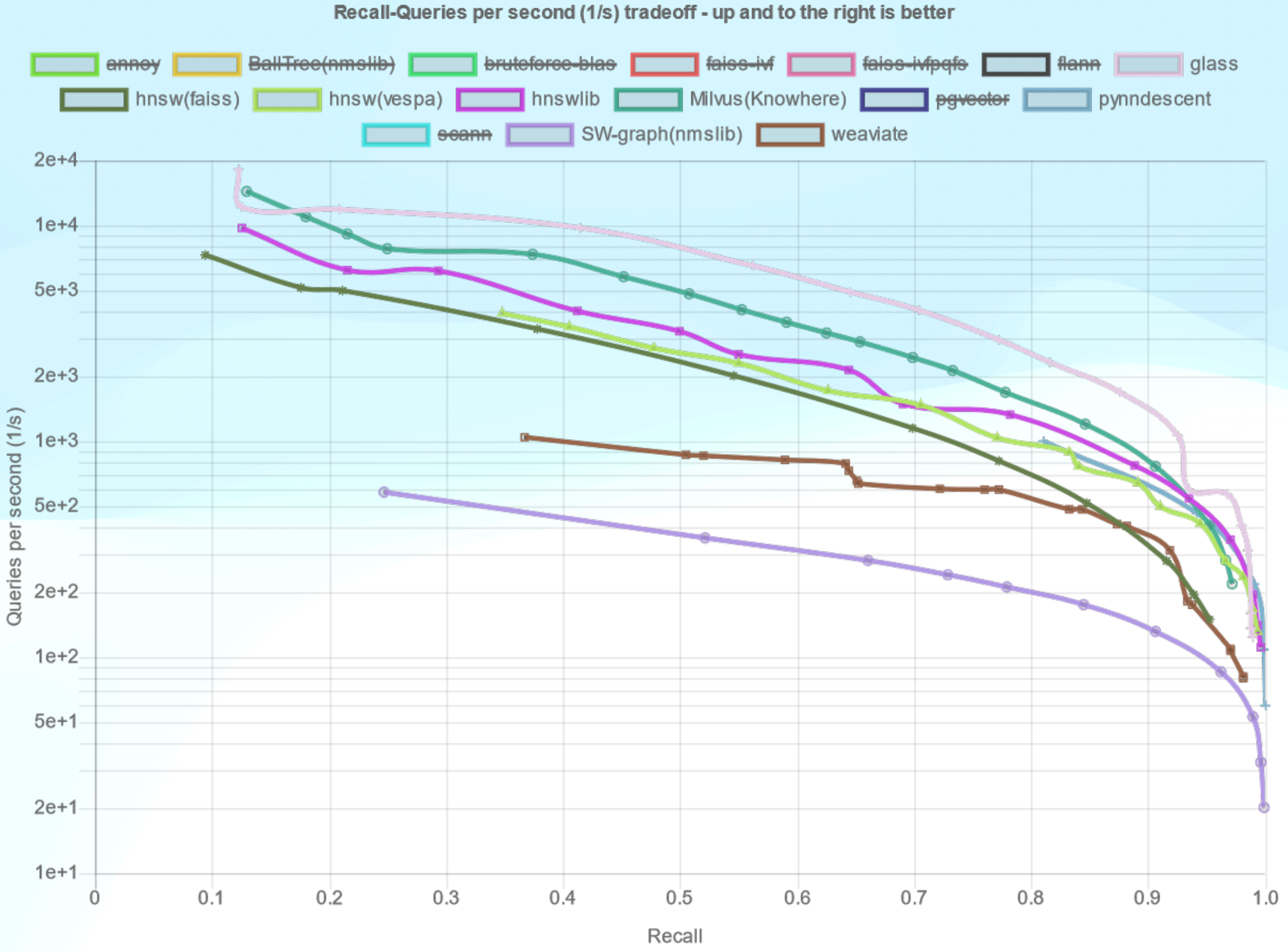





Recall-Queries per second (1/s) tradeoff - up and to the right is better
annoy
BallTree(nmslib)
bruteforce-blas
faiss-ivf
faiss-ivfpqfs
flann
glass
hnsw(faiss)
hnsw(vespa)
hnswlib
Milvus(Knowhere)
pgvector
pynndescent
scann
SW-graph(nmslib)
weaviate
Queries per second (1/s)
2e+4
1e+4
5e+3
2e+3
1e+3
5e+2
2e+2
1e+2
5e+1
2e+1
1e+1
5e+0
2e+0
1e+0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Recall


Recall-Queries per second (1/s) tradeoff - up and to the right is better
annoy
BallTree(nmslib)
bruteforce-blas
faiss-ivf
faiss-ivfpqfs
flann
glass
hnsw(faiss)
hnsw(vespa)
hnswlib
Milvus(Knowhere)
pgvector
pynndescent
scann
SW-graph(nmslib)
weaviate
Queries per second (1/s)
2e+3
1e+3
5e+2
2e+2
1e+2
5e+1
2e+1
1e+1
5e+0
2e+0
1e+0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Recall

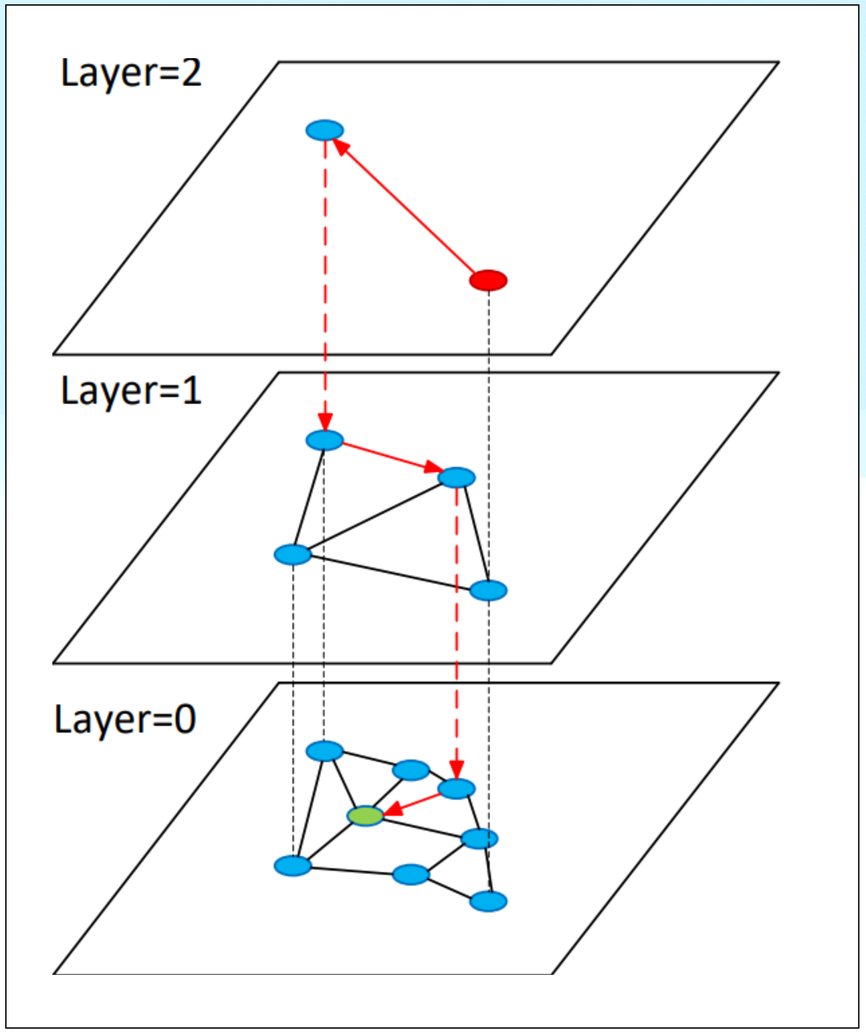

Recall-Queries per second (1/s) tradeoff - up and to the right is better
annoy
BallTree(nmslib)
bruteforce-blas
faiss-ivf
faiss-ivfpqfs
flann
glass
hnsw(faiss)
hnsw(vespa)
hnswlib
Milvus(Knowhere)
pgvector
pynndescent
scann
SW-graph(nmslib)
weaviate
Queries per second (1/s)
4e+3
2e+3
1e+3
5e+2
2e+2
1e+2
5e+1
2e+1
1e+1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Recall


Recall-Queries per second (1/s) tradeoff - up and to the right is better
annoy
BallTree(nmslib)
bruteforce-blas
faiss-ivf
faiss-ivfpqfs
flann
glass
hnsw(faiss)
hnsw(vespa)
hnswlib
Milvus(Knowhere)
pgvector
pynndescent
scann
SW-graph(nmslib)
weaviate
Queries per second (1/s)
2e+4
1e+4
5e+3
2e+3
1e+3
5e+2
2e+2
1e+2
5e+1
2e+1
1e+1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Recall

HNSW

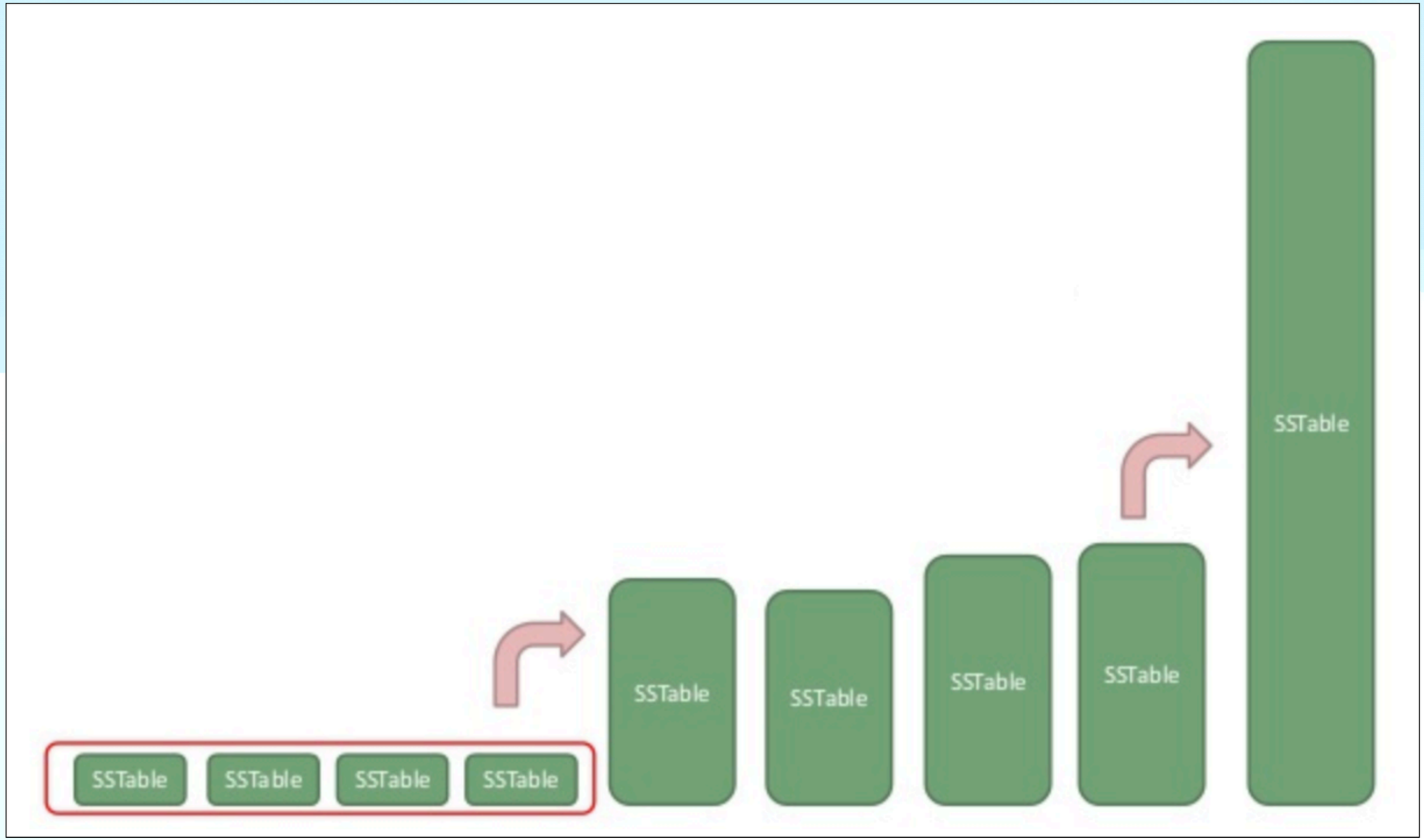
As seen in:

- Lucene
  - Elastic, Solo, OpenSearch
  - Mongo?
- Weaviate
- Qdrant
- Astra (June 2023)
- …



SAI (Storage Attached Indexes)

SSTable lifecycle



Global ANN everywhere

```
SELECT * FROM demo
ORDER BY
  embedding ANN OF ?
LIMIT 10
```

Composes with partitioning

```
SELECT * FROM demo
WHERE partition_id = ?
ORDER BY
  embedding ANN OF ?
LIMIT 100
```

Composes with other SAI indexes

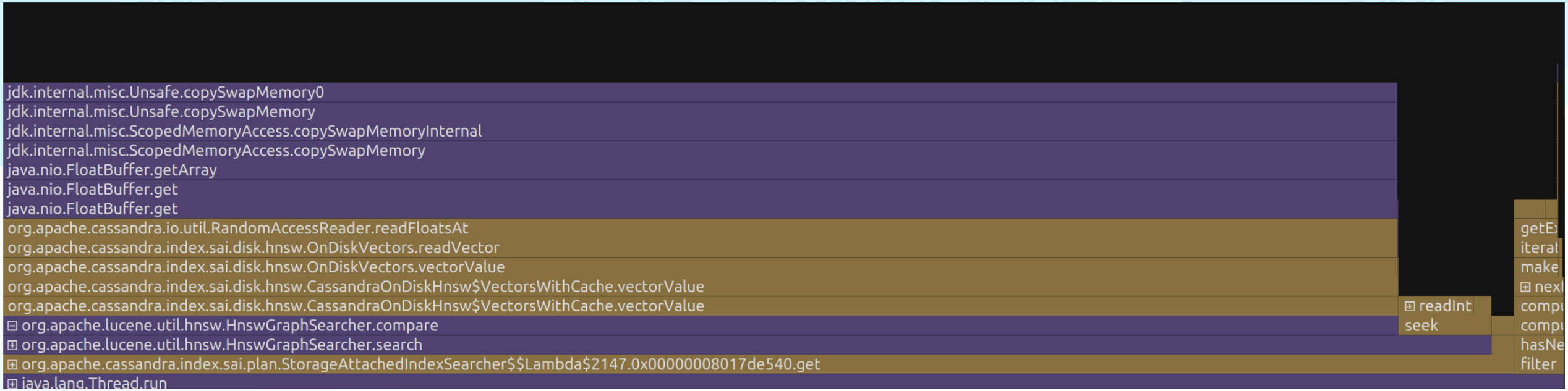
```
SELECT * FROM demo
WHERE (c1 = ? AND c2 = ?)
   OR c3 = ?
ORDER BY
  embedding ANN OF ?
LIMIT 5
```

Executing ANN queries

1. Apply SAI predicates across all stables
2. Query each ANN index for top K relevant rowIDs (or brute force it)
3. Merge by score
4. Reorder by primary key
5. Send results to coordinator

HNSW in production

100M Wikipedia vectors



Current SOTA

DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

Suhas Jayaram Subramanya* Carnegie Mellon University suhas@cmu.edu

Devvrit* University of Texas at Austin devvrit.03@gmail.com

Rohan Kadekodi* University of Texas at Austin rak@cs.texas.edu

Ravishankar Krishaswamy Microsoft Research India rakri@microsoft.com

Harsha Vardhan Simhadri Microsoft Research India harshasi@microsoft.com

Abstract

Current state-of-the-art approximate nearest neighbor search (ANNS) algorithms generate indices that must be stored in main memory for fast high-recall search. This makes them expensive and limits the size of the dataset. We present a new graph-based indexing and search system called DiskANN that can index, store, and search a billion point database on a single workstation with just 64GB RAM and an inexpensive solid-state drive (SSD). Contrary to current wisdom, we demonstrate that the SSD-based indices built by DiskANN can meet all three desiderata for large-scale ANNS: high-recall, low query latency and high density (points indexed per node). On the billion point SIFT1B *bigann* dataset, DiskANN serves $>$ 5000 queries a second with $<$ 3ms mean latency and 95%+ 1-recall@1 on a 16 core machine, where state-of-the-art billion-point ANNS algorithms with similar memory footprint like FAISS [18] and IVFOADC+G+P [8] plateau at around 50% 1-recall@1. Alternately, in the high recall regime, DiskANN can

SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search

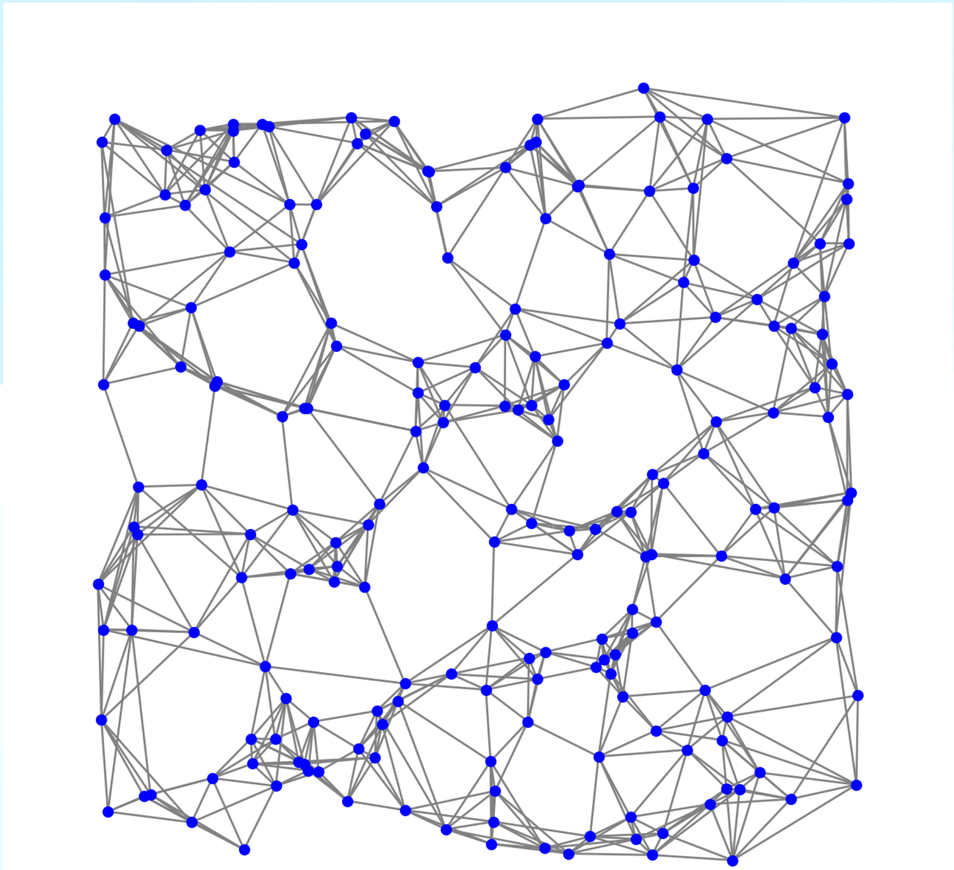
Qi Chen[1, *] **Bing Zhao**[1, 2, †] **Haidong Wang**[1] **Mingqin Li**[1] **Chuanjie Liu**[1, 3, †]
Zengzhong Li[1] **Mao Yang**[1] **Jingdong Wang**[1, 4, *, †]
[1]Microsoft [2]Peking University [3]Tencent [4]Baidu
[1]{cheqi, haidwa, mingqli, jasol, maoyang}@microsoft.com
[2]its.bingzhao@pku.edu.cn [3]liu.chuanjie@outlook.com [4]wangjingdong@outlook.com

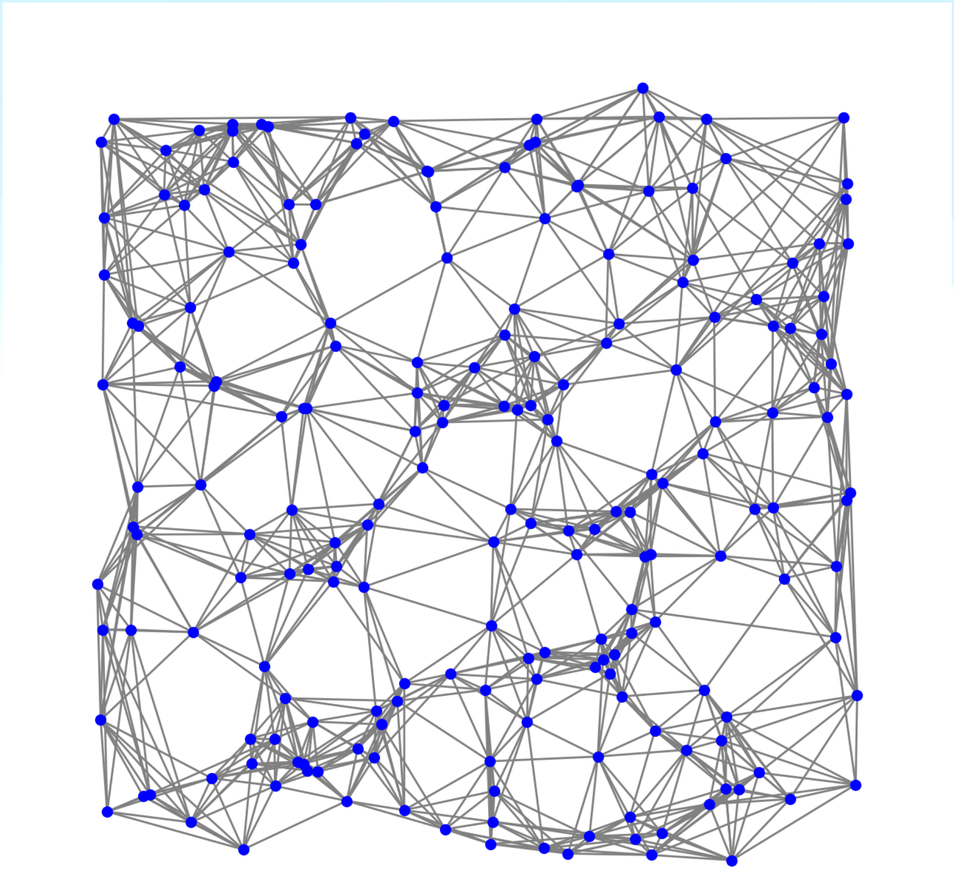
Abstract

The in-memory algorithms for approximate nearest neighbor search (ANNS) have achieved great success for fast high-recall search, but are extremely expensive when handling very large scale database. Thus, there is an increasing request for the hybrid ANNS solutions with small memory and inexpensive solid-state drive (SSD). In this paper, we present a simple but efficient memory-disk hybrid indexing and search system, named SPANN, that follows the inverted index methodology. It stores the centroid points of the posting lists in the memory and the large posting lists in the disk. We guarantee both disk-access efficiency (low latency) and high recall by effectively reducing the disk-access number and retrieving high-quality posting lists. In the index-building stage, we adopt a hierarchical balanced clustering algorithm to balance the length of posting lists and augment the posting list by adding the points in the closure of the corresponding clusters. In the search stage, we use a query-aware scheme to dynamically prune the access of unnecessary posting lists. Experiment results demonstrate that SPANN is 2× faster than the state-of-the-art ANNS solution DiskANN to reach the same recall quality 90%

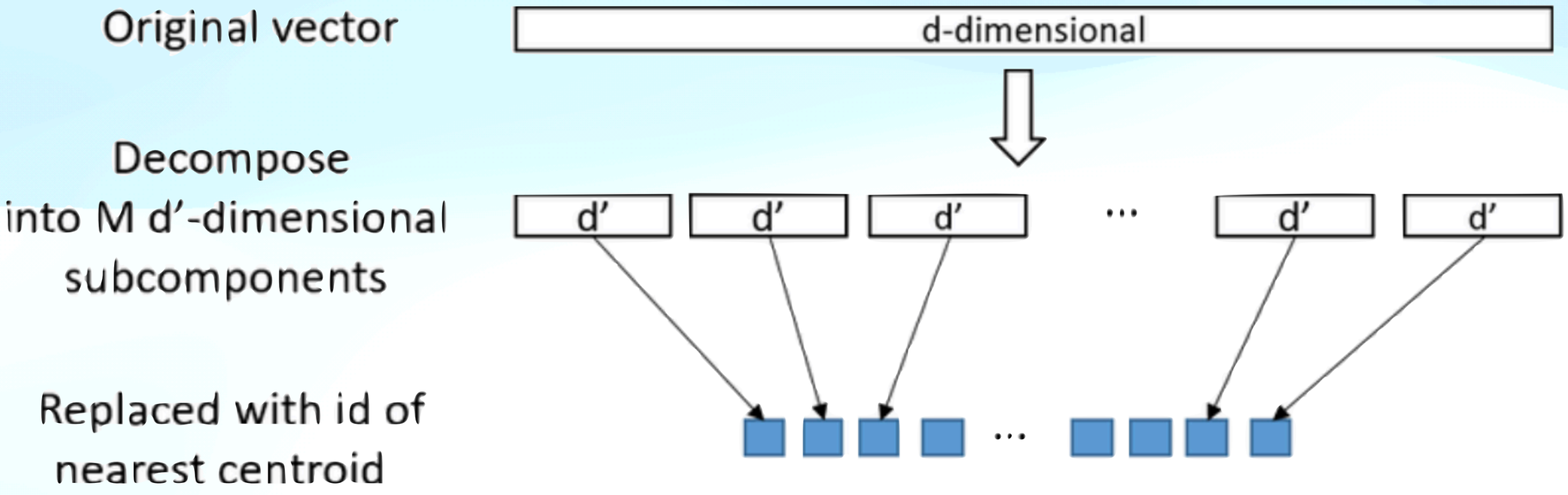
DiskANN

Vamana

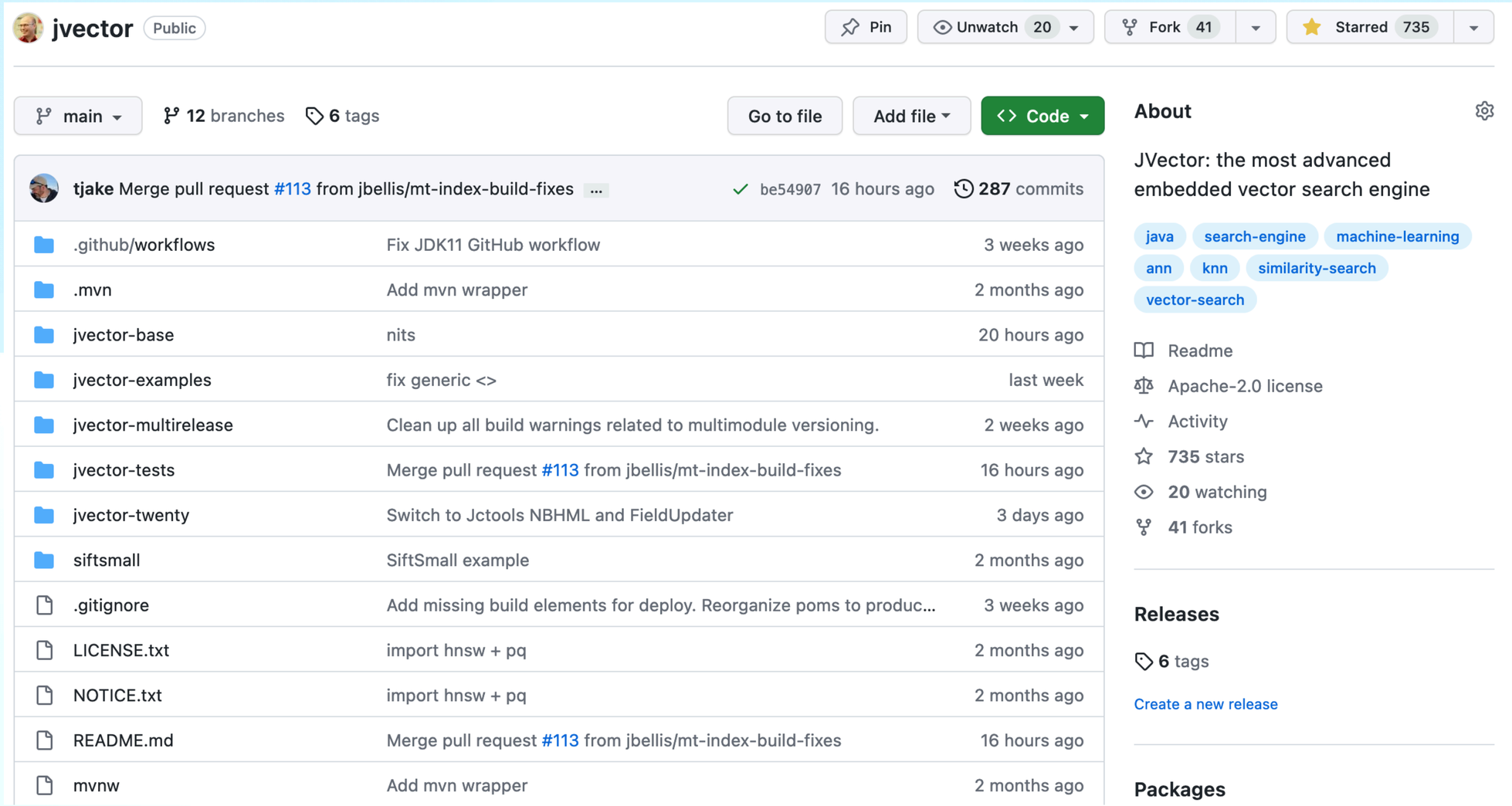




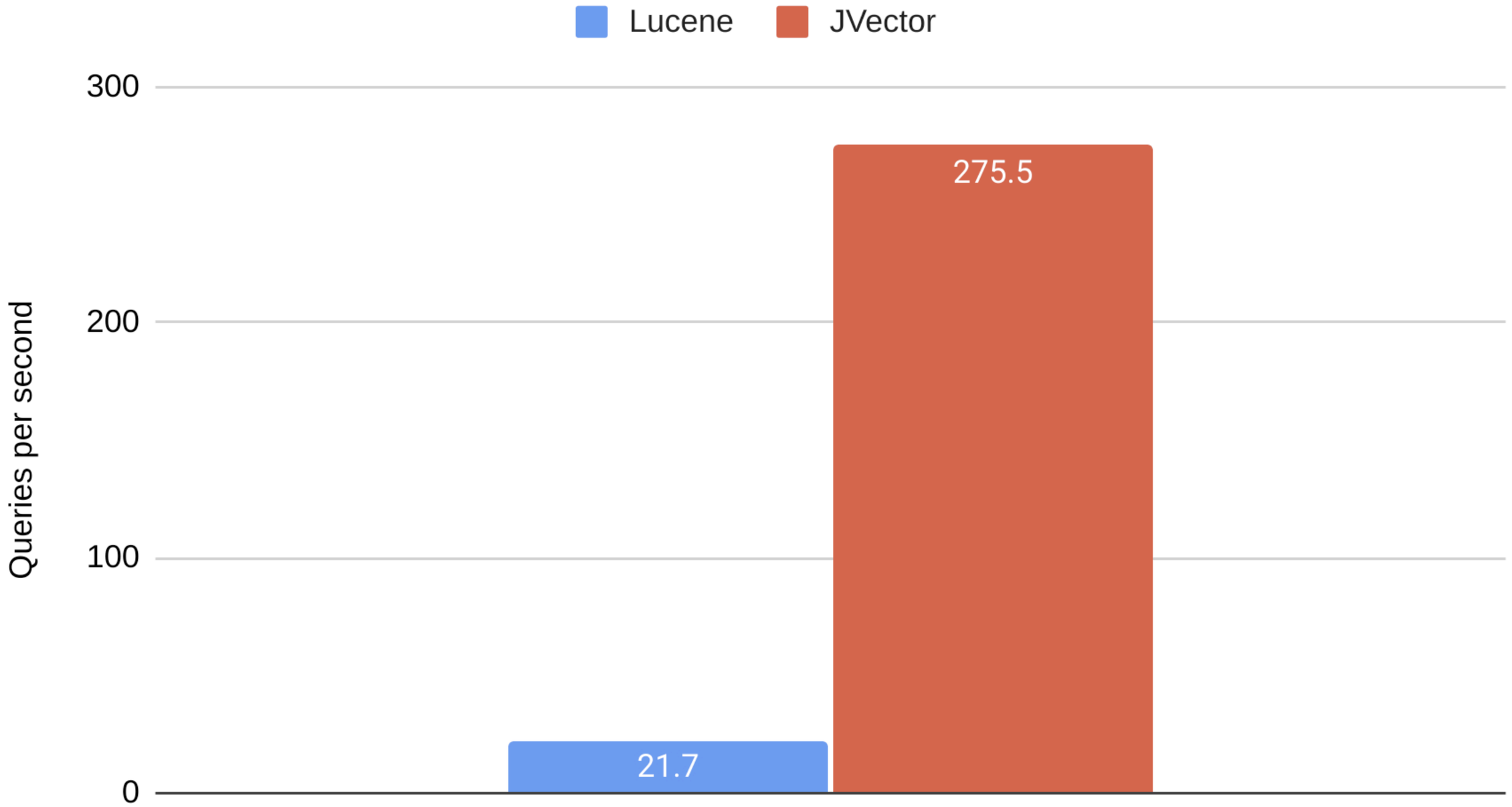
Product quantization



JVector


jvector
Public
Pin
Unwatch 20
Fork 41
Starred 735
main
12 branches
6 tags
Go to file
Add file
Code
tjake Merge pull request #113 from jbellis/mt-index-build-fixes ...
be54907 16 hours ago
287 commits
.github/workflows Fix JDK11 GitHub workflow 3 weeks ago
.mvn Add mvn wrapper 2 months ago
jvector-base nits 20 hours ago
jvector-examples fix generic <> last week
jvector-multirelease Clean up all build warnings related to multimodule versioning. 2 weeks ago
jvector-tests Merge pull request #113 from jbellis/mt-index-build-fixes 16 hours ago
jvector-twenty Switch to Jctools NBHML and FieldUpdater 3 days ago
siftsmall SiftSmall example 2 months ago
.gitignore Add missing build elements for deploy. Reorganize poms to produc... 3 weeks ago
LICENSE.txt import hnsw + pq 2 months ago
NOTICE.txt import hnsw + pq 2 months ago
README.md Merge pull request #113 from jbellis/mt-index-build-fixes 16 hours ago
mvnw Add mvn wrapper 2 months ago
About
JVector: the most advanced embedded vector search engine
java
search-engine
machine-learning
ann
knn
similarity-search
vector-search
Readme
Apache-2.0 license
Activity
735 stars
20 watching
41 forks
Releases
6 tags
Create a new release
Packages

QPS on Deep100M dataset (24GB Macbook)



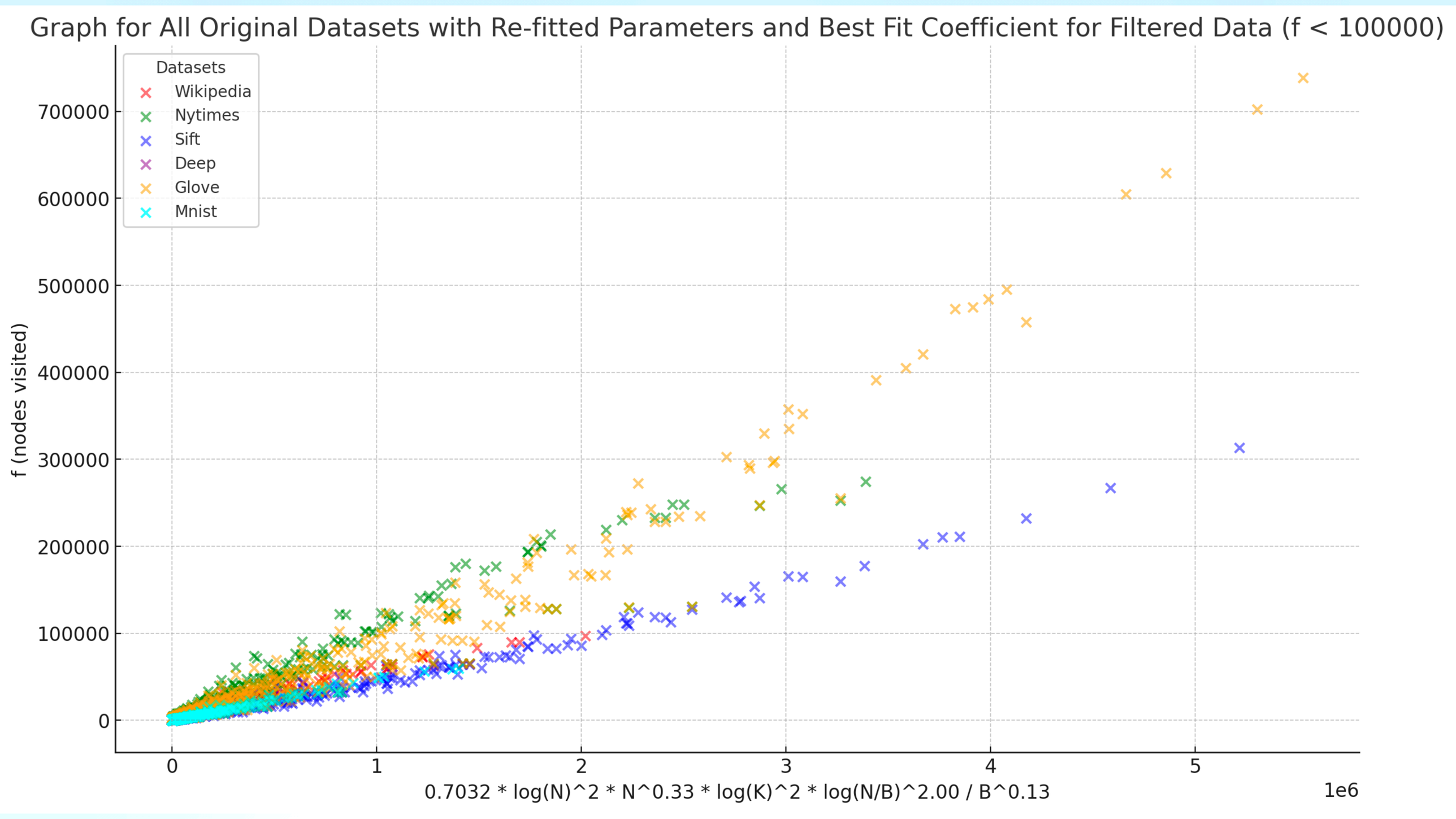
Closing thoughts

The fine print
(things that don’t work yet)

- Paging
- CL > 1
- … That’s it

Lessons learned

- Not correct to apply indexes to SSTables separately
- OpenAI embeddings are ridiculously overparmeterizehd
  - Use e5-v2
- Kmeans++ doesn’t need a lot of iterations
- Vamana needs room to add edges
- Surprisingly hard to predict how many nodes will be visited when searching for the top K of B items in a graph of N


Graph for All Original Datasets with Re-fitted Parameters and Best Fit Coefficient for Filtered Data (f < 100000)
Datasets
Wikipedia
Nytimes
Sift
Deep
Glove
Mnist
f (nodes visited)
700000
600000
500000
400000
300000
200000
100000
0
0
1
2
3
4
5
1e6
0.7032 * log(N)^2 * N^0.33 * log(K)^2 * log(N/B)^2.00 / B^0.13

Where’s the code?

- JVector: https://github.com/jbellis/jvector/
- Cassandra + vector search (today): https://github.com/datastax/cassandra/tree/vsearch
- Cassandra + vector search (RSN): https://issues.apache.org/jira/browse/CASSANDRA-18557