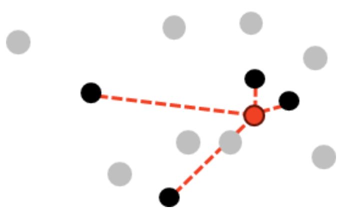
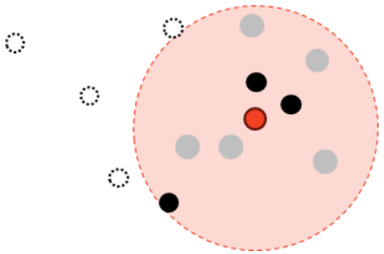
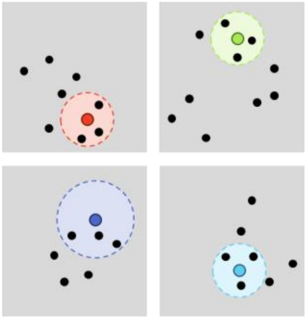
IV. ADVANCED INDEXING

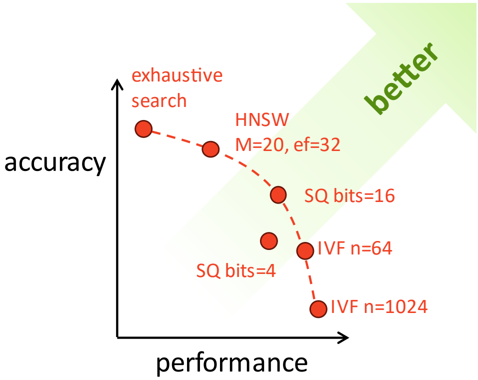
PREVIOUSLY, ON
TOPICS IN VECTOR DATABASES

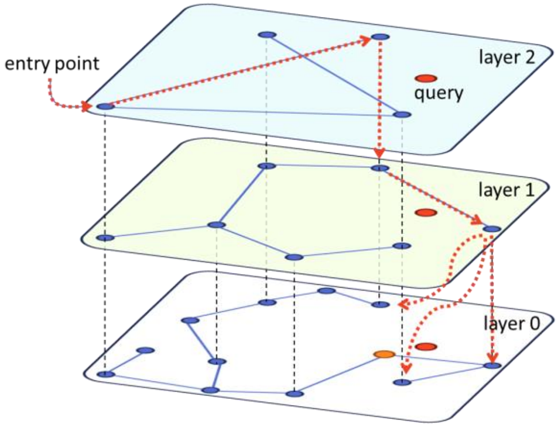
- Queries:
  - kNN
  - Filtered queries.
    - Prefilter / postfilter / single-stage
  - Multi-vector queries (and challenges)
  - Reranking
  - The need to index
- Index:
  - Tradeoffs and recall.
  - Flat index (for <100K vectors)
  - LSH (elegant but suboptimal)
  - IVF (cluster-based index)
  - HNSW (graph-based index)

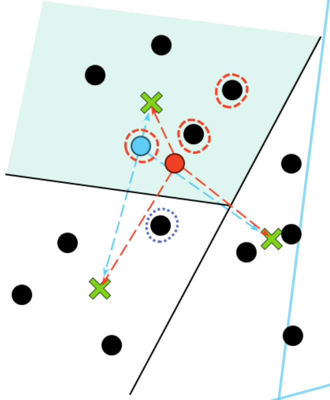






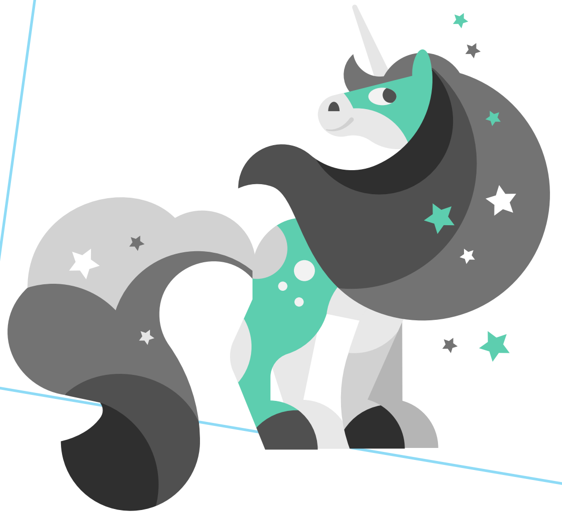






AND NOW…

- Dealing with large datasets.
- Performance numbers!
- How to make updates and influence rebuilds.



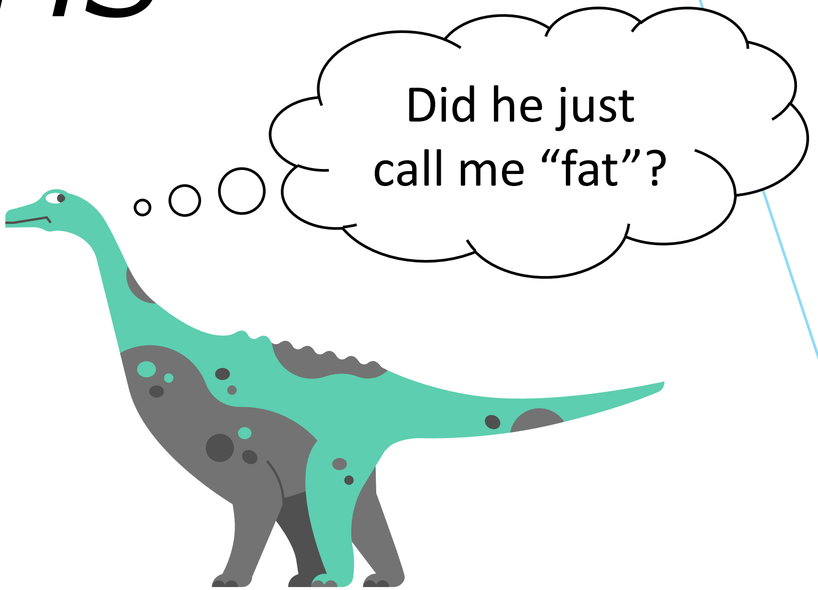
All this, today in…

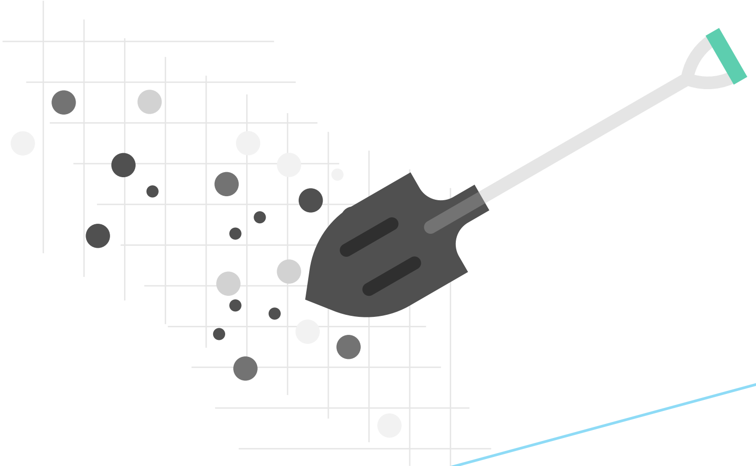
TOPICS IN VECTOR DATABASES!

TWO COMMON PROBLEMS

Some indexes suffer from:

1. **Large memory footprint**:
   → 1. Sharding
   → 2. Quantization
   → 3. Composite index
   → 4. Disk-resident index
2. **Need to rebuild periodically**:
   → 5. Liveness layer
   → 6. Segmenting
   → 7. Updatable index



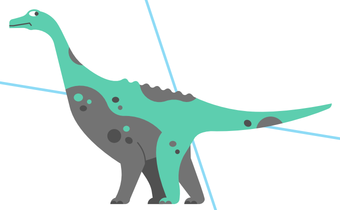


1. SHARDING

1. Split data to *k* disjoint sets
    - *N/k* points per shard
2. Build index per shard
3. Distribute shards across machines
4. Query in parallel, merge results

- Benefits:
    - *N/k* fits in machine RAM.
    - Search (and insert) in parallel.
- Downsides:
    - Need *k* machines, *k* can be large.
    - How to deal with edges?
    - Still lots of RAM.
    - Only delays the issue.

Sharding *is* used by most systems.
But not really a solution for memory.



2. QUANTIZATION

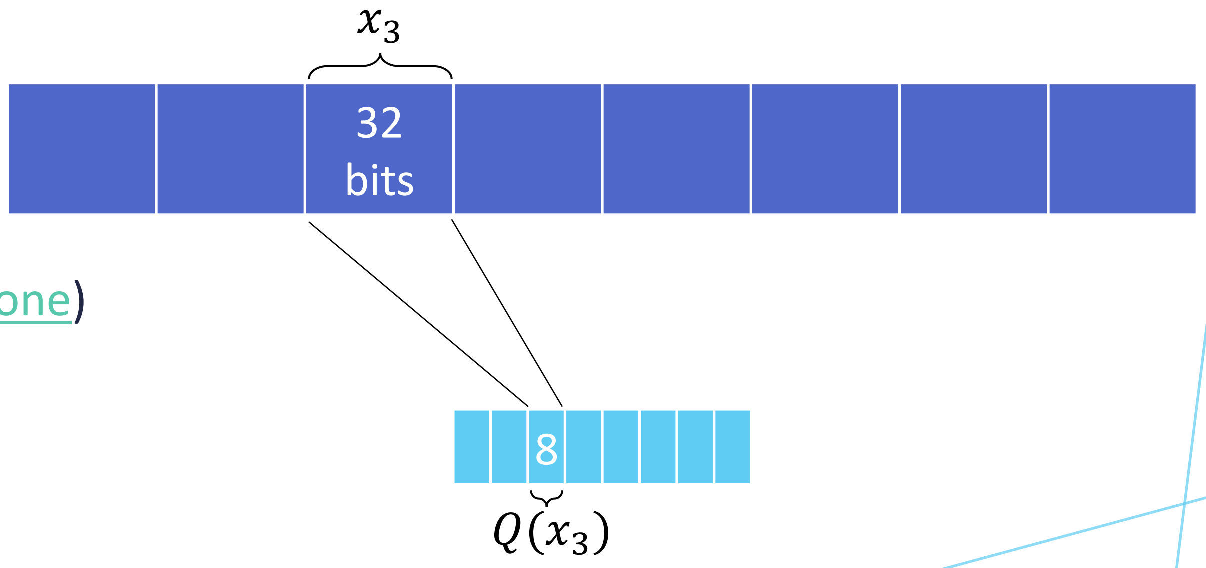
- Represent vector with fewer bits
  - Still has D dimensions!
- Loses accuracy
- Several main approaches:
  - Scalar Quantization – quantize each element
  - Vector Quantization – represent entire vector as “code word”
  - Product Quantization – combine VQ on parts of vector [Jegou, TPAMI’10]

SQn: SCALAR QUANTIZATION

- Reduce each component to *n* bits
- Uniform quantization:

$$\mathrm{binsize_i} = \frac{\max(x_i) - \min(\mathrm{x_i})}{2^n} \qquad \mathrm{q}(x_i) \cong \frac{x_i - \min(x_i)}{\mathrm{binsize_i}}$$

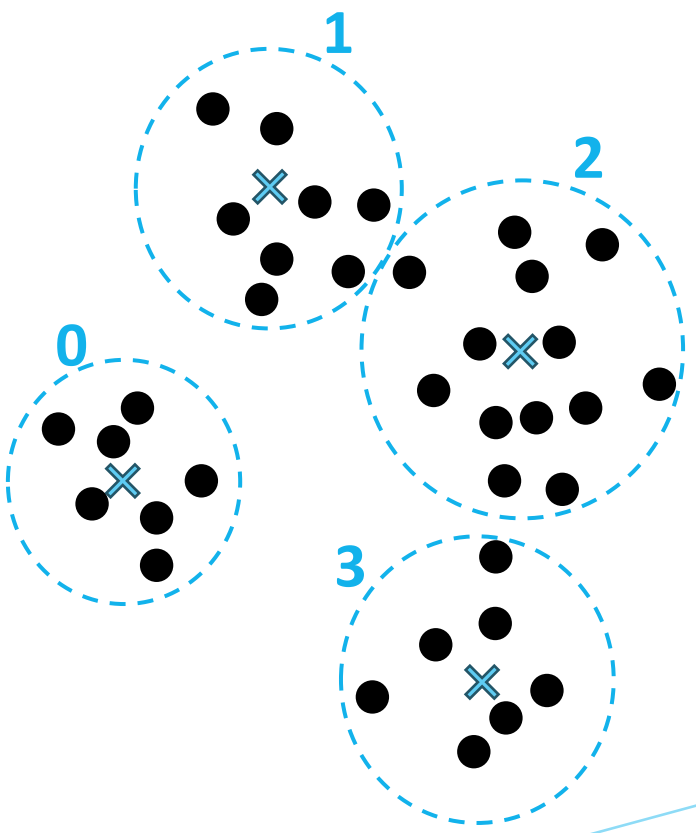
- Example: 32 → 8 (“SQ8”)
  - x4 less memory
  - x2 faster comparison [Qdrant, 2024]
  - ~1% recall loss [Qdrant, 2024]
  - Commonly used with other indices (Pinecone)
- *n* < 8 not common, inaccurate
  - SBQ @ Timescale uses 2 bits



VQ: VECTOR QUANTIZATION

1. Cluster vectors.
   - Codebook = set of centroids.
2. Assign code word to each cluster.
3. $q(x)$ maps x to nearest cluster:
   - Encode x as cluster code word
   - Use centroid in distance computation

- $O(DNk)$ time per k-means iteration
- $O(kD)$ space for codebook
- $O(N \log k)$ space for vectors



VQ: VECTOR QUANTIZATION

1. Cluster vectors.
    - Codebook = set of centroids.
2. Assign code word to each cluster.
3. $q(x)$ maps x to nearest cluster:
    - Encode x as cluster code word
    - Use centroid in distance computation

- $O(DNk)$ time per k-means iteration
- $O(kD)$ space for codebook
- $O(N \log k)$ space for vectors

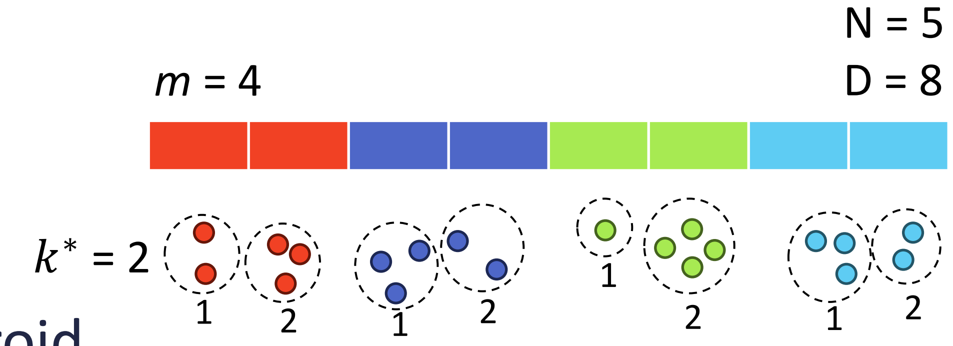
- Problem: k must be large
    - D-dimensional space → k regions
    - Resolution grows exponentially in D
    - …so k must also grow exponentially!
    - Small k → large error
- How large? Very large
    - k = **1K to 1M** for SIFT1B D=128 [Jegou, TPAMI'10]

→ **Too slow!**

→ **Too big!**

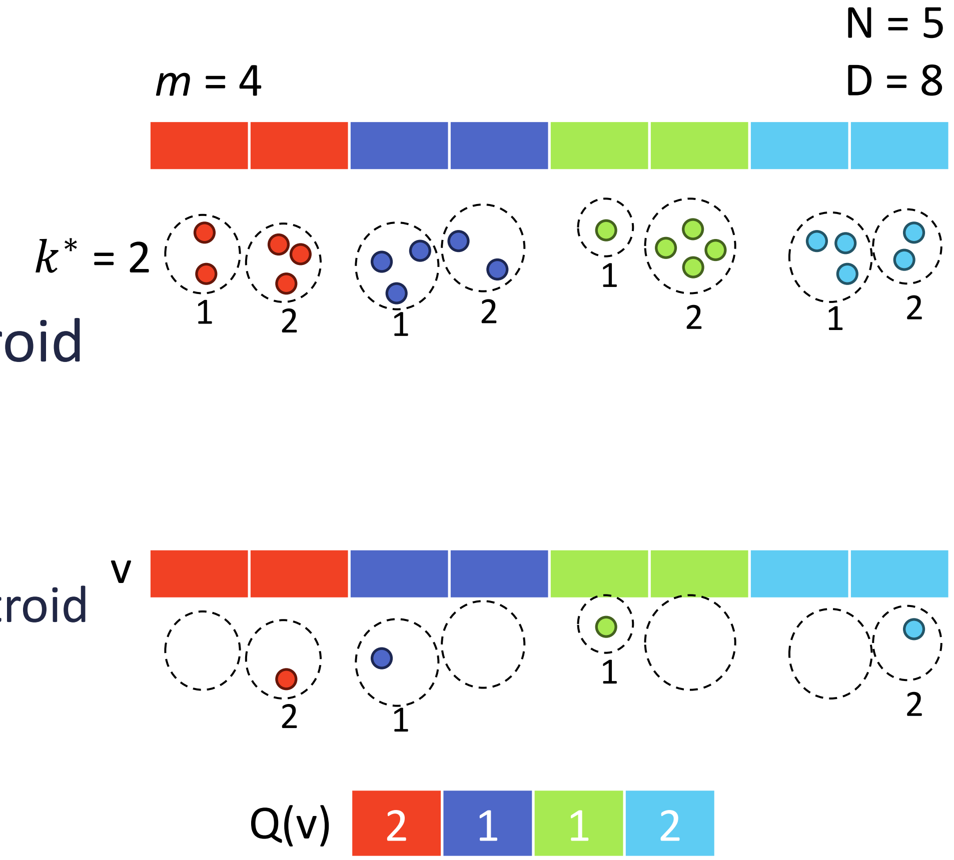
PQ: PRODUCT QUANTIZATION

- Split space to m chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id 1 … k^* to each subspace centroid



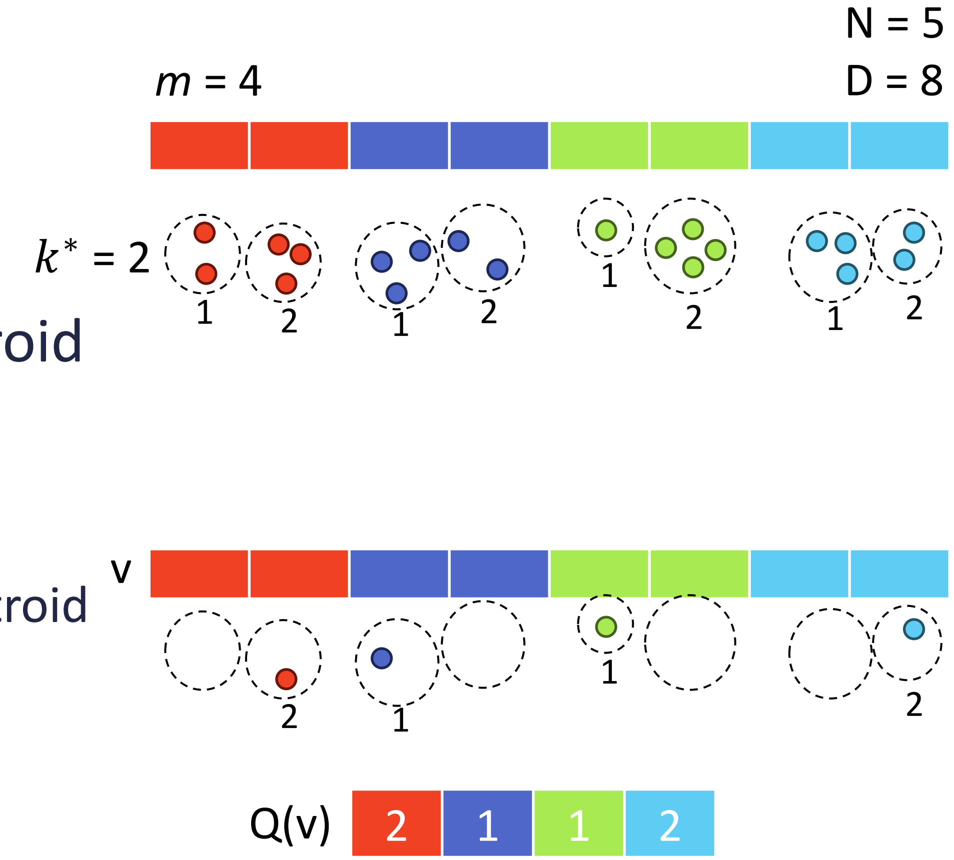
PQ: PRODUCT QUANTIZATION

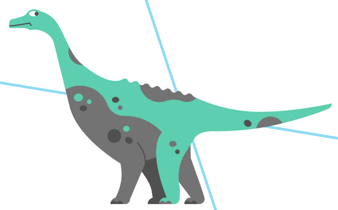
- Split space to *m* chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id 1 … k^* to each subspace centroid
- To quantize vector v:
  - Split
  - Replace each chunk with id of nearest centroid
  - Concatenate



PQ: PRODUCT QUANTIZATION

- Split space to *m* chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id 1 … k^* to each subspace centroid
- To quantize vector v:
  - Split
  - Replace each chunk with id of nearest centroid
  - Concatenate
- To approximate v from Q(v):
  - Concatenate centroids indicated by ids





BENEFITS OVER VQ

- Assign n_{bits} to each subspace
  - Choose $k^* = 2^{n_{bits}}$
- **Strong representational power:**
  - Represent $k = (k^*)^m$ centroids in $\mathbb{R}^D$
  - *m* subspaces of D/*m* dimensions
  - $\{1..\ k^*\}$ x $\{1..\ k^*\}$ x … x $\{1..\ k^*\}$
- **Faster k-means clustering**:
  - With VQ: $O(DN\boldsymbol{k})$ per iteration
  - With PQ: $O(m)O\left(\frac{D}{m}N\boldsymbol{k^{1/m}}\right)$ per iteration

Run k-means m times

Example:

D = 128, FP32, *m* = 8, n_{bits}=8 (k^*=256, $\boldsymbol{k}$**=2**^{64})
Without PQ: 32x128 = **4096** bits per vector
With PQ: 8x8 = **64** bits per vector

- **Lower storage**:
  - Without PQ: $32DN$ bits (*D* floats per vector)
  - With VQ: $\log_2 k$ bits per vector
    $+\ 32k\boldsymbol{D}$ bits for codebook (k centroids)
  - With PQ: $m\log_2 k^{1/m} = m \cdot n_{bits}$ bits per vector
    $+\ 32k\boldsymbol{\frac{D}{m}}$ bits for codebook

BENEFITS OVER VQ

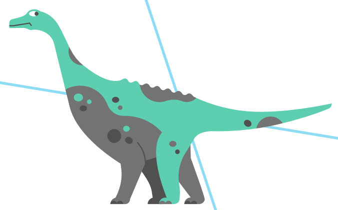
Compared to VQ with k clusters

- Faster clustering: $O(\cdots k) \rightarrow O\left(\cdots k^{1/m}\right)$
- Smaller storage: $O(\cdots D) \rightarrow O\left(\cdots \frac{D}{m}\right)$
- Similar representational power (approximately)
  - k centroids in $\mathbb{R}^D$

Compared to raw vectors:

- Fast comparison: precalc k^* subcentroid distances m times

Run k-means m times



USING PQ

- Is PQ + brute force good enough?
- SIFT 1M dataset: D=128, N=1000000
- As before, D = 128, FP32, m = 8, n_{bits}= 8 (k^*=256, k=2^{64})

	Flat Index	PQ
Memory	512 MB	4 MB
Query latency	8.26 ms	1.49 ms
Recall	100%	**50%**

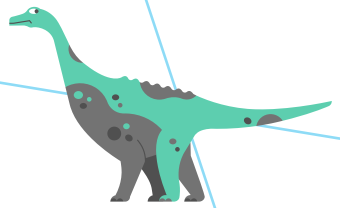
[Briggs, 2024]

Much smaller

Bit faster, but not enough

Less accurate (sometimes sufficient!)

- PQ: **excellent memory** usage**, poor accuracy**
  - Get to ~75% recall with larger n_{bits}, m

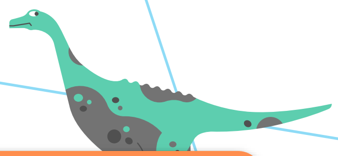


3. COMPOSITE INDEX

- Combine **index** and **quantizer** for double benefit
  - Or index + index! (e.g. IVF + HNSW)
  - Even index + index + quantizer!
- Example: IVF + PQ
- Variations:
  - Transform vectors before quantization (OPQ)
  - Re-rank after query using true values
  - Residual: quantize *v* - centroid, not *v* (IVFADC)
  - Asymmetric: do not quantize *q* when searching (IVFADC)

Top-of-the line, production indexes today are usually **composite** and/or **graph-based**

IVFPQ = IVF + PQ



- During build:
  - Partition to cells with IVF
  - Learn PQ codebook
- Insert:
  - Select cell with IVF
  - Store code
- Query:
  - Select cell with IVF
  - Search quantized vectors in cell
  - (Optional: reorder vectors using original data)

Still small ☺

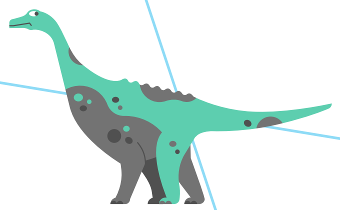
Very fast!

	Flat Index	PQ	IVFPQ	IVF256 PQ32x8
Memory	512 MB	4MB	9MB	40MB
Query latency	8.26 ms	1.49 ms	0.09 ms	0.73 ms
Recall	100%	50%	52%	74%

Same accuracy

$m = 32$
$n_{bits} = 8$

Composite indexing:
- Speed up PQ
- Maintain low memory
- Hard to completely overcome PQ error (use SQ8 for higher accuracy)



IVF+HNSW

- Build:
  - Create many small cells with IVF
    - E.g., 4096
  - Store centroids in HNSW
- Insert *v*:
  - Use HNSW to find cell
    - (cell selection now approximate!)
  - Insert *v* to cell
- Query *q*:
  - Use HNSW to find cell
  - Compare *q* to vectors in cell

	Flat	IVF256 PQ32x8	IVF4096 HNSW32	IVF4096 HNSW32 PQ32
Memory	512 MB	40MB	523MB	43MB
Query latency	8.26 ms	0.73 ms	0.55 ms	0.55 ms
Recall	100%	74%	90%	69%

IVF+HNSW
- Fast!
- Excellent recall
- Memory heavy

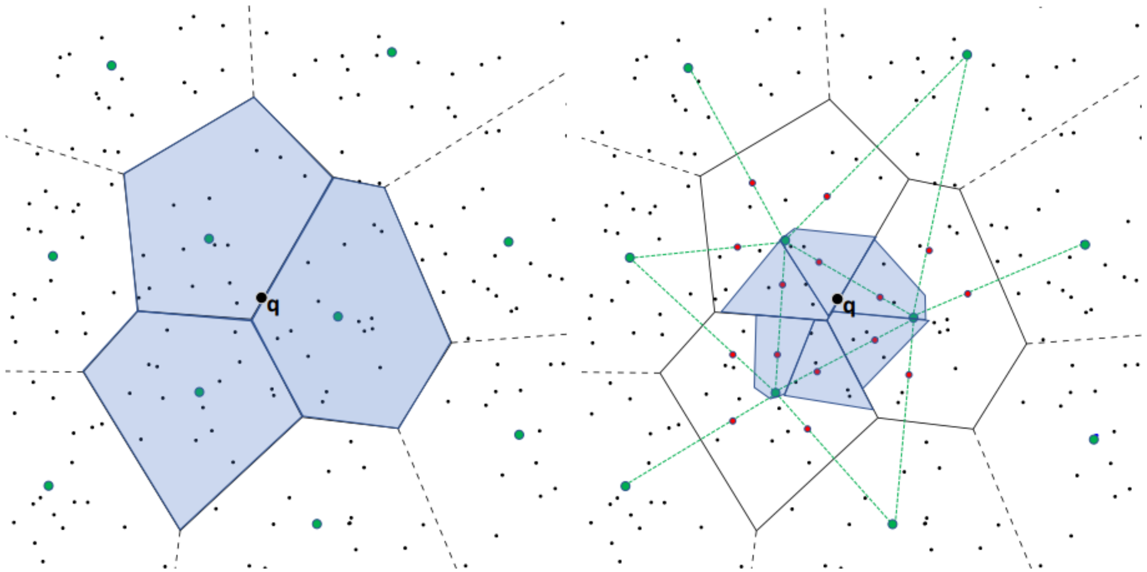
Add PQ:
- Still fast, less memory, decent recall

THE QUANTIZATION/COMPOSITE INDEX CINEMATIC UNIVERSE

- Lots of research
  - Relevant research – techniques used in practice! **(we shall see a few)**
- OPQ: rotate vectors for optimal PQ [Ge et al., TPAMI 2013]
- IVFADC-R: Three-level quantization + re-ranking [Jégou et al., ICASSP’11]
- IVFOADC+G+P: Near SotA composite index, very fast [Baranchuk, ECCV’18]
- Fast SIMD implementation [André et al., PAMI’19] [Guo et al., ICML’20]
- Additive quantizers [Babenko & Lempitsky, CVPR’14].
- Residual vector quantizers [Liu et al, arXiv’15]
- Find more in FAISS docs, [Matsui, MTA’18], and [Pan, VLDBJ ’24]

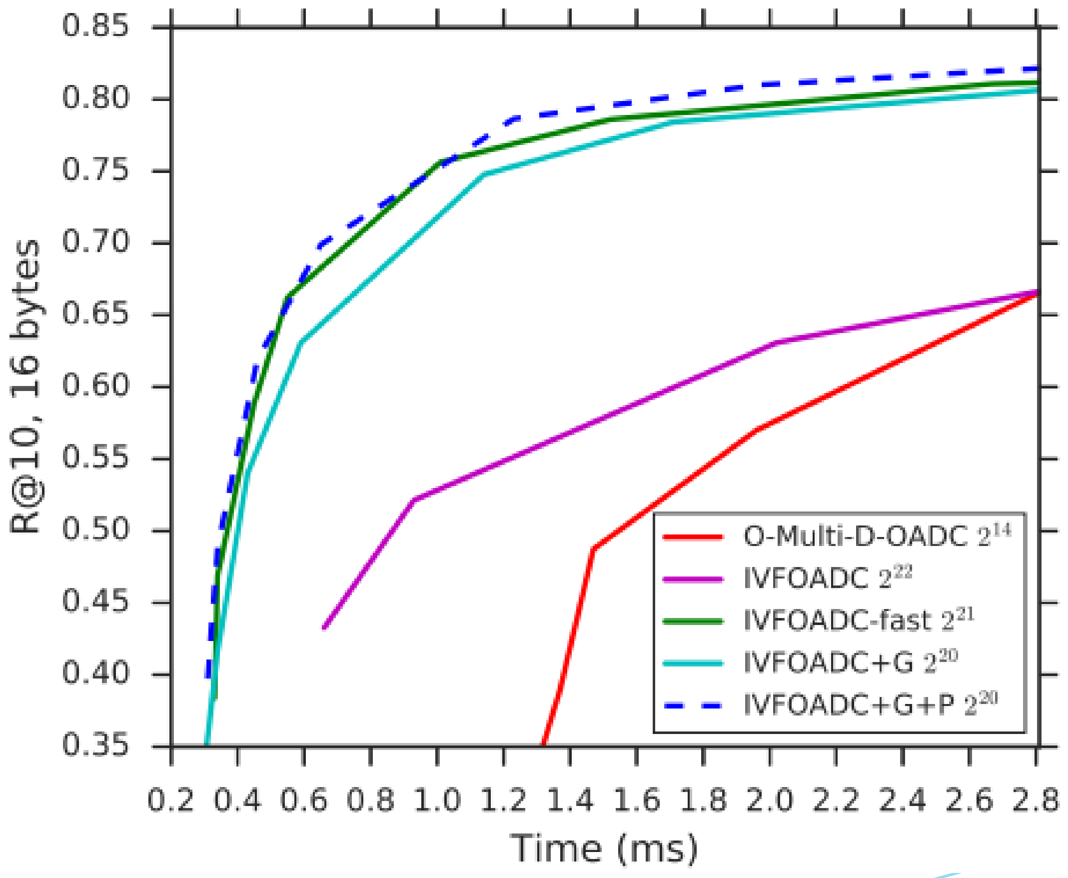
IVFOADC+G+P [BARANCHUK, ECCV'18]

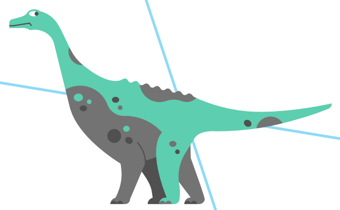
- Near SotA composite index
- Combines existing techniques:
  - IVF, HNSW, OPQ, residual encoding (IVFADC), asymmetric distance
- Novel grouping, pruning procedure:
  - Subdivide clusters (without extra memory!)
  - Skip subdivisions far from query.



IVFOADC+G+P *[BARANCHUK, ECCV'18]*

- Near SotA composite index
- Combines existing techniques:
  - IVF, HNSW, OPQ, residual encoding (IVFADC), asymmetric distance
- Novel grouping, pruning procedure:
  - Subdivide clusters (without extra memory!)
  - Skip subdivisions far from query.
- Results:
  - Very fast (can go <1ms)
  - Low recall (very low for low memory)





4. DISK RESIDENT INDEXES

- What if N > 1B ?
  - Indexes are memory-intense
  - Quantization reduces recall
- Offload index to SSD
  - New index structures with careful IO optimizations
  - Updates, rebuilds now more expensive
- For static data:
  - ANNOY – tree-based [Bernhardsson ’20].
  - DiskANN – graph-based [Subramanya, NeurIPS ‘19]
  - SPANN – learned hash [Chen, NeurIPS’21]
- For dynamic data:
  - FreshDiskANN – graph based [Singh, arXiv ‘21]
  - Neos – flat index [Huang, ICDE’24]

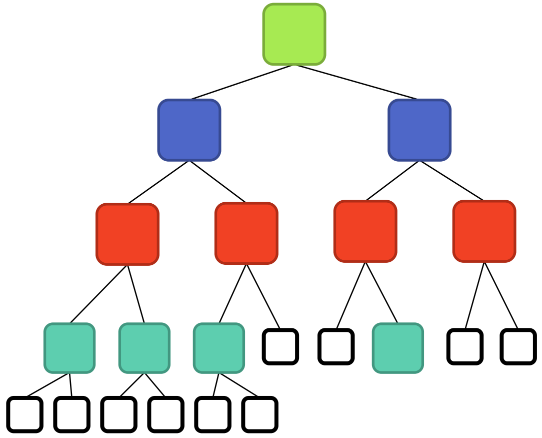
Active research area
We shall see several papers

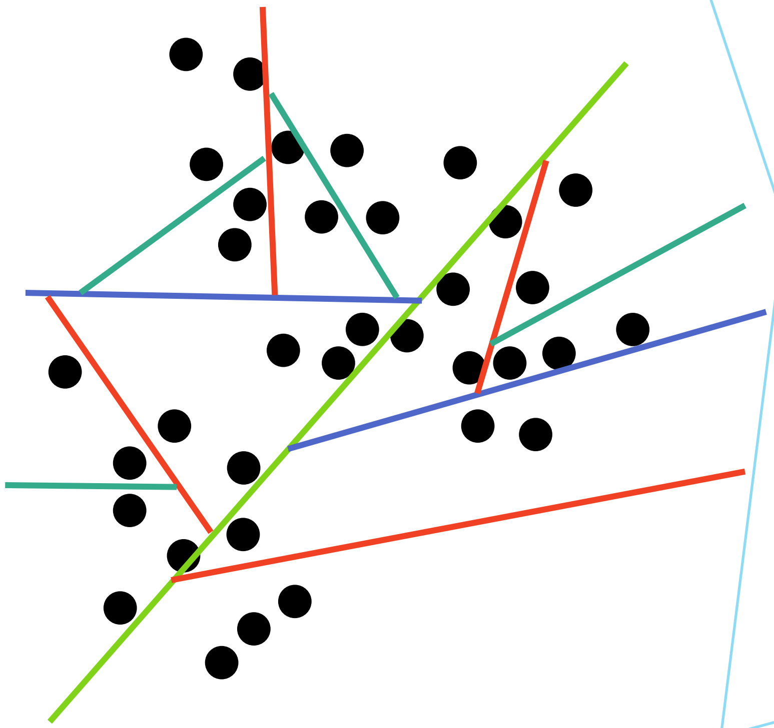
ANNOY [BERNHARDSSON '20]

- Variation of Random Projection Tree (RPTree)
- Recursively split dataset randomly

1. Choose random direction u
2. Project data on u
3. Split points, half on each side
   - Find median projection t
   - Based on $x^T u \geq t$
4. Recurse until $< k$ items per leaf

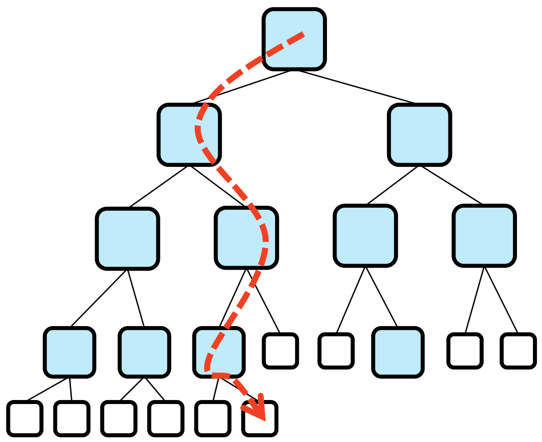
- Build random forest for accuracy

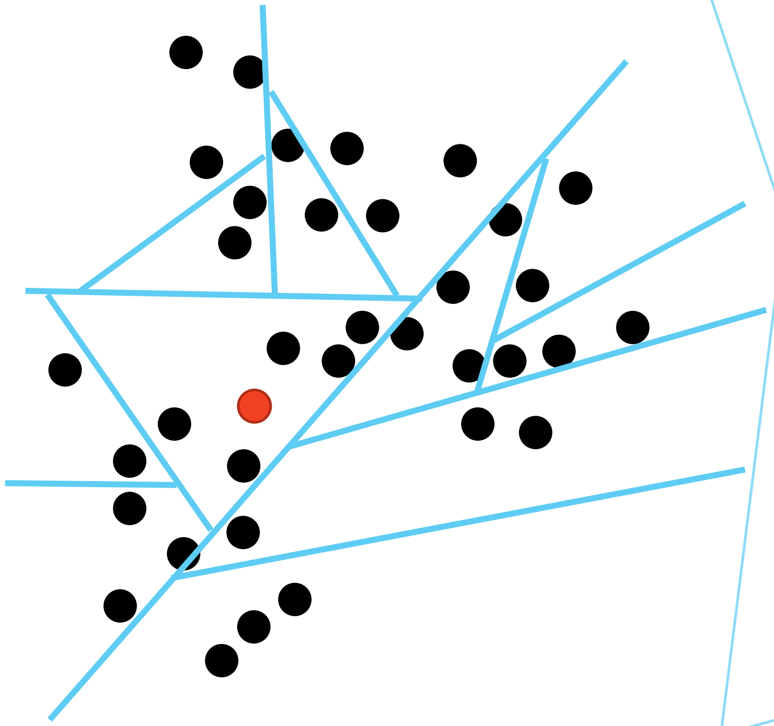




ANNOY [BERNHARDSSON '20]

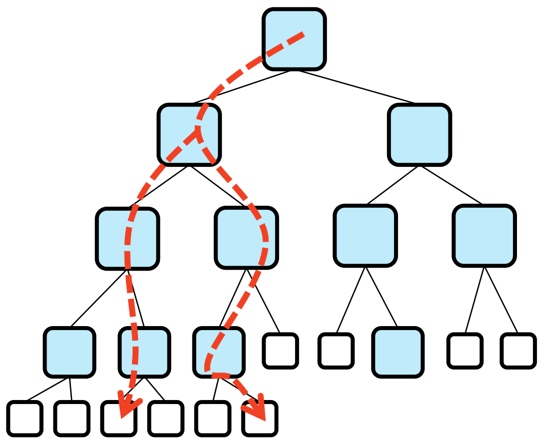
- To query, search binary tree
  - At each split, check if $q^T u \geq t$

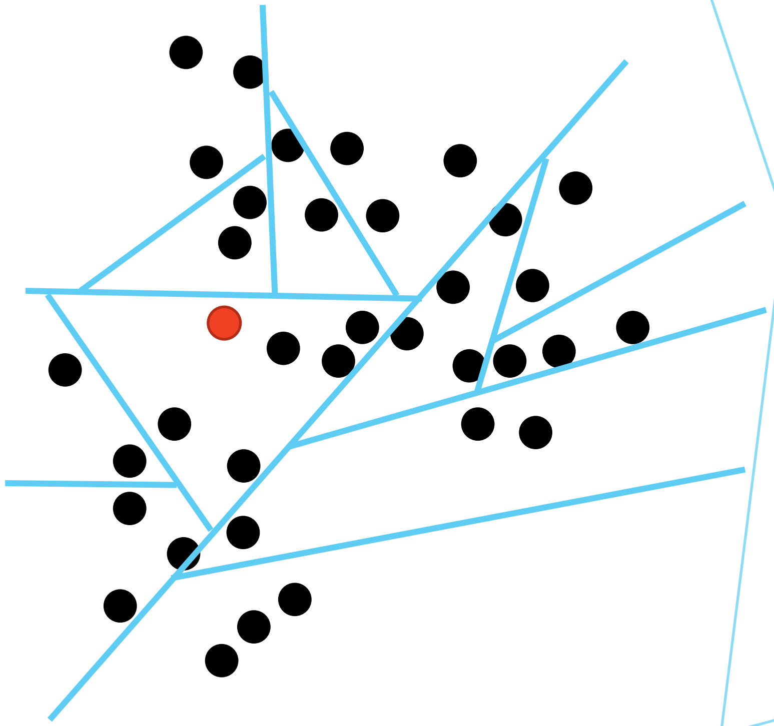




ANNOY [BERNHARDSSON '20]

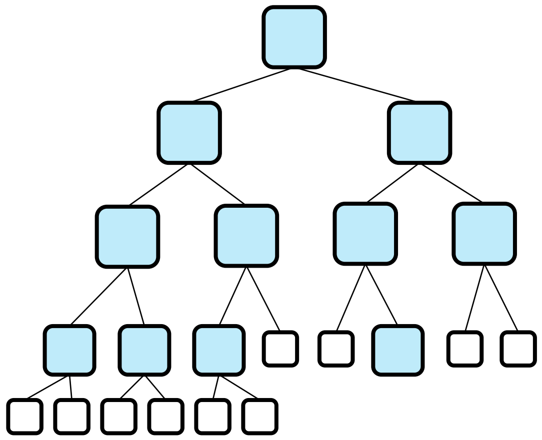
- To query, search binary tree
  - At each split, check if $q^T u \geq t$
- q near split?
  → go down both paths!
- Priority queue:
  - "both paths"
  - Fast search across all trees.
  - Parallel searchers.

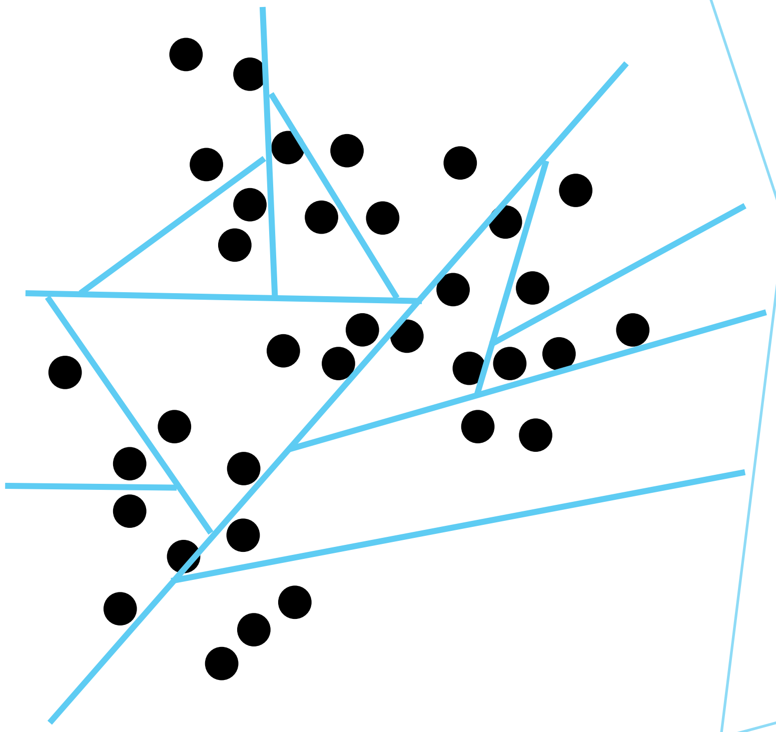


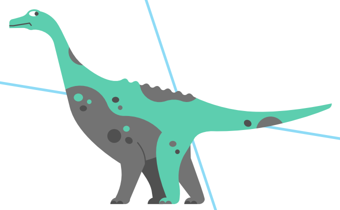


ANNOY [BERNHARDSSON '20]

- Encoded as static file.
  - **Just mmap and query**
  - Fast loads, unloads.
  - Page cache handles memory.
  - Easy to share across processes
- Can build to disk directly
- Quite fast.
- No updates, needs rebuilding.
- High memory:
  - $O(ND)$ for split planes plus
  - $O(N/k)$ for nodes
- Used by Spotify, ClickHouse.







DISKANN [Subramanya, NeurIPS '19]

- Why not drop index on SSD?
- SSD performance:
  - Throughput limited by random reads
  - Latency limited by num requests (round-trips)
- Standard graph-based index:
  - Complex structure

  → Lots of random reads

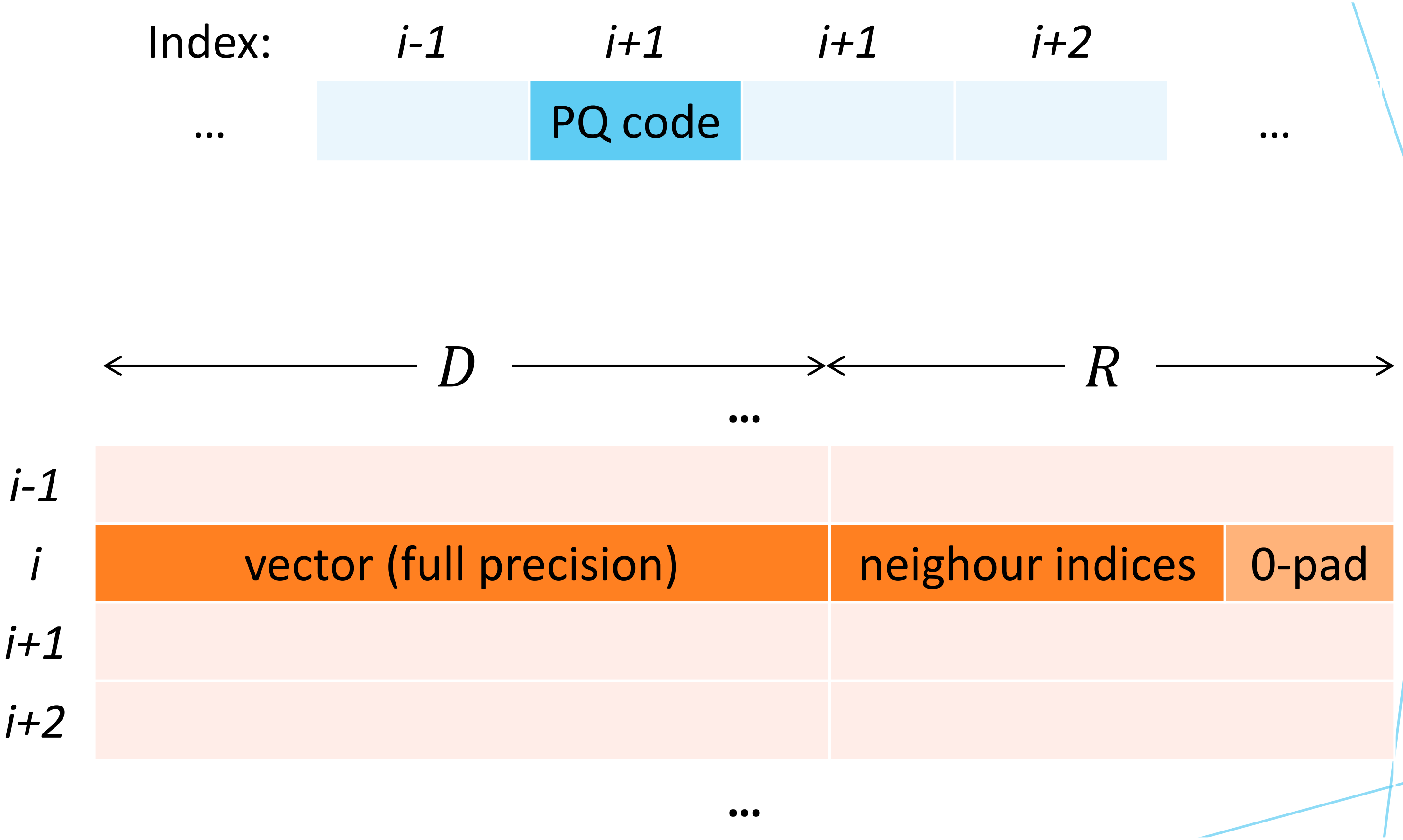
  → Hundreds of I/O roundtrips

→ Redesign index: few reads, few IO requests

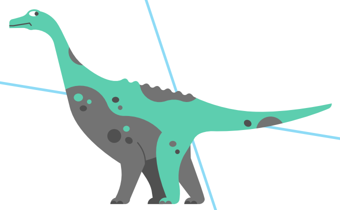
Why graph based?
SotA for high recall,
fast results

DISKANN LAYOUT

- In RAM:
  - PQ-compressed vectors
- On disk:
  - Full precision vector
  - Index for neighbours (up to R, zero padded)
- Easy to compute offset for vector *i*



DISKANN GRAPH CREATION



- Graph hop = disk access
- Want to reduce hops!
- Make graph where:
  - Distance to q decreases **exponentially**

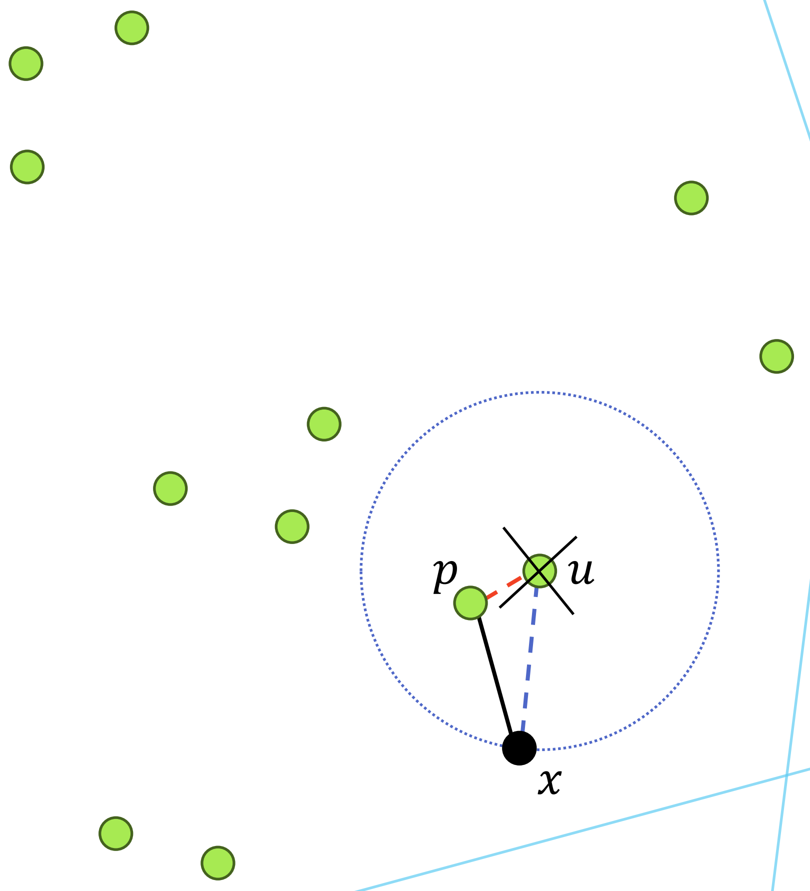
    → logarithmic steps in greedy search

    → less I/O
  - Bounded out-degree by R

    → sparse graph → less bandwidth

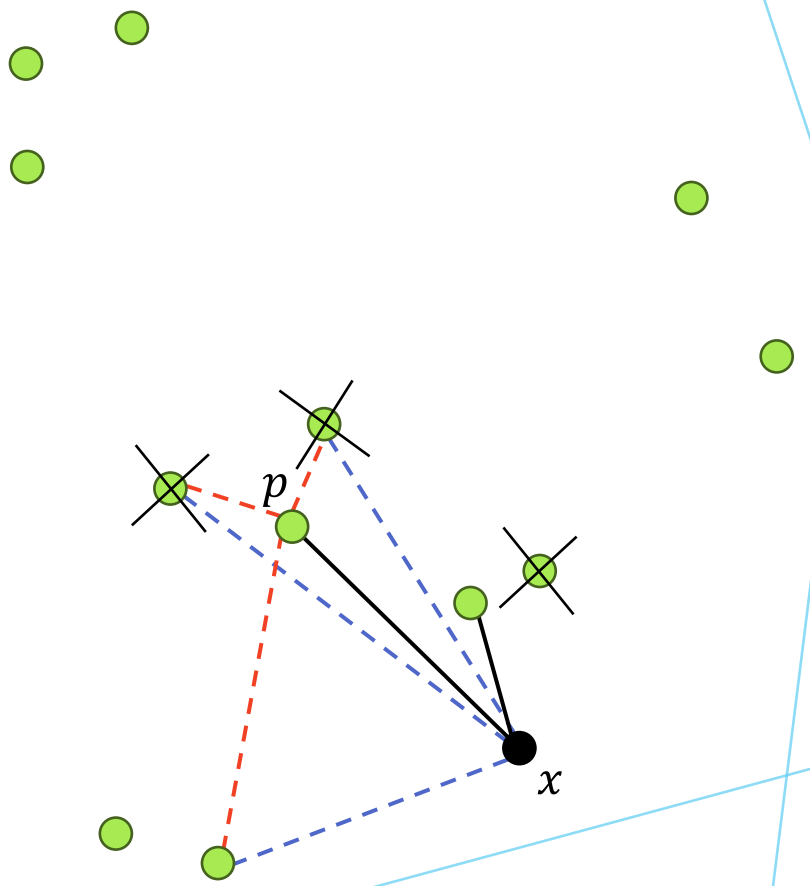
PRUNING IN HNSW, NSG

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \to p$
  - Discard nodes in V near p:
    - If u closer to p than to x: $d(p,u) < d(u,\mathrm{x})$
    - Remove u from candidates V
  - Repeat



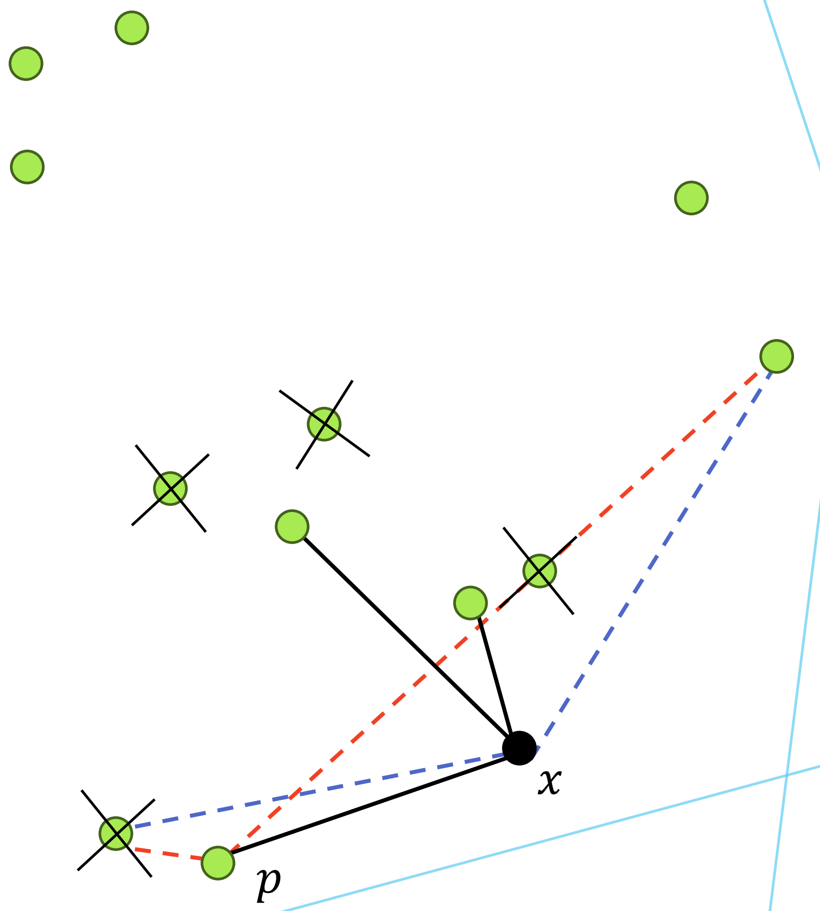
PRUNING IN HNSW, NSG

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \rightarrow p$
  - Discard nodes in V near p:
    - If u closer to p than to x: $d(p,u) < d(u,\mathrm{x})$
    - Remove u from candidates V
  - Repeat



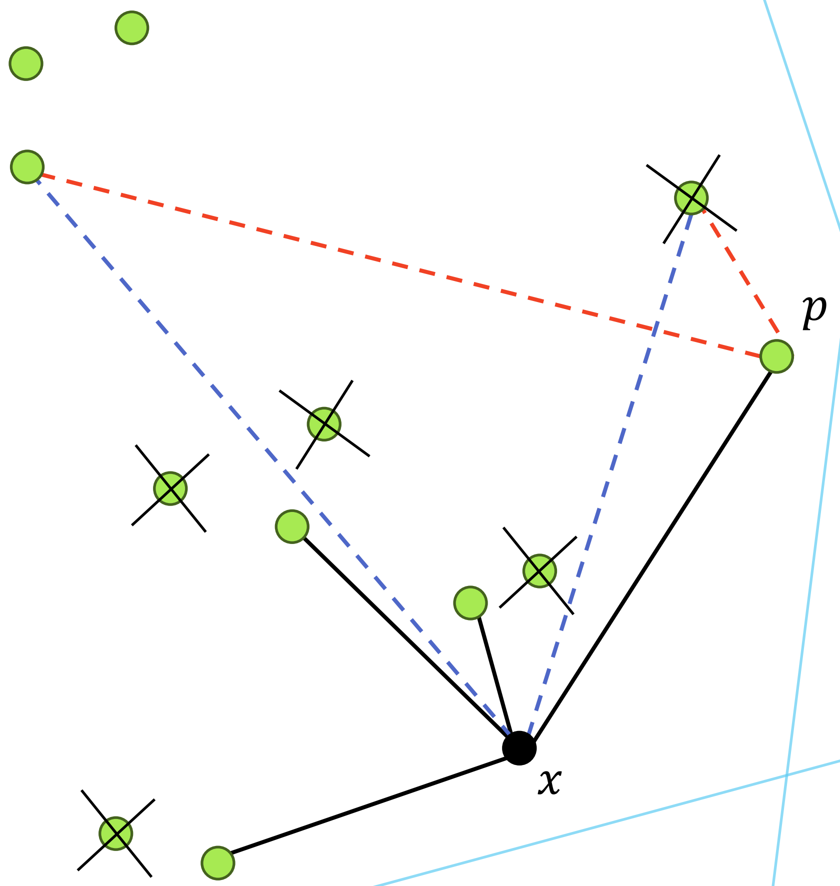
PRUNING IN HNSW, NSG

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \to p$
  - Discard nodes in V near p:
    - If u closer to p than to x:
      $d(p, u) < d(u, \mathrm{x})$
    - Remove u from candidates V
  - Repeat



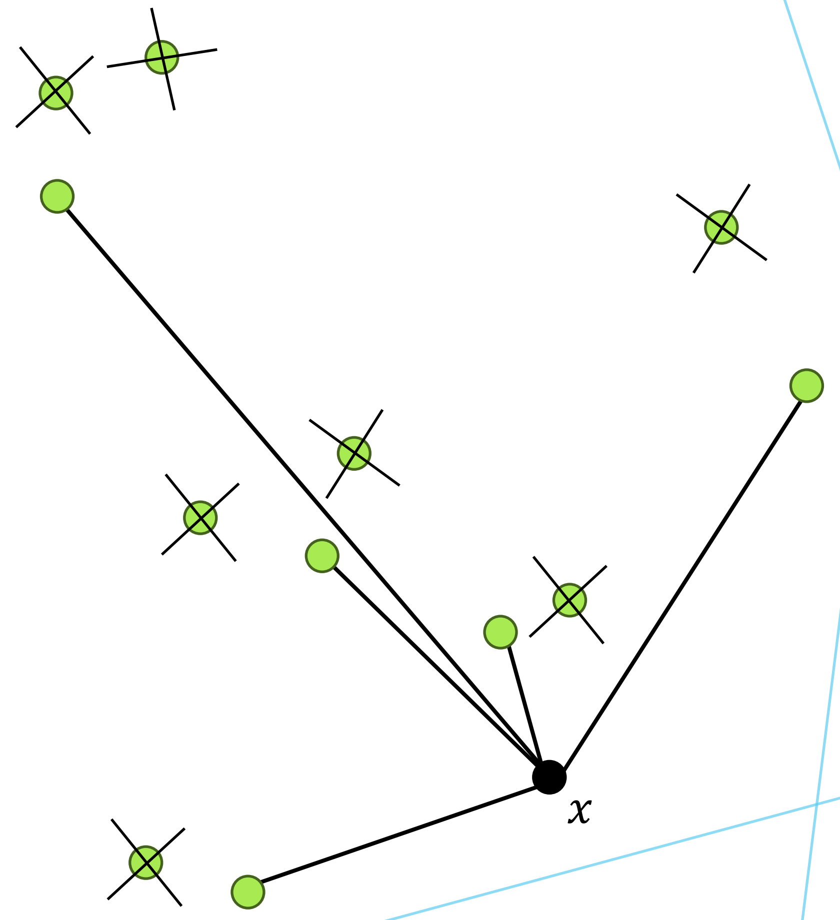
PRUNING IN HNSW, NSG

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \to p$
  - Discard nodes in V near p:
    - If u closer to p than to x: $d(p,u) < d(u,\mathrm{x})$
    - Remove u from candidates V
  - Repeat



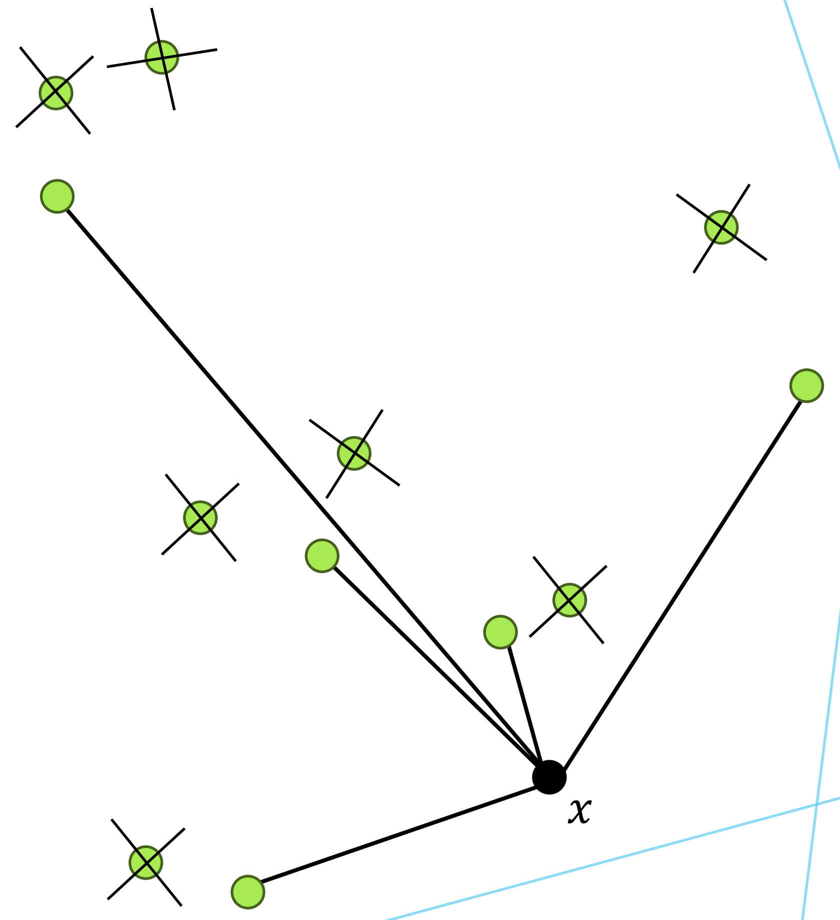
PRUNING IN HNSW, NSG

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \to p$
  - Discard nodes in V near p:
    - If u closer to p than to x: $d(p,u) < d(u,\mathrm{x})$
    - Remove u from candidates V
  - Repeat



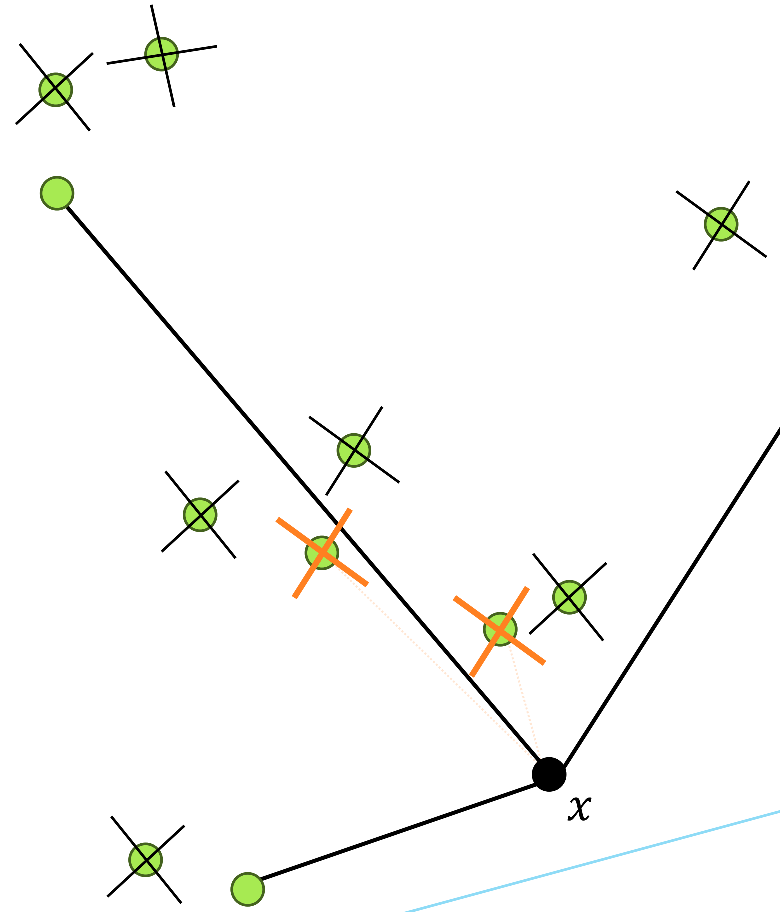
ROBUST PRUNING IN VAMANA

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \rightarrow p$
  - Discard nodes in V near p:
    - If u closer to p than to x:
      $\boldsymbol{\alpha} \cdot d(p,u) < d(u,\mathrm{x})$
    - Remove u from candidates V
  - Repeat
- **Distance increases by $\alpha > 1$ each hop**



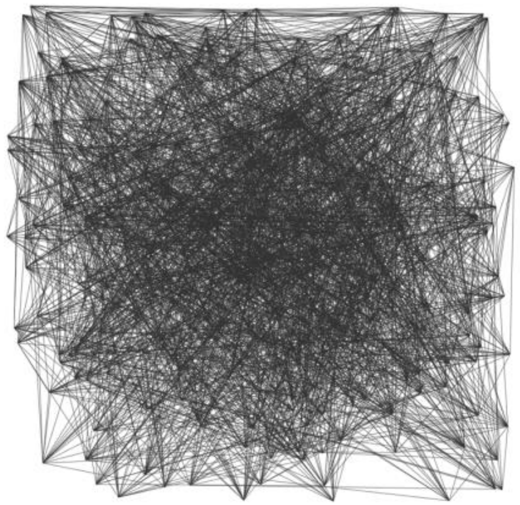
ROBUST PRUNING IN VAMANA

- Build x's out-edges:
  - V = points near path from entry to x
  - Find p = closet to x in V
  - Add edge $x \rightarrow p$
  - Discard nodes in V near p:
    - If u closer to p than to x: $\boldsymbol{\alpha} \cdot d(p, u) < d(u, \mathrm{x})$
    - Remove u from candidates V
  - Repeat
- **Distance increases by $\alpha > 1$ each hop**



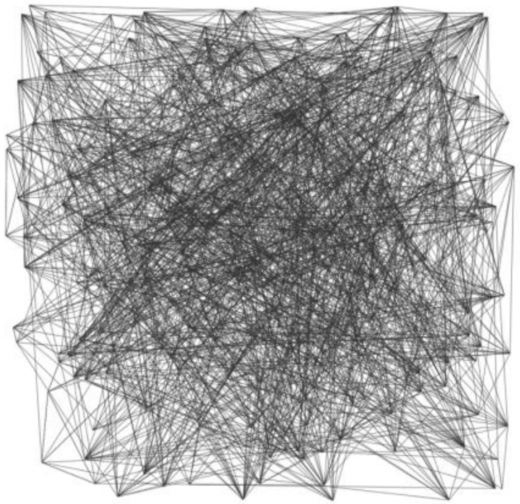
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges



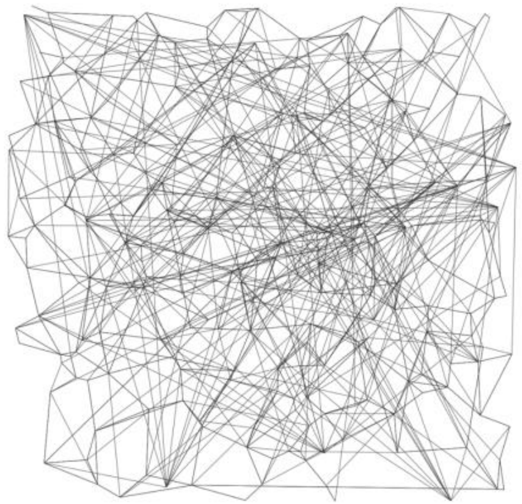
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$



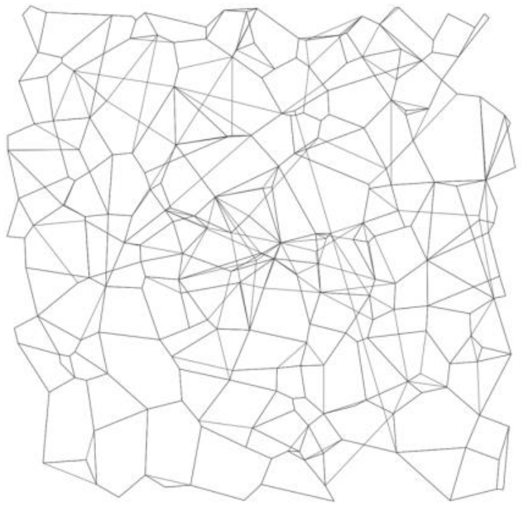
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$



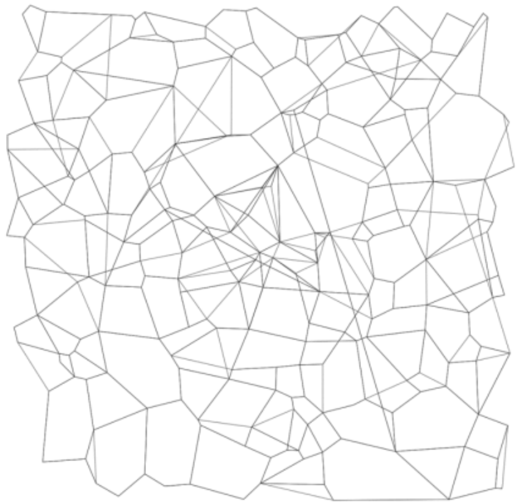
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$



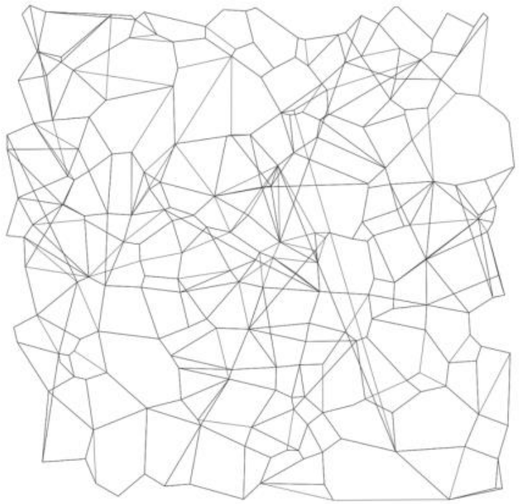
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$
  - Long-range pass → refine edges with $\alpha > 1$



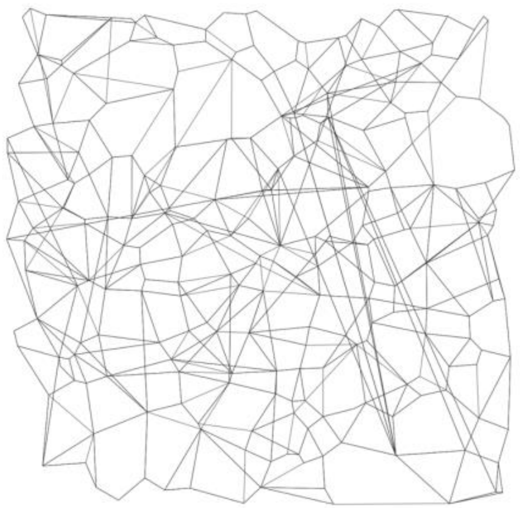
DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$
  - Long-range pass → refine edges with $\alpha > 1$



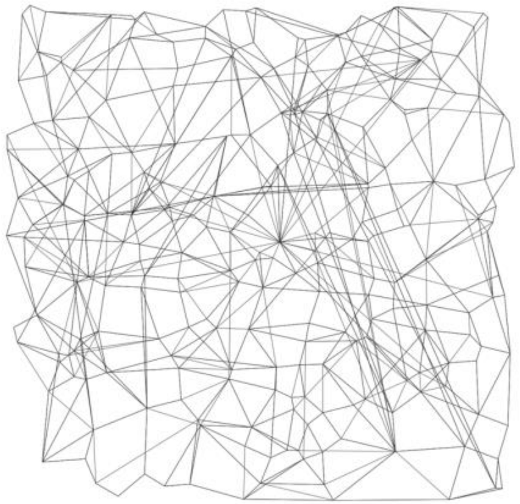
DISKANN VAMANA ALGORITHM

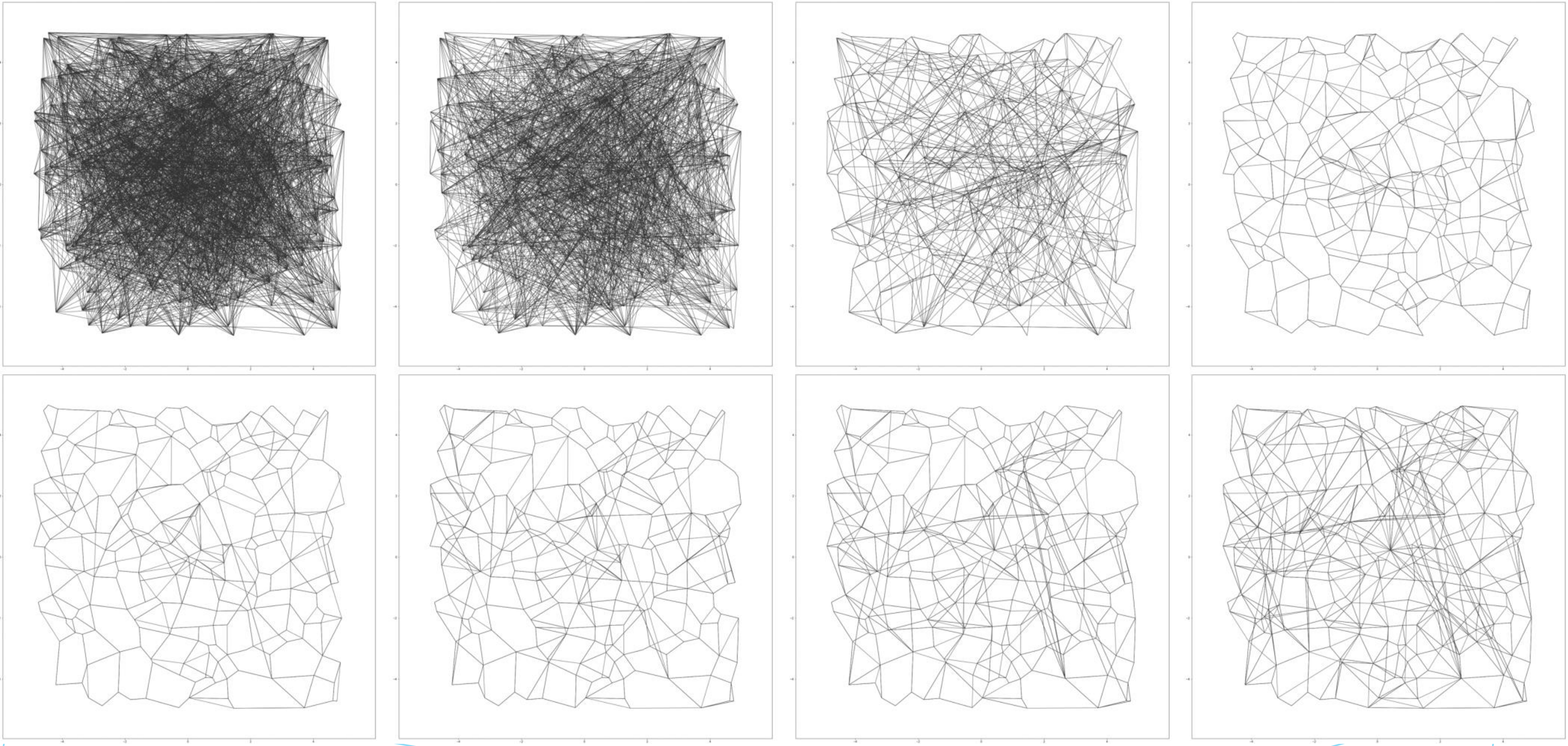
- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$
  - Long-range pass → refine edges with $\alpha > 1$



DISKANN VAMANA ALGORITHM

- Vamana algorithm:
  - Initialize with R random edges
  - Short-range pass → refine edges with $\alpha = 1$
  - Long-range pass → refine edges with $\alpha > 1$





DISKANN IMPLEMENTATION

In practice, DiskANN code differs from paper:

- Start with **empty graph** (not random!)
  - Possibly from FreshDiskANN
- **Single pass** over nodes (not two!)
  - When adding v, iterate over neighbour candidates twice ($\alpha = 1$ and $\alpha = 1.2$)

  → Not sure two passes even do anything

  → My implementations work well with single pass
- **Allow more than R out-edges** during indexing
  - Trim if $1.3 \cdot R$, or after indexing.
  - Likely to thread synchronization.

DISKANN FOR LARGE GRAPHS

1. Cluster to k
2. Shard using cluster
   - List vector in $\ell > 1$ shards
3. **Create graph per shard**
4. **Merge graphs** (union of edge lists)
   - Preserve $< R$
5. Quantize with PQ
   - Stored in RAM
   - Used for querying

Otherwise too big to hold in RAM
$k = 40$

Preserves connectivity
(no need to probe many shards)
$\ell = 2$

Typically 256 bits

DISKANN QUERYING

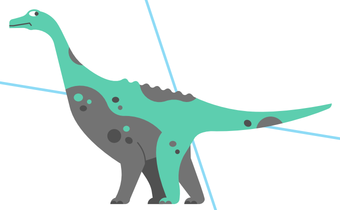
- Recall greedy search:
  1. $p \leftarrow$ best unvisited candidate
  2. Add p's neighbours to candidate list
  3. Prune candidates to best L
  4. Mark p as visited
  5. Repeat
- Used by most graph indexes
- DiskANN adds several optimizations!

Best = nearest to q

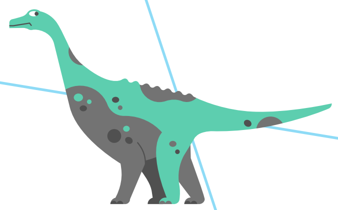
Otherwise finding next p is slow

Stop when all candidates visited

DISKANN OPTIMIZATIONS



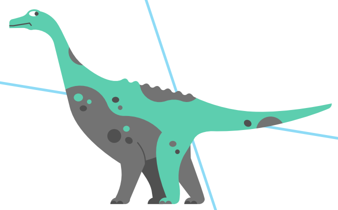
- Use PQ during querying
  - Avoids reading vectors for all neighbours
  - RAM-resident



DISKANN OPTIMIZATIONS

- Use PQ during querying
  - Avoids reading vectors for all neighbours
- Beam search:
  - Expand candidate list by $W > 1$.

1. $p \leftarrow$ best unvisited candidate
2. Add p's neighbours to candidate list
3. Prune candidates to best L
4. Mark p as visited
5. Repeat

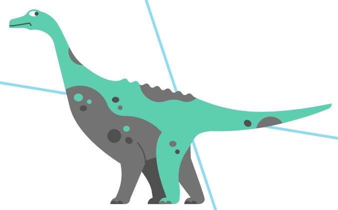


DISKANN OPTIMIZATIONS

- Use PQ during querying
  - Avoids reading vectors for all neighbours
- Beam search:
  - Expand candidate list by $W > 1$.
  - SSD has “deep” I/O queue (32+)
  - Can read W random pages in parallel
  - $W = 1$ → regular greedy search
  - Large W → wasting bandwidth + compute → increased latency
  - Sweet spot: $W \in [2,4,8]$

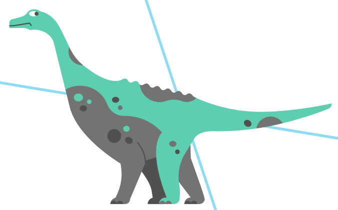
1. $\boldsymbol{p_1 \ldots p_w} \leftarrow \boldsymbol{W}$ best unvisited candidates
2. Add $\boldsymbol{p_1 \ldots p_w}$’s neighbours to candidate list
3. Prune candidates to best L
4. Mark p as visited
5. Repeat

DISKANN OPTIMIZATIONS

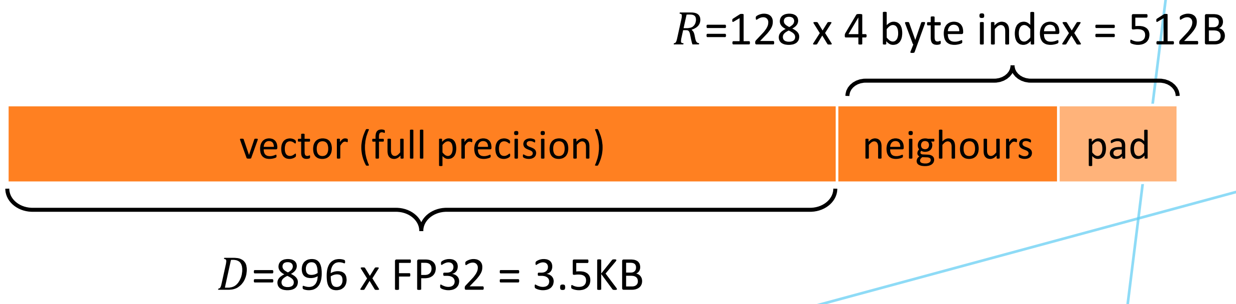


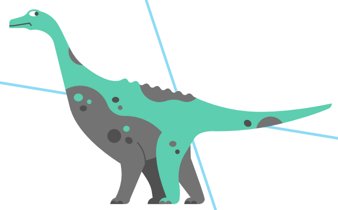
- Use PQ during querying
  - Avoids reading vectors for all neighbours
- Beam search:
  - Expand candidate list by $W > 1$.
- Cache vectors near entry point
  - Keep in RAM entry point neighbourhood
  - Vectors up to C hops from entry point
  - Cost $R + R^2 + \cdots + R^C = O(R^{C+1})$ vectors
  - Generally $C = 3$ or $C = 4$

DISKANN OPTIMIZATIONS



- Use PQ during querying
  - Avoids reading vectors for all neighbours
- Beam search:
  - Expand candidate list by $W > 1$.
- Cache vectors near entry point
  - Keep in RAM entry point neighbourhood
- Rerank using full precision
  - Load vector with its neighbourhood
  - No extra reads: 512B ≅ 4KB
  - Rerank when selecting best candidates





DISKANN QUERYING

- Greedy search with candidate list:
  1. Set $p \leftarrow$ nearest unvisited candidate to q
  2. Add p's neighbours to candidate list
  3. Add p to visited set
  4. Repeat

Prune to L candidates nearest q

- … with several optimizations!
  - PQ for distances (no need to load all vectors!)
  - Beam search: expand W candidates
  - Cache vectors near entry point (3-4 hops)
  - Load vectors with neighbourhoods, re-reank

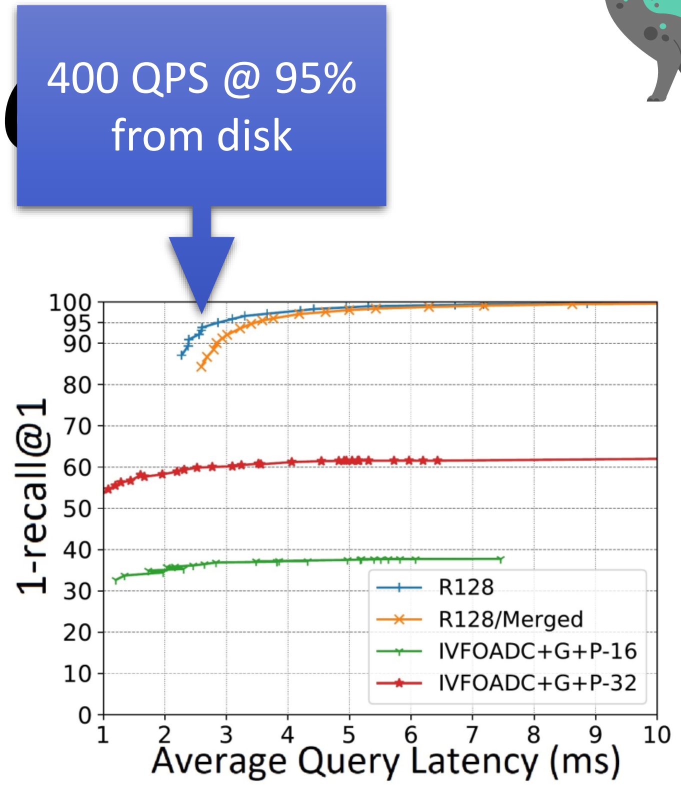
DISKANN PERFORMANCE

(on SIFT1B)

✓ Latency < 3ms @ 95% recall… **from SSD**
  - 2 x Samsung 960 EVO in RAID-0

✓ Much **better recall than composite index**
  - HNSW+IVF+OPQ
  - But not always as fast

- Build time:
  - **single:** 2 days on dual Xeon E7-8890v3s (32-vCPUs) with 1792GB
  - **merged:** 5 days on Dual Xeon E5-2620v4s (16 cores) 64GB
  - (Latency on merged index 4-5ms @ 95% recall)



DISKANN IN-MEMORY PERF

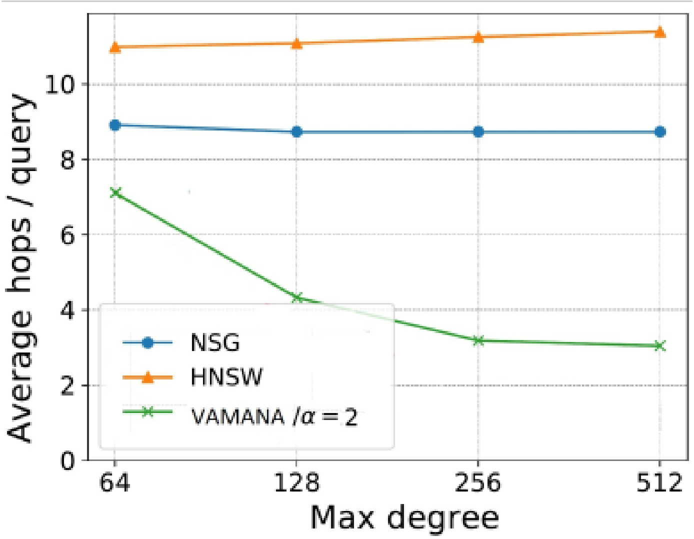
(on SIFT1M, GIST1M, DEEP1M<)

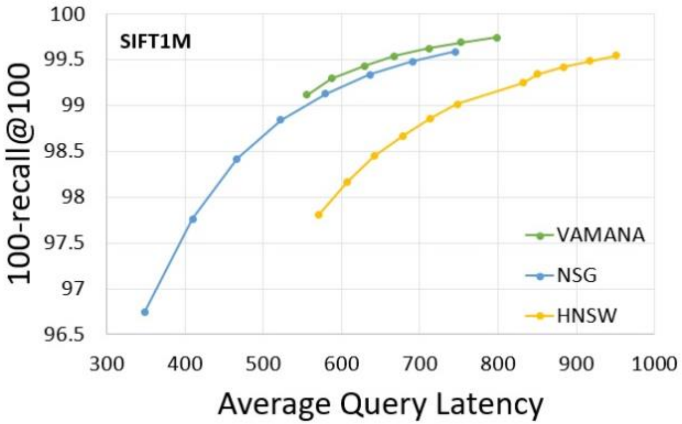
✓ Fewer hops than HNSW, NSG

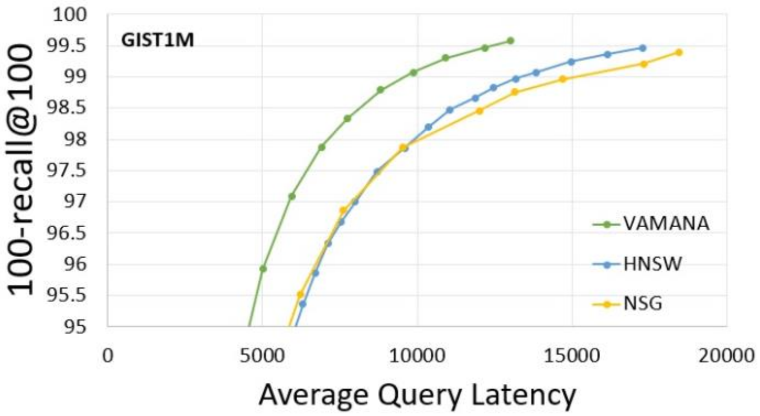
✓ Faster indexing, less memory

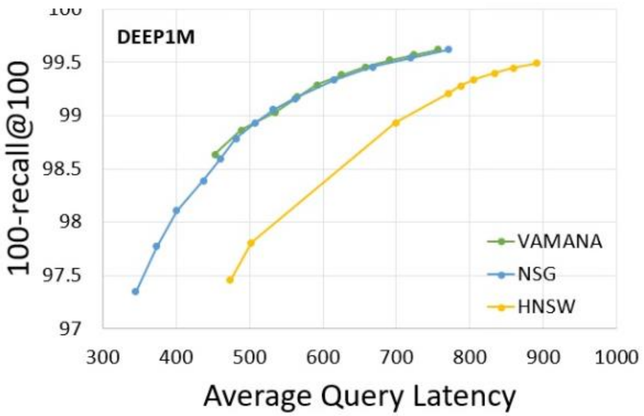
- Vamana: 149 sec , HNSW: 219 sec, NSG: 480 sec
- Dual Xeon E5-2620v4s (16 cores) w 64GB RAM

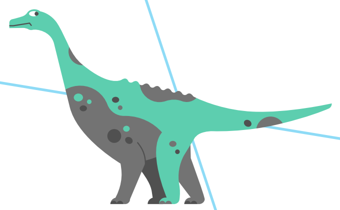
✓ Faster querying or fast as HNSW, NSG







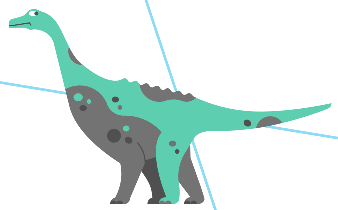




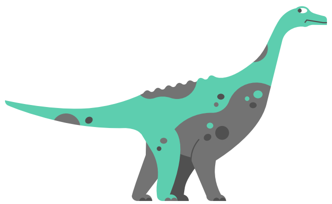
DISKANN DOWNSIDES

- × No delete, insert, update
- × Frequent rebuilds
- × Attributes and predicated queries?
- Work continues:
  - FreshDiskANN [Singh, arXiv ‘21] adds updates
  - Filtered-DiskANN [Gollapudi, WWW ‘23] adds filtering on attributes

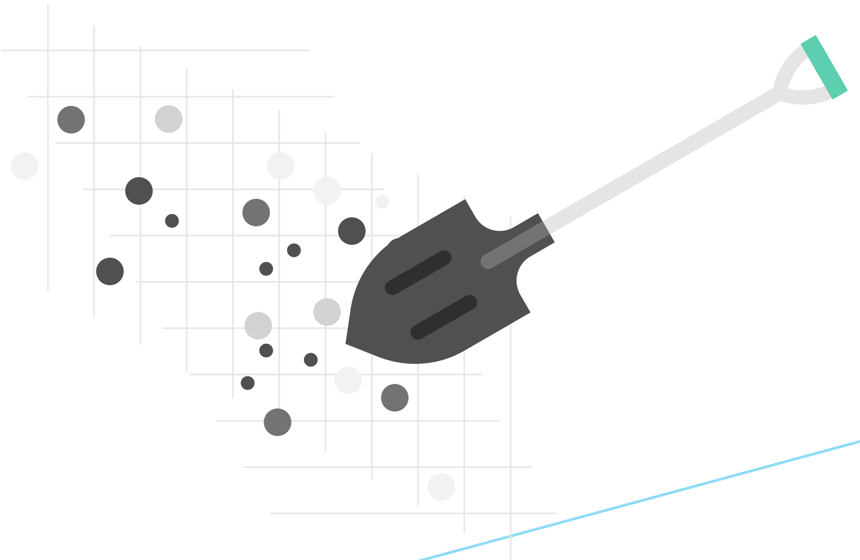
INTERIM SUMMARY



- Dealing with very large N
  - Sharding
  - Quantization (SQ8, PQ)
  - Composite index (IVF + PQ, IVF + HNSW + PQ)
  - Disk-resident index (ANNOY, DiskANN)

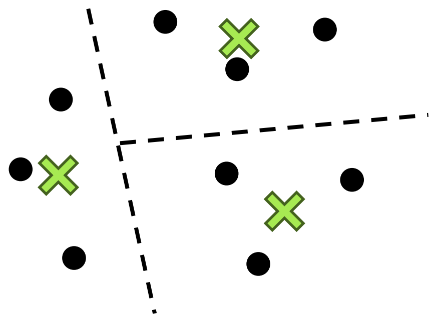


- Next, index rebuilds and freshness

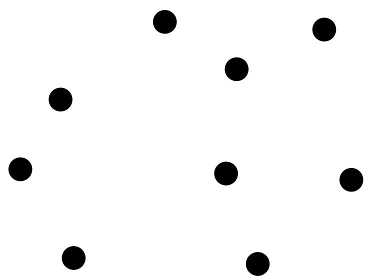


UPDATES DEGRADE INDEX

CLUSTER-BASED

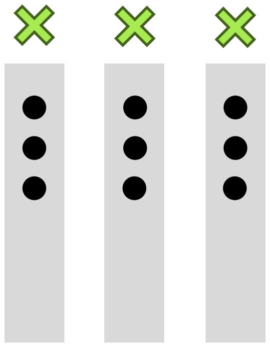


GRAPH-BASED

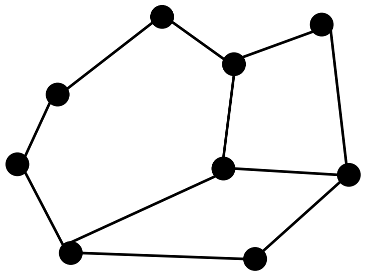


UPDATES DEGRADE INDEX

CLUSTER-BASED

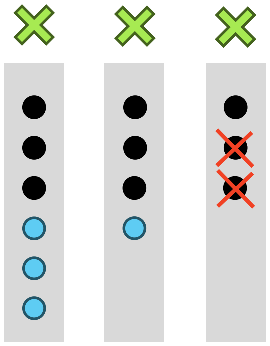


GRAPH-BASED



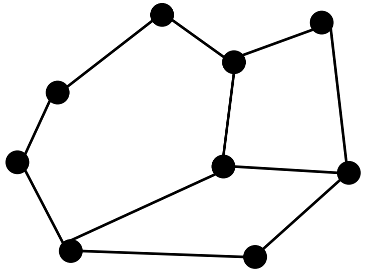
UPDATES DEGRADE INDEX

CLUSTER-BASED

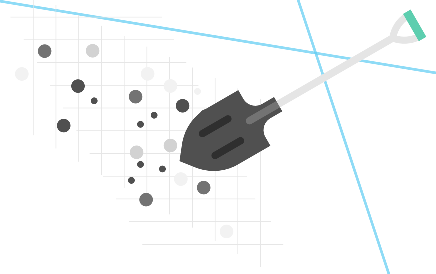


- Updates cause unbalanced partitions
- Large partitions → high latency
- Static centroids → low accuracy

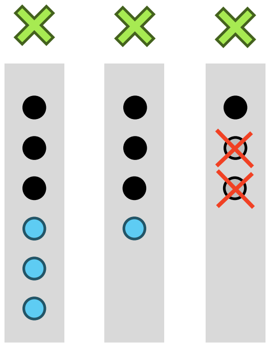
GRAPH-BASED



UPDATES DEGRADE INDEX

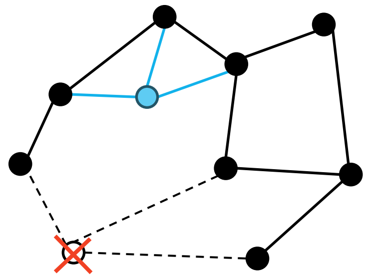


CLUSTER-BASED



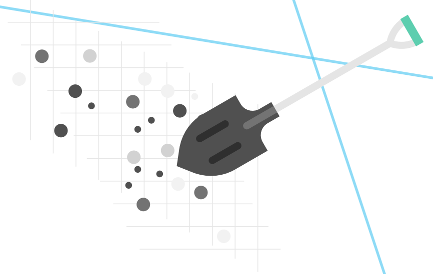
- Updates cause unbalanced partitions
- Large partitions → high latency
- Static centroids → low accuracy

GRAPH-BASED



- Update links during insert/delete?
- No → degrade recall, latency, memory
- Yes → very slow, resource intensive

WHAT TO DO?



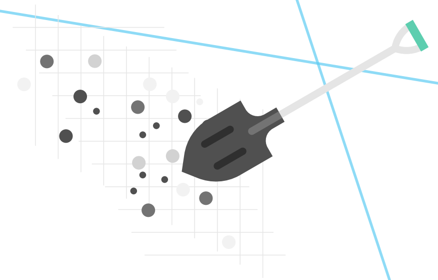
PROBLEMS

- Cannot update at all
  - E.g., DiskANN
- Degraded performance
  - E.g., IVF, HNSW
- Updates too slow
  - E.g., HNSW **x100** slower than query
- Data-dependent index
  - E.g., clustering in IVF, PQ

SOLUTION

- Out-of-place updates!
- Rebuild index periodically.
  - Use old index during build.
  - Switch to new when ready.
- Called **blue-green indexing**.
- Common in VDBMS
- … not perfect!

REBUILDS ARE A PROBLEM



- Rebuilds are **long** and **expensive**
  - Takes days.
  - Use extra resources (CPU, RAM, disk).
- In the meanwhile…
  - Degraded performance.
  - Stale query results.
  - Paying extra.
- Reduce staleness → **freshness layer**
- Avoid rebuilds → **segmenting updatable index**

- In-memory index:
  - N=1B, assume insert at 10K inserts/sec → 100K seconds = 1.1 days to rebuild
  - HNSWlib: N=100M, 48-core machine → 2 hours
- On-disk index with N=1B:

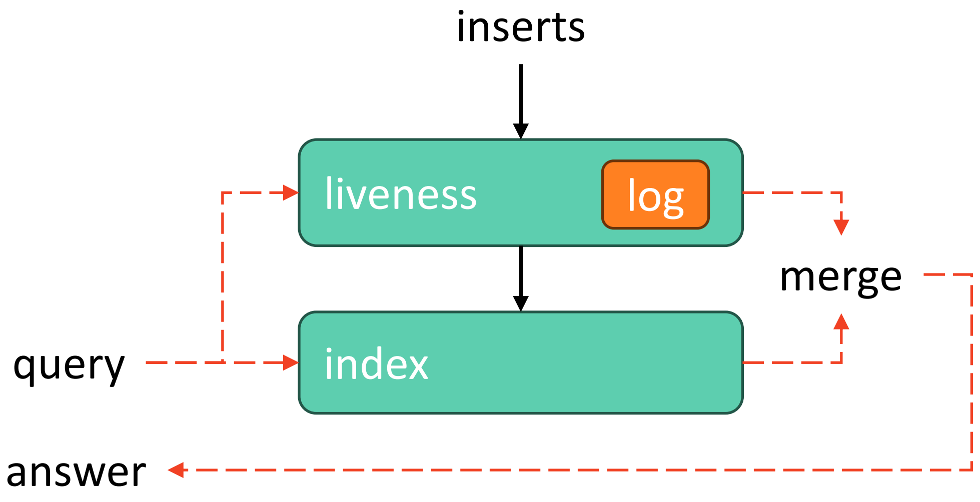
	Memory	CPU	Time
DiskANN	1100 GB	32 cores	2 days
	64GB	16 cores	5 days
SPANN	260 GB	45 cores	4 days

[Xu, SOSP'23]

5. FRESHNESS LAYER

(also called Secondary Index)

- Buffer incoming updates in memory
  - On-disk log for durability
  - Update/delete → mark tombstone
  - Maintain fast-to-update index (flat, IVF)
- When querying:
  - Query main index and buffer
  - Merge results
- Retire items:
  - Incrementally (if supported)
  - During periodic rebuild

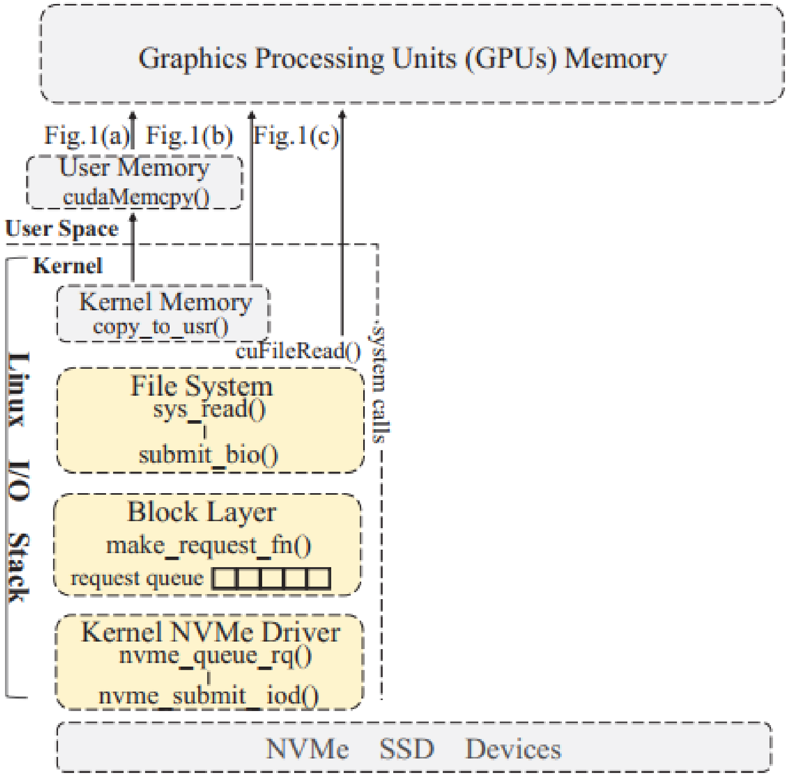


IMPLEMENTING FRESHNESS LAYERS

- Specific implementations vary
- General considerations:
  - Large memory cost
  - Maintaining consistency
  - Dealing with bursts
  - Extra IO
- Pinecone:
  - WAL + liveness layer
  - Secondary index
- Neos [Huang, ICDE'24]:
  - Stored on SSD
  - Flat index on GPU
  - Direct GPU-SSD access
  - LSM to access by ID
- Manu [Guo, VLDB'22]:
  - Piggy-back on distributed queue/WAL (Kafka/Pulsar)
  - IVF secondary index

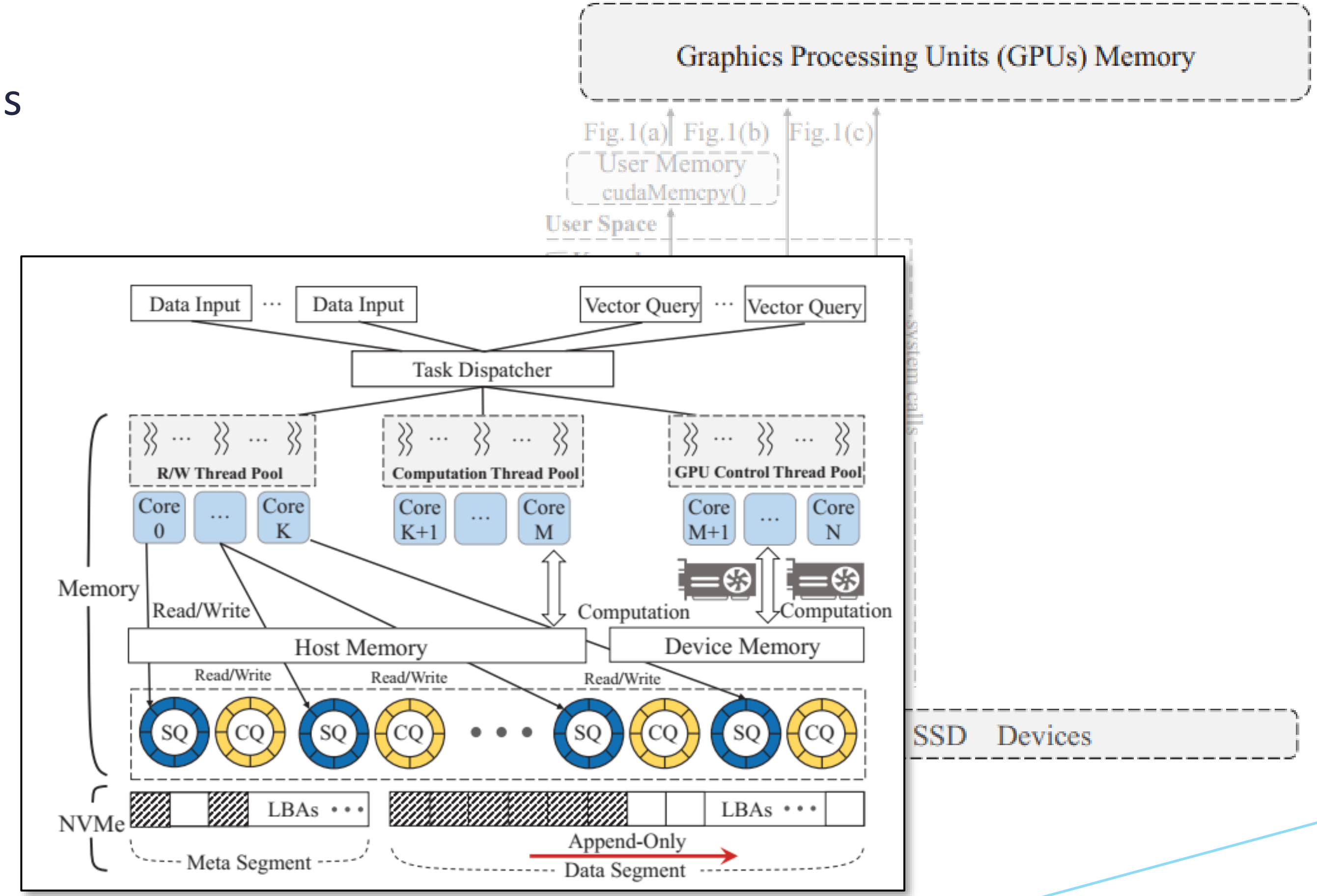
NEOS *[HUANG, ICDE'24]*

- Disk-resident freshness layer!
  - Real-time updates without index
- Problems with RAM-resident freshness:
  - Need secondary index for search
  - Index writes x100 – x1000 slower → can't keep up
  - Secondary index grows big → lots of RAM
- Obstacles to addressing:
  - Search on CPU too slow
  - Traditional GPU I/O too slow
  - Complex storage structure constraints P2P I/O



NEOS IDEAS

1. Replace index with GPUs
   - Brute force search on multi-GPUs
   - Fed from SSD
2. Bypass storage stack entirely:
   - Simple on-SSD structure
   - Direct NVMe → GPU copy
   - Pinned GPU memory + SPDK
3. Task scheduler
   - Isolate search vs write I/O
   - Load balancing
   - Predict task time to avoid sync overhead



NEOS PERFORMANCE

- Setup
  - 4x NVIDIA V100 GPUs
  - Intel Optane DC P5800X (extremely fast SSD, 2K$)

Intel P5800X:
- Limited capacity
- 1.5M IOPS
- 7.2 GBPS
- 5 us P99 latency (random 4K read)
- 3D XPoint discontinued

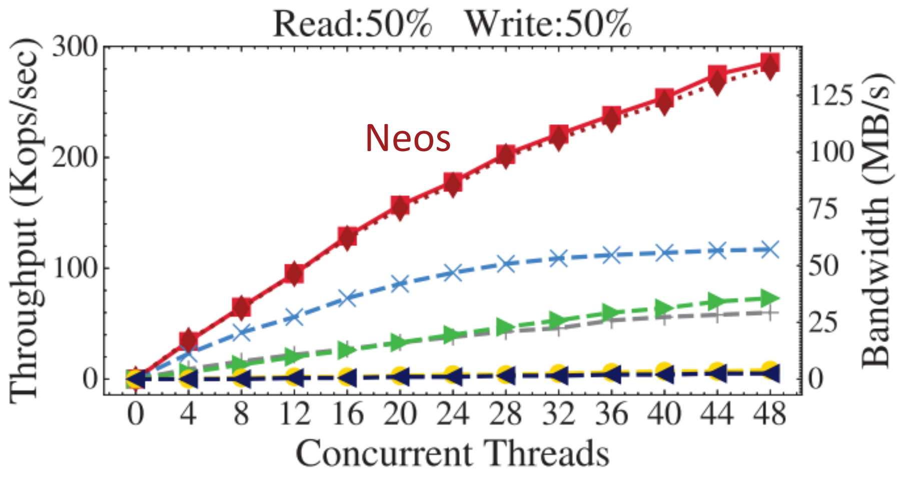
Intel

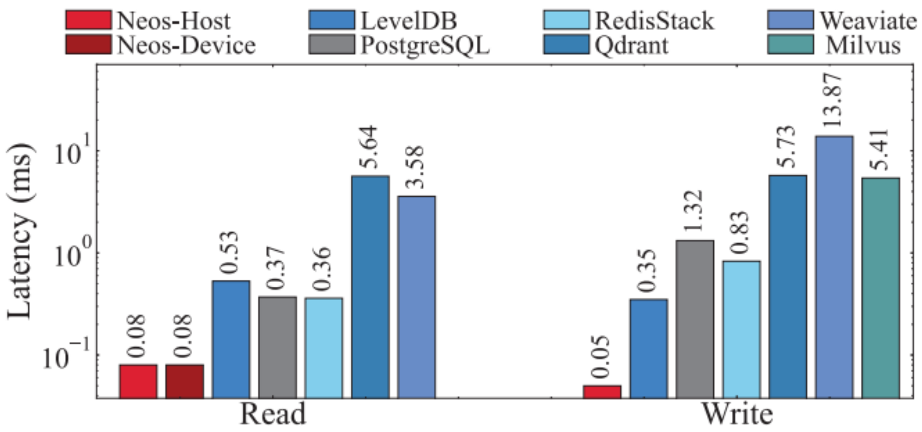
Not coming to a DC near you!

NEOS RAW PERFORMANCE

- Setup
  - 4x NVIDIA V100 GPUs
  - Intel Optane DC P5800X (extremely fast SSD, 2K$)

✓ **Strong raw performance**
  - 50-80 micro-sec latency
  - Excellent scaling
  - These are not queries! (get by key, not kNN)





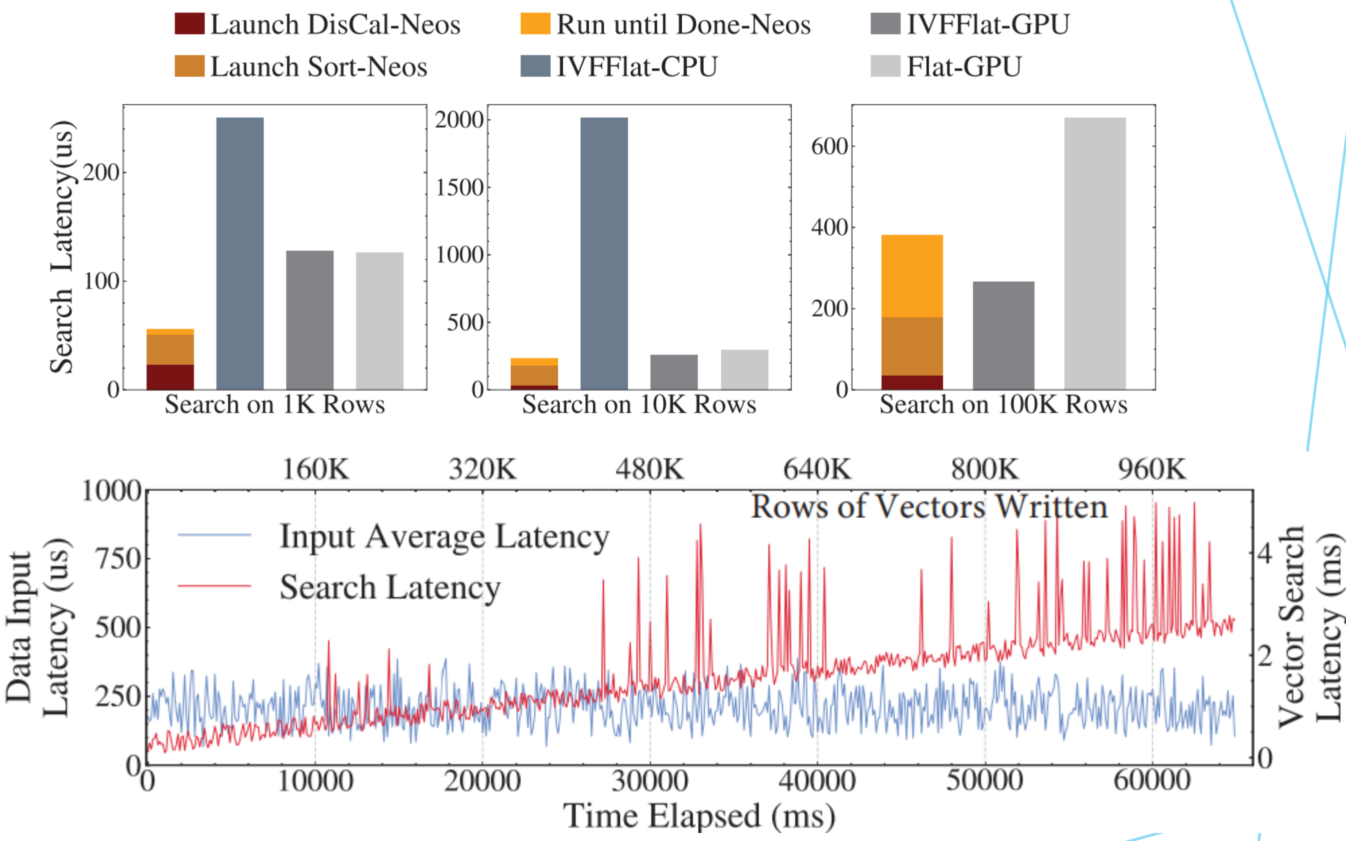
NEOS QUERY PERFORMANCE

✓ **Competitive kNN performance**:

- Recall tuned to > 95%
- *N* < 10K: faster than IVF
- *N* = 100K: slightly slower
- IVF → fast inserts, common
- Unlike IVF: no rebuilds, degradation

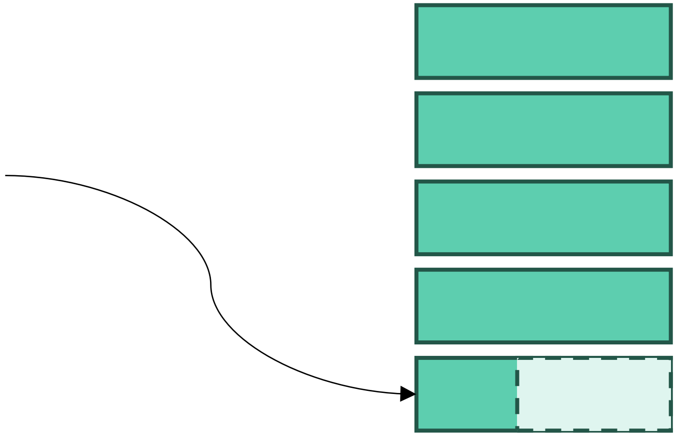
✓ **Fast with mixed workload:**

- Insert *N* = 1M vectors
- 1:2 inserts-to-queries
- Insert latency stable 100us-400us
- Query latency grows < 4ms for



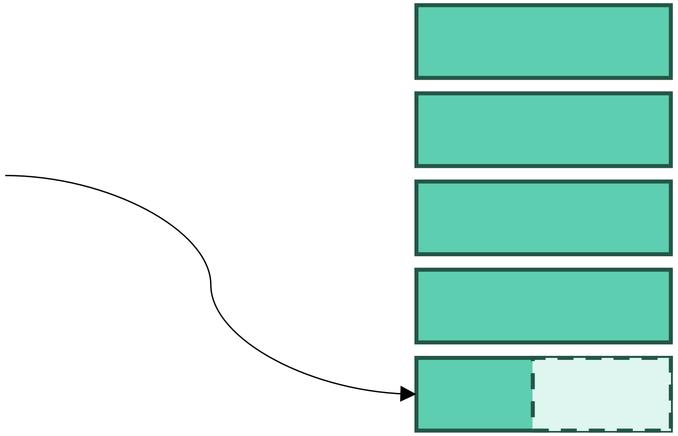
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment



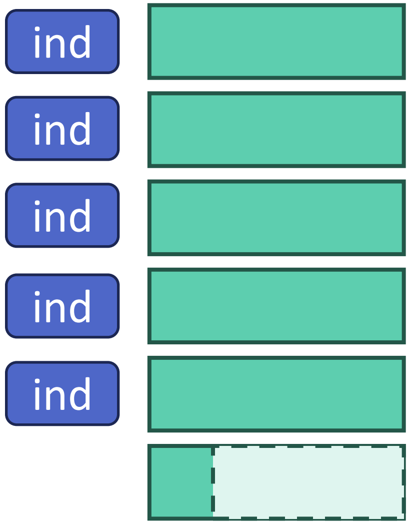
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment



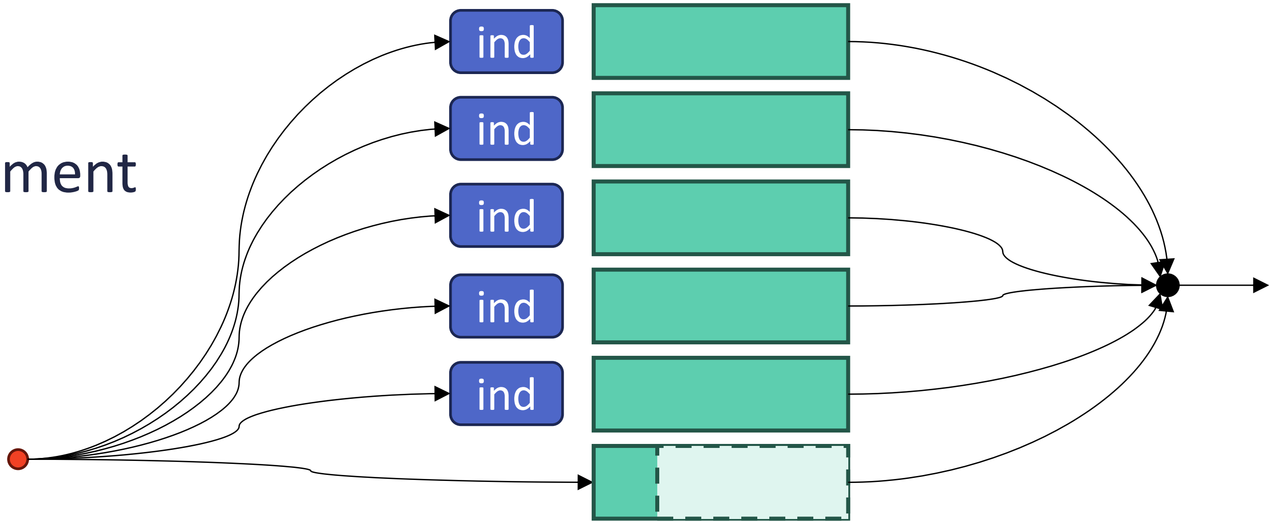
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment



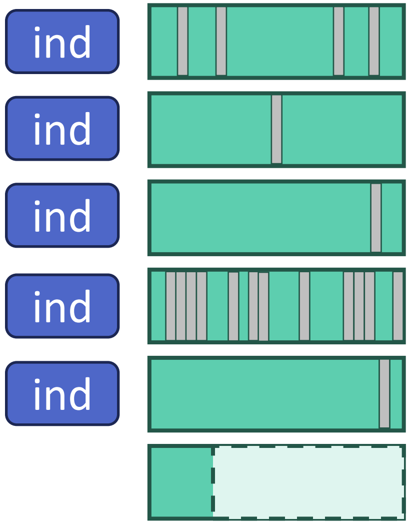
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment
- Query all segments, combine



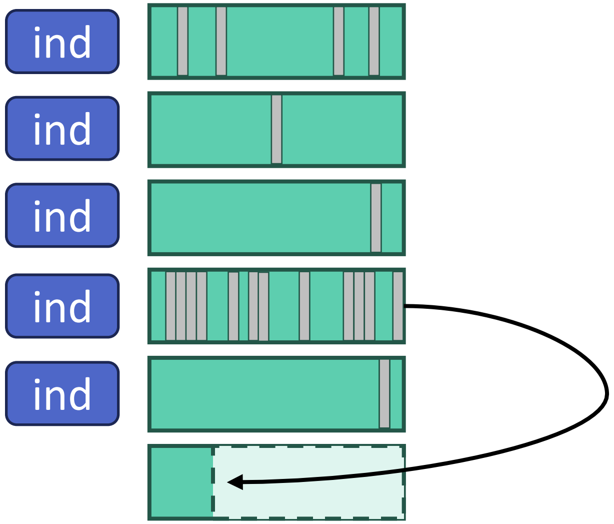
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment
- Query all segments, combine
- Mark deleted vectors (tombstones)



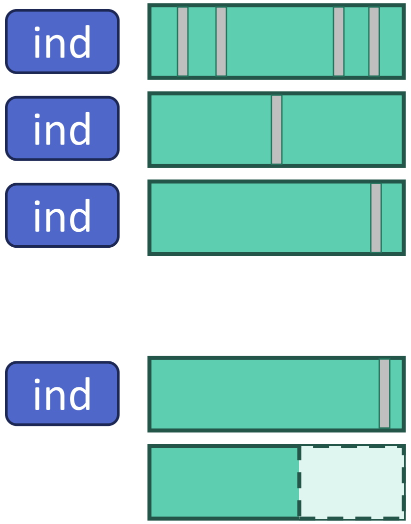
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment
- Query all segments, combine
- Mark deleted vectors (tombstones)
  - Merge mostly-empty segments



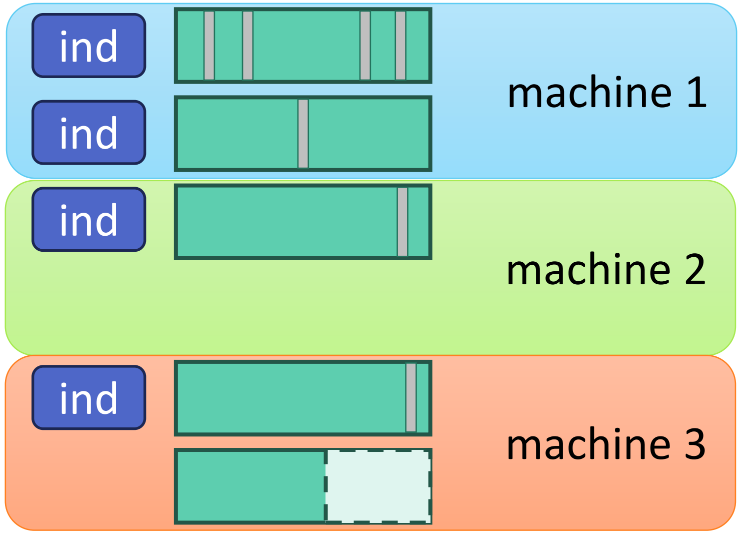
6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment
- Query all segments, combine
- Mark deleted vectors (tombstones)
  - Merge mostly-empty segments



6. SEGMENTING

- Split collection to **segments**
  - Example 1M vector/seg
- Insert: append to growing segment
- Index segment when full
  - Open new growing segment
- Query all segments, combine
- Mark deleted vectors (tombstones)
  - Merge mostly-empty segments
- Distribute segments to parallelize index, querying



6. SEGMENTING BENEFITS

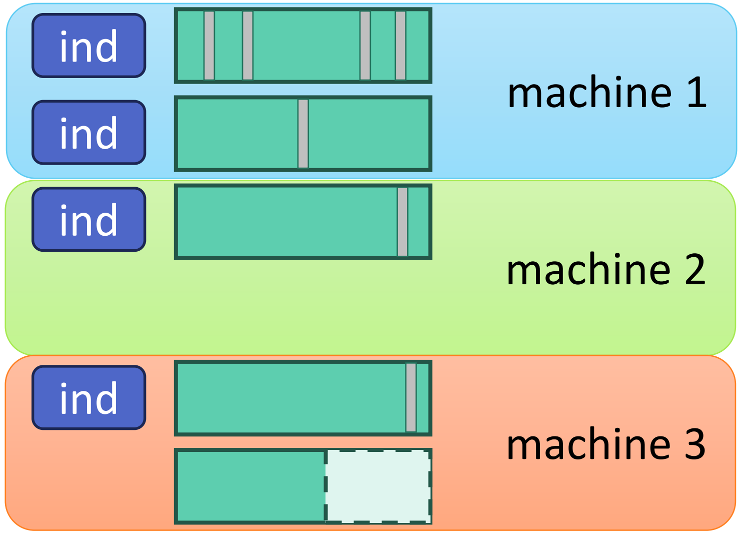
✓ **No more rebuilds**
  - Segments are static
  - Build on full segment, on merge

✓ Each index is small

✓ Growing segment = freshness layer

✓ Easy to distribute work
  - Example: allocate segments to shards

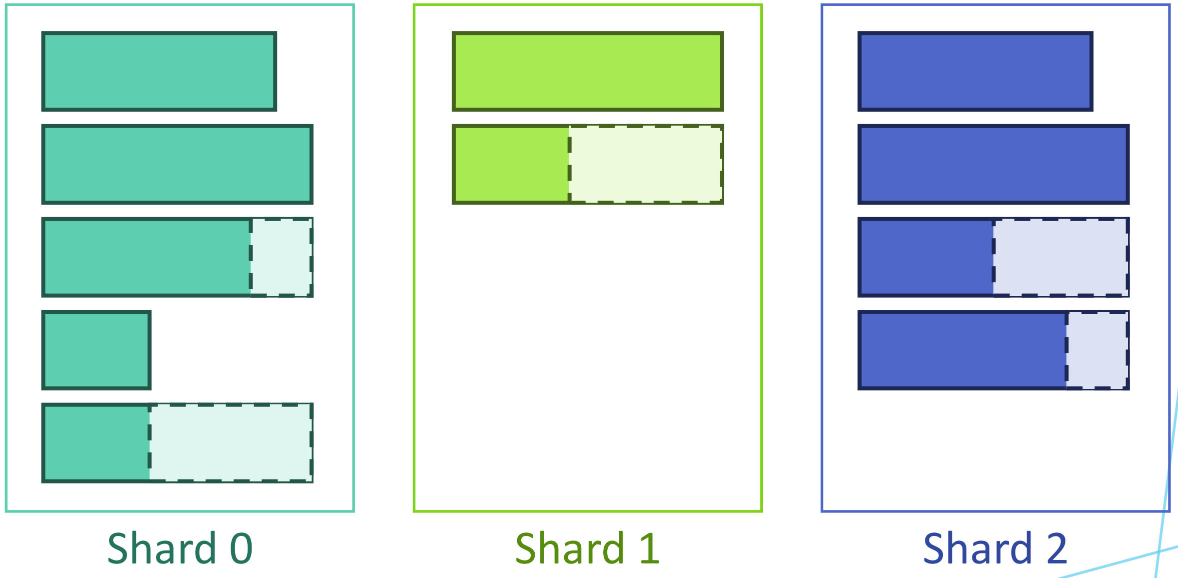
- Downsides:
  - Must query **all** segments
  - Write amplification if update-heavy



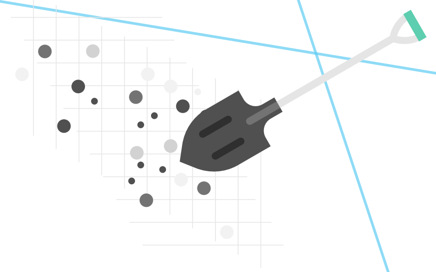
Used in many VecDBs! (e.g., Milvus, Qdrant)

6. SEGMENTING THOUGHTS

- Segmenting ≠ sharding
  - Sharding: distribute data across machines
  - Segmenting: avoid reindexing, accommodate growth
- Work well together
  - Shard by key and segment each shard
  - Qdrant, Milvus
- Other perspectives:
  - Sharding – insert/write performance
    Segmenting – query performance
  - When adding data →
    num shards fixed, shards grow
    num segments grows, segments do not

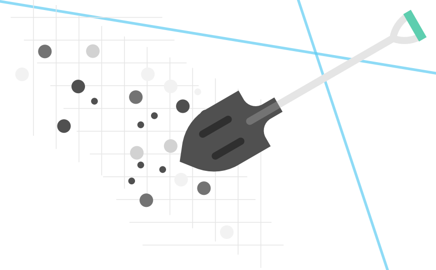


7. UPDATABLE INDEXES



- Avoid rebuilding!
- Different approaches
  - Re-balancing
  - In-place updates
  - Data-independent index
- Especially for disk-resident
  - SPFresh – cluster-based, on-disk index without rebuilding [Xu, SOSP’23]
  - FreshDiskANN – graph-based on disk-index [Singh, arXiv ‘21]
- **Active research area** (we shall see several)

FRESHDISKANN *[Singh, arXiv '21]*



GOAL

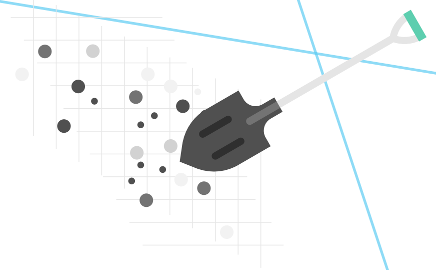
- Support:
  - 1B vectors
  - > 1K delete/updates/inserts per second
  - > 1K searches per second
  - 95% 5-recall@5
  - Realtime freshness
- …on single machine:
  - 48-cores
  - 2TB SSD
  - 128GB RAM

HOW

- "Long-term" SSD index (DiskANN-like)
- In-memory index (freshness layer)
  - Insert list and delete list
  - Periodically merged to disk (every 30M updates)
- Write-optimized merge algorithm
  - Merges in-memory into disk:

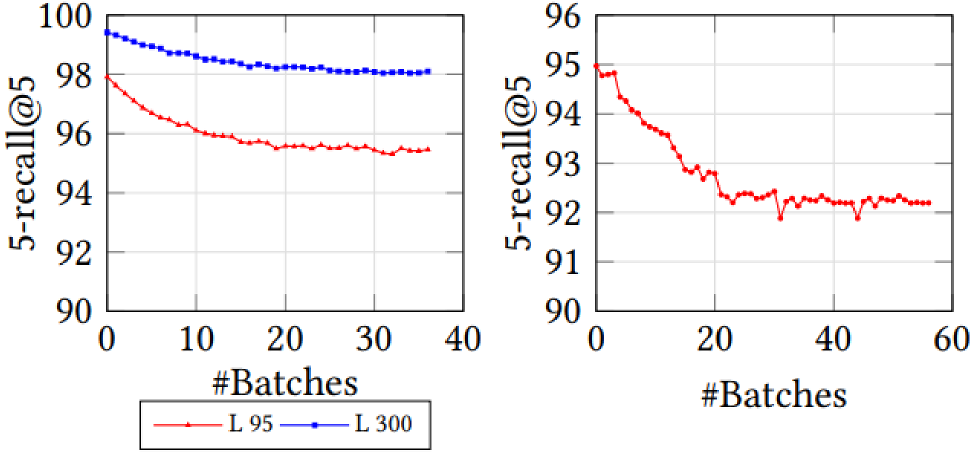
    1. Delete block-by-block: reconnect nodes, prune
    2. Insert: add edges to in-memory patch buffer
    3. Patch block-by-block: apply patch, prune

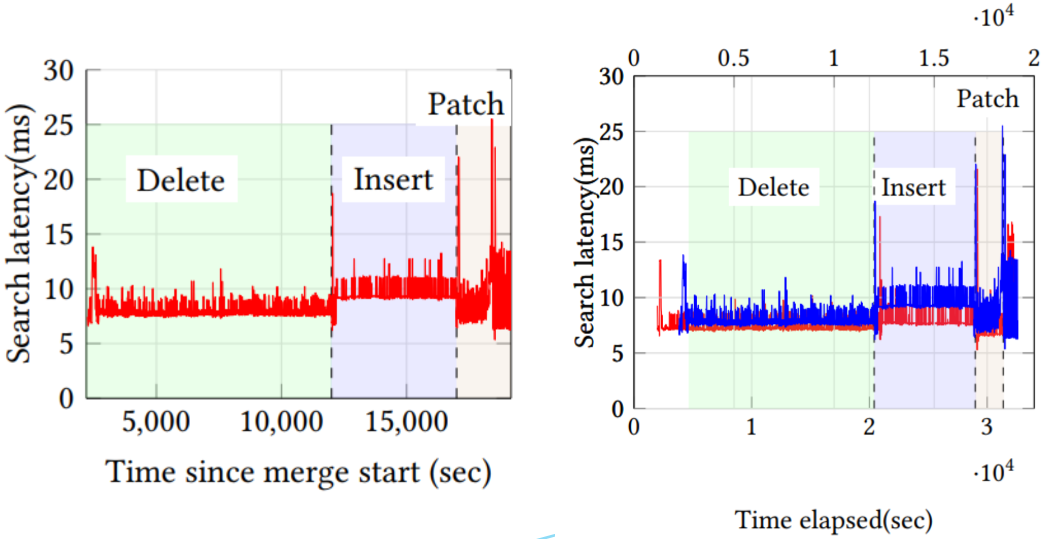
    → Cost: $O(\#\text{updates})$



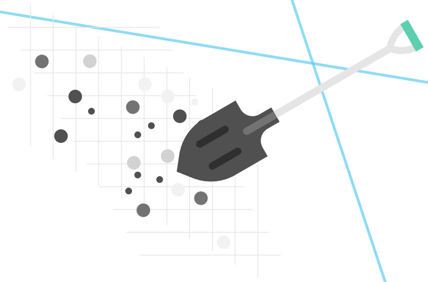
FRESHDISKANN RESULTS

- Fast inserts/deletes
  - 1.8K/s inserts + 1.8K/s deletes (sustained)
  - 40K/s burst
  - < 1ms during merge
- Decent search performance:
  - 1K/s queries
  - 95% recall@5
  - 20ms avg latency
- Recall stable long-term
- < 10% cost of rebuild
- Higher mean latency during merge
  → tail latency likely explodes



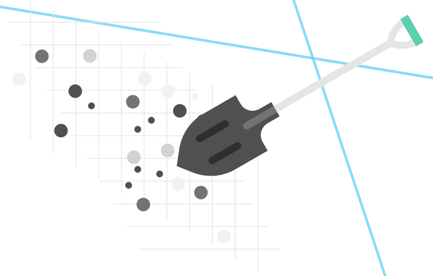


SPFRESH *[Xu, SOSP'23]*



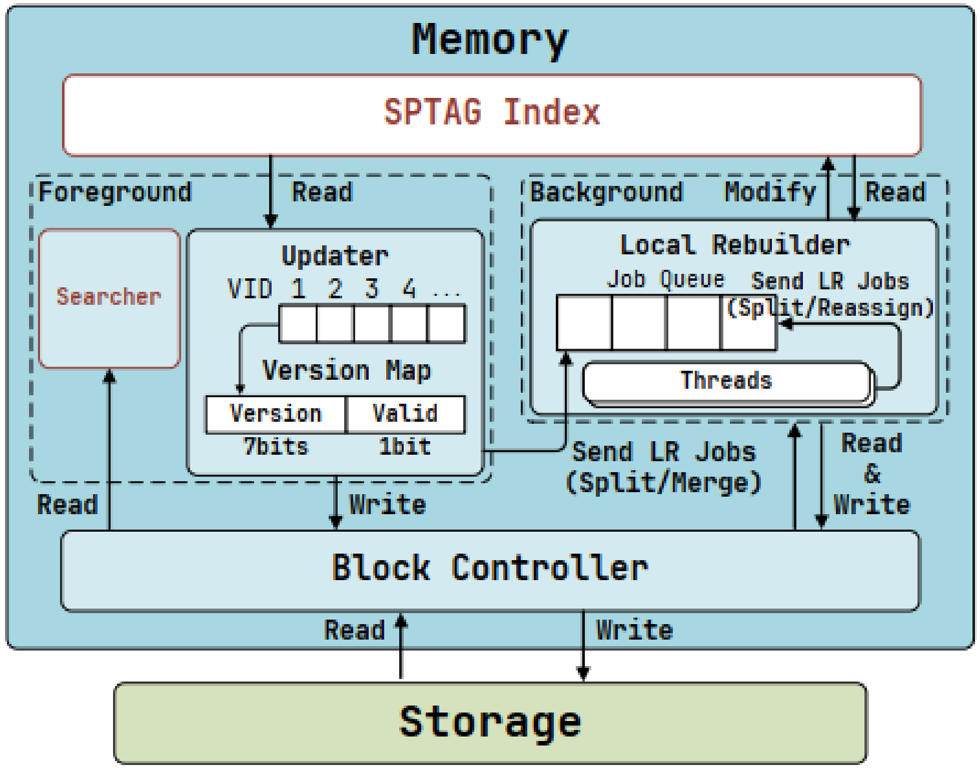
- Composite: cluster-based index + graph-based index for centroids
- Idea: small updates + well-balanced index = local changes.
- LIRE protocol:
  - Maintains uniform size (by splitting, merging clusters)
  - Small, local adjustments (by reassigning few vectors)
  - Fast updates (delay split/reassign for later)
  - Avoid global rebuilds
- SPFresh system:
  - SSD backend reuses SPANN and SPTAG [Chen, NeurIPS'21]
  - Prioritize reads, fast appends
  - Delay rebuilds.

SPFRESH: SOME POINTS



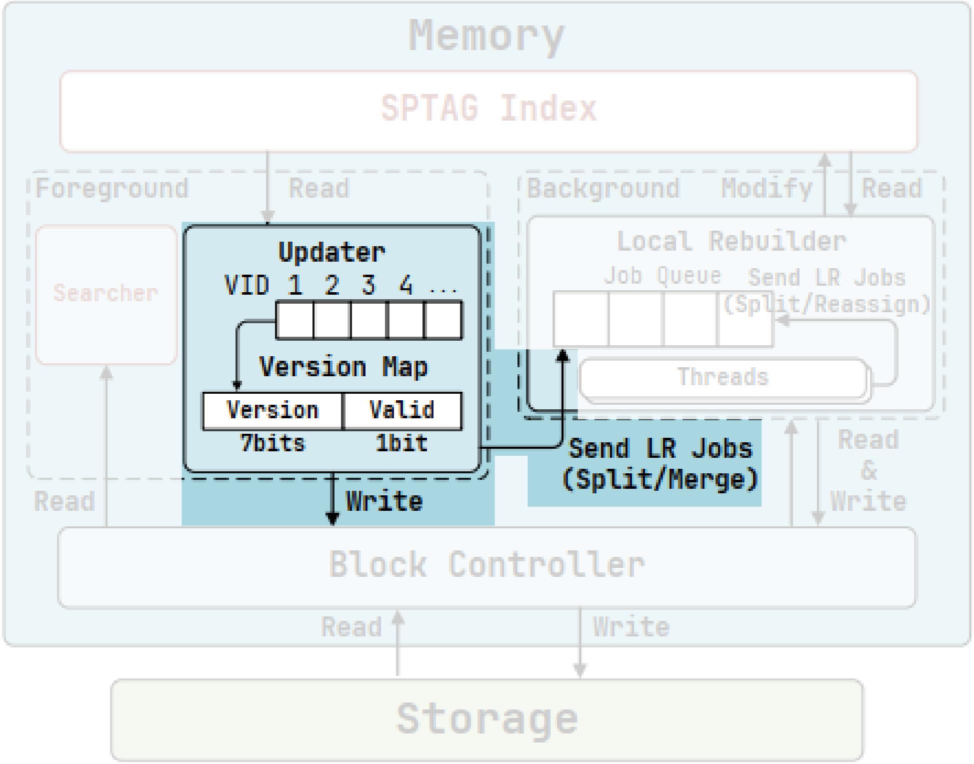
- Updates are tricky:
  - Split + merge → centroids move → must reassign vectors.
  - Reassign → unbalanced partitions → split and merge
  - Cascade: Reassign → split & merge → reassign → split & merge
- Algorithmic details:
  1. Identifying small set of vectors to reassign
  2. Reassign beyond split or merged partition
  3. Proof that cascade converges (but given bound is *trivial:* **#splits < N**)
- **Key tricks/optimizations are systems!**

SPFRESH: ARCHITECTURE

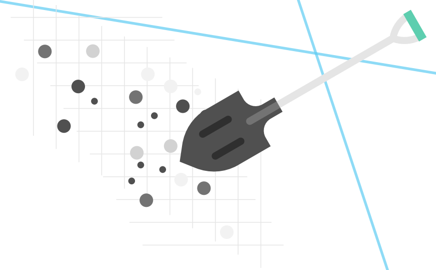


SPFRESH: ARCHITECTURE

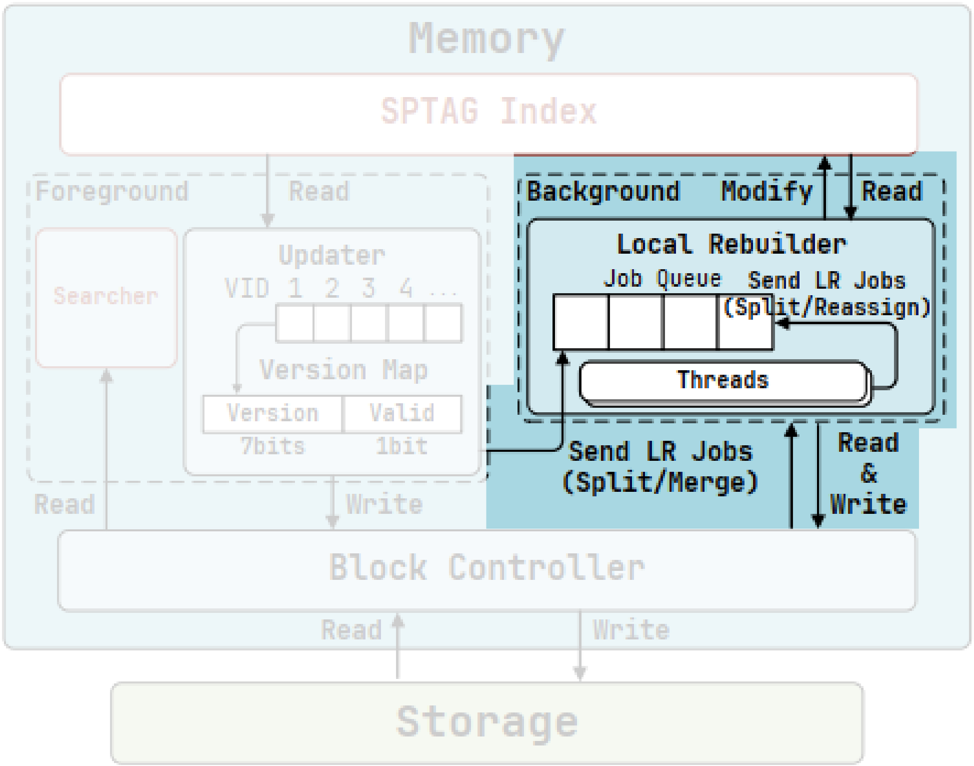
- Fast *updater*:
  - Append vector at end.
  - Version tag identifies stale data.



SPFRESH: ARCHITECTURE

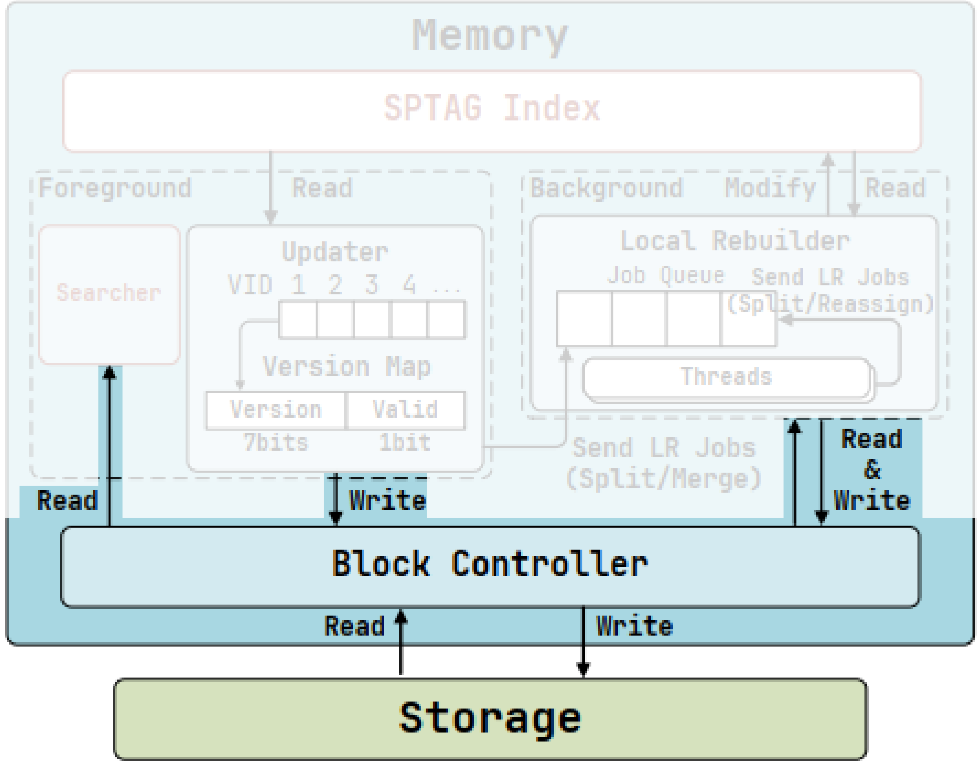


- Fast *updater*:
  - Append vector at end.
  - Version tag identifies stale data.
- Multithreaded *rebuilder*:
  - Run split/merge/reassign.
  - Scheduled by inserts, delete, queries.
  - Garbage collects during split.
  - Careful concurrency control.



SPFRESH: SYSTEMS TRICKS

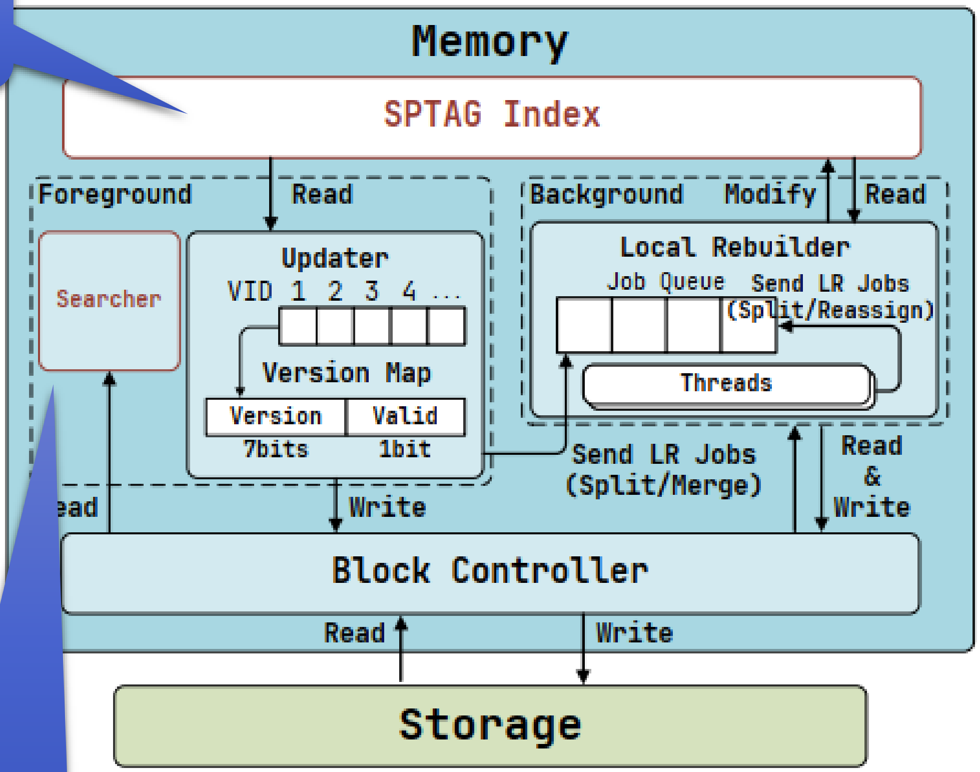
- Fast *updater*:
  - Append vector at end.
  - Version tag identifies stale data.
- Multithreaded *rebuilder*:
  - Run split/merge/reassign.
  - Scheduled by inserts, delete, queries.
  - Garbage collects during split.
  - Careful concurrency control.
- Block (storage) *controller*:
  - Controls SSD storage.
  - SPDK to bypass storage stack.
  - Append-only disk layout.



SPFRESH: SYSTEMS TRICKS

- Fast *updater*:
  - Append vector at end.
  - Version tag identifies stale data.
- Multithreaded *rebuilder*:
  - Run split/merge/reassign.
  - Scheduled by inserts, delete, queries.
  - Garbage collects during split.
  - Careful concurrency control.
- Block (storage) *controller*:
  - Controls SSD storage.
  - SPDK to bypass storage stack.
  - Append-only disk layout.

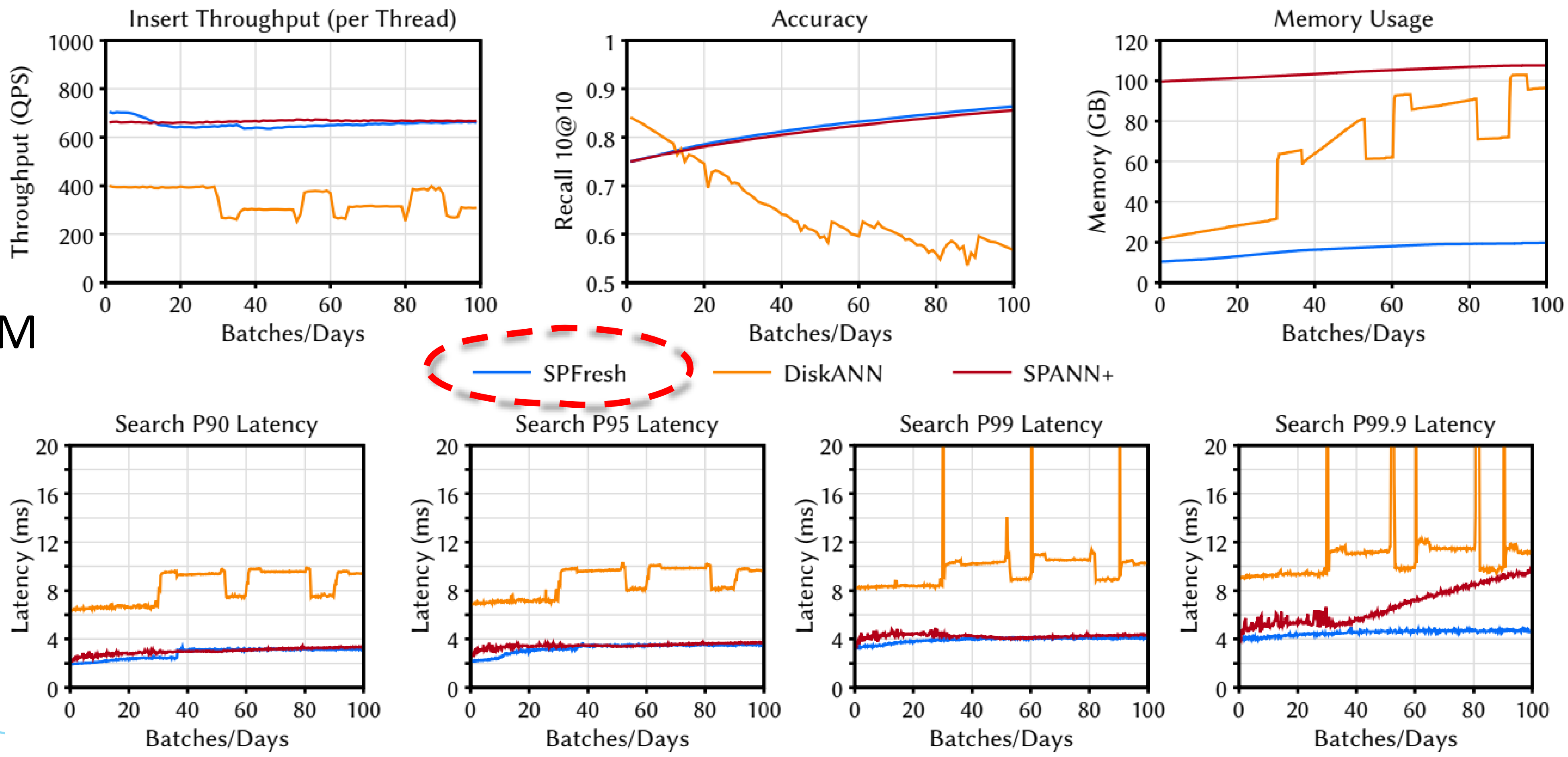
Graph-based index for cluster centroids



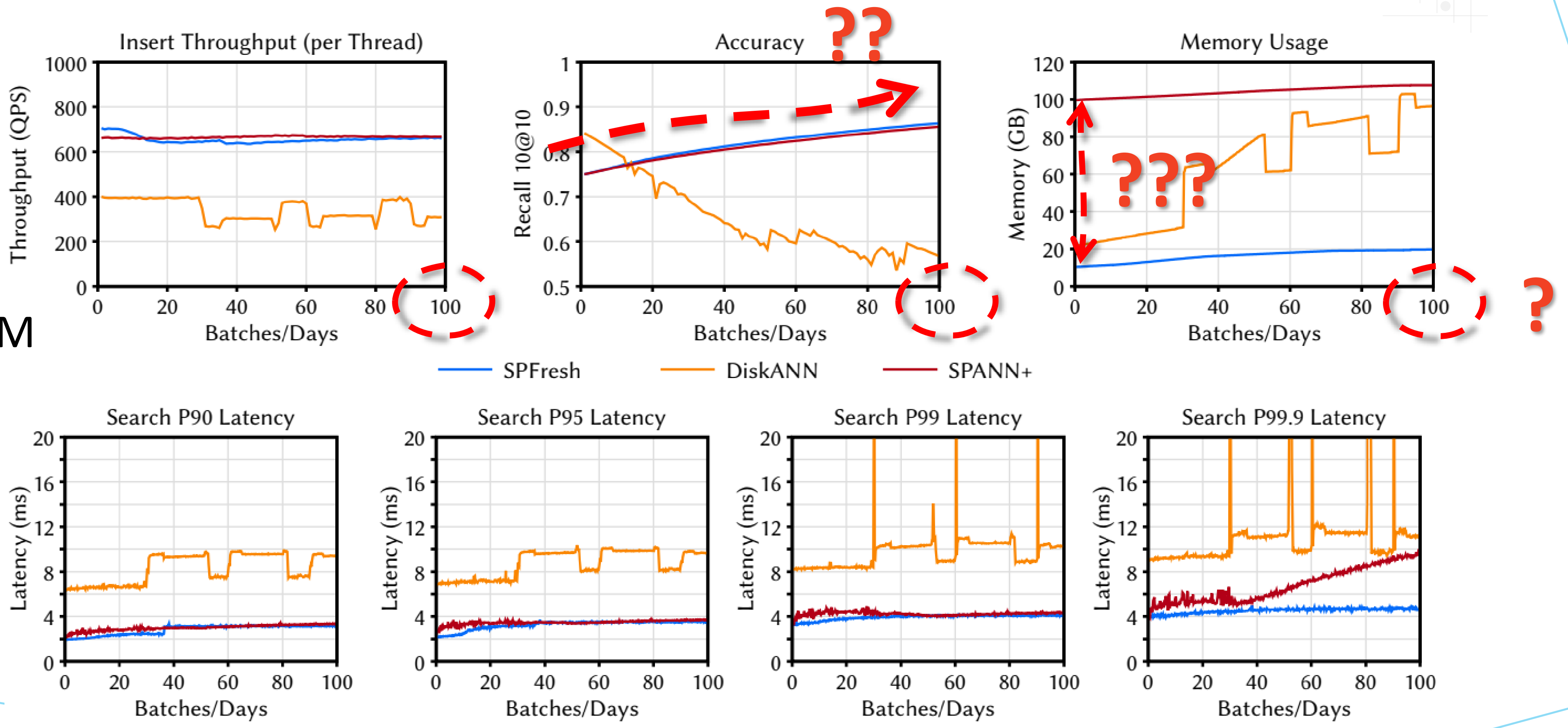
Lock-free search

SPFRESH: STABLE PERFORMANCE

N = 100M
D = 100

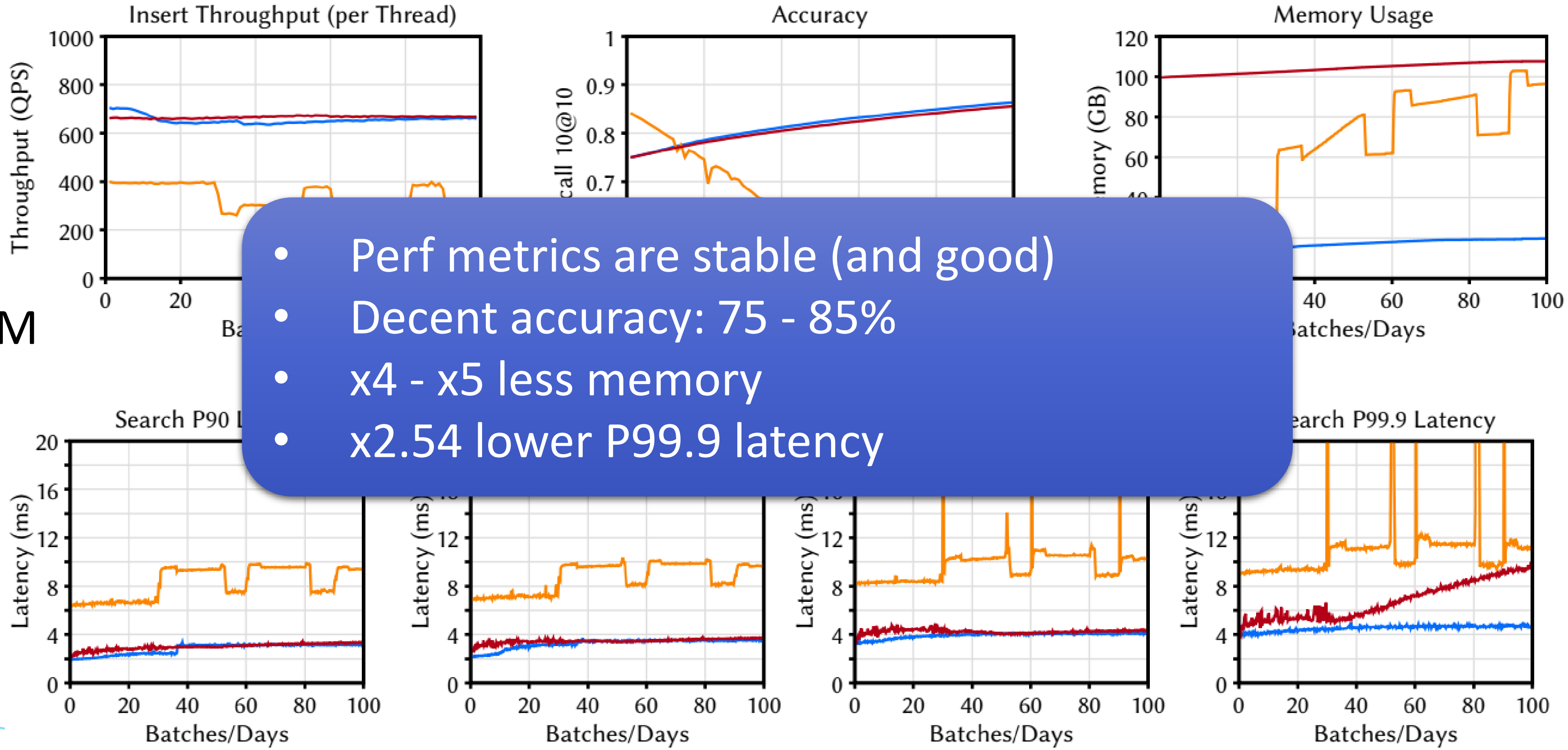


SPFRESH: ODDNESS



N = 100M
D = 100

SPFRESH: STABLE PERFORMANCE



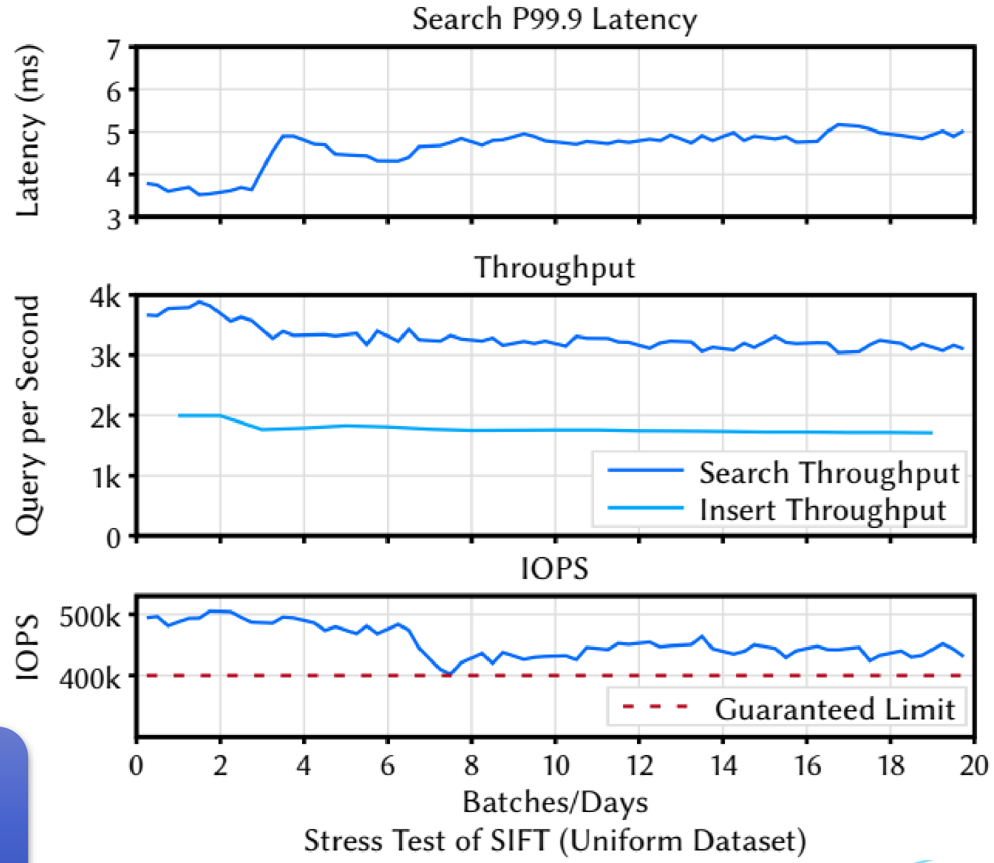
- Perf metrics are stable (and good)
- Decent accuracy: 75 - 85%
- x4 - x5 less memory
- x2.54 lower P99.9 latency

N = 100M
D = 100

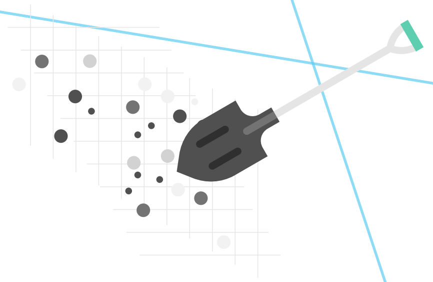
SPFRESH: SIFT1B DATASET

- Setup: (FreshDiskANN)
  - 16-core machine — 48 cores
  - 1% daily inserts — 5% ins + 5% del
- Almost 2K insert/sec — **1.8K + 1.8K**
- Over 3K queries/sec — **1K**
- Accuracy > 0.86 — **>95%**
- 5ms latency (P99.9) — **20ms (avg)**
- Peak memory: 74GB — **<128 GB**
- Stable, saturates SSD.

The new benchmark?



SPFRESH: PROBLEMS



- "Loose" conditions for reassignment
  - Not very selective
- NPA not maintained!
  - Ignored violations during merge
  - On split: only check few clusters
- Bound on cascading splits is…
  1. **Impractical**: upper-bounded by *N*
  2. **Wrong**: assumes no NPA violations, but there are.
- Writes not aligned with SSD erase block
  - Causes more write-amplification?
- Skewed data → imbalanced clusters
  - Can't really fix this!
  - May cause constant swaps, bad recall
- Experimental issues:
  - Odd empirical results.
  - Runs maybe too short?
  - No ablation?
- Potentially hard to distribute
  - Unlike SPANN

SUMMARY

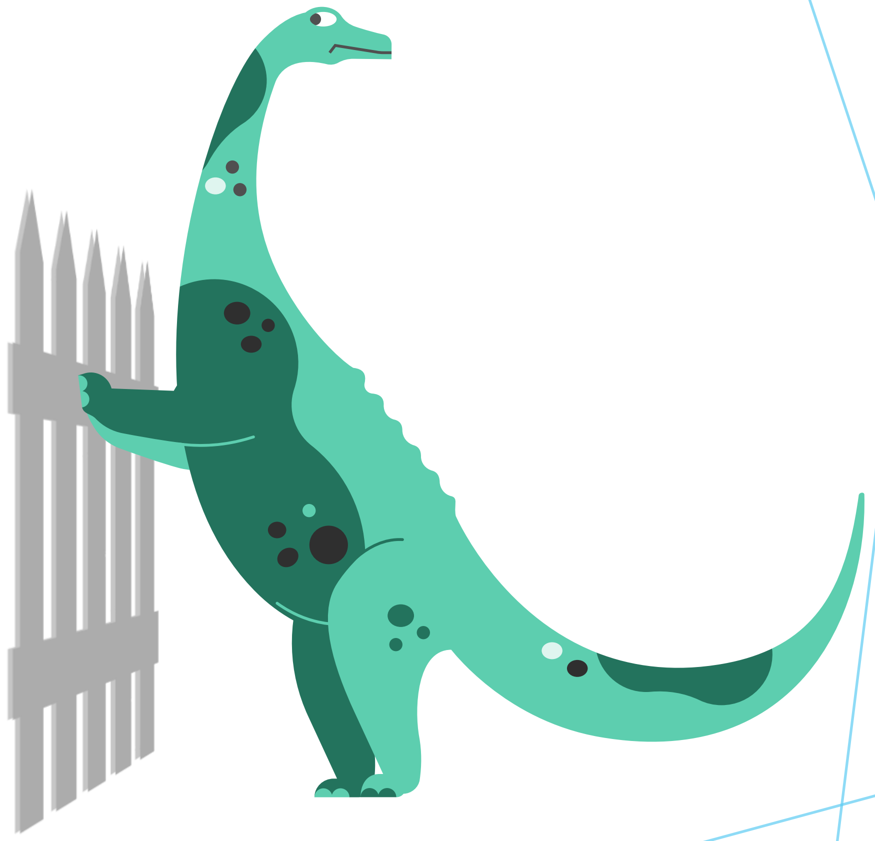
- Indexes govern VDBMS capabilities:
  - Cluster-based: Flat, IVF
  - Graph-based: HNSW
  - Others: LSH, tree-based
- Considerations:
  - Performance: recall/latency/memory
  - Very large collections
  - Rebuilds and updates
- Techniques
  - Quantization: SQ, PQ
  - Composite
  - Liveness layer
  - Sharding
- Most modern SotA indexes are **graph-based**
  - HNSW (custom variants, implementations)
  - Composite (graph + IVF + quantization)
  - Sometimes disk-resident

RESEARCHING INDEXES

- High-quality implementations:
  - FAISS –most indexes, composite, GPU
  - ANNOY
  - HNSWlib
  - DiskANN (incl. Fresh-…, Filtered-… variants)
  - …
- Common datasets:
  - SIFT1M, SIFT1B
  - GIST1M
  - DEEP1B
  - GloVe
  - …
- Comparisons:
  - https://ann-benchmarks.com/
  - https://github.com/erikbern/ann-benchmarks
  - Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search
  - Recent Approaches and Trends in Approximate Nearest Neighbor Search
  - Above comparisons also contain links to implementations, datasets

OPEN PROBLEMS

- NeurIPS'21 challenge [Simhadri, PMLR'22]:
  - Better support for predicated & multi-vector queries.
  - Stable, robust updates (insert, delete, update)
  - Out-of-distribution queries
  - Compression with higher recall
- Recent survey [Pan, VLDBJ '24]:
  - Score design, selection
  - Index design: disk, updates, concurrency
  - ~~Incremental kNN (retrieve next neighbours)~~
  - Security, privacy, federated search



NEXT

- Indexes are **not** everything!
  - Liveness
  - Storage
  - Multitenancy
  - Garbage (tombstone) collection
  - Retrieval (query optimization, planning)
  - Access layer
  - Fault tolerance
  - Access control
- **… but cannot cover everything.**

- Next session: **VDBMS architectures**
  - Classic: Vearch [Li, Middleware'18]
  - Modern: Manu [Guo, VLDB'22]