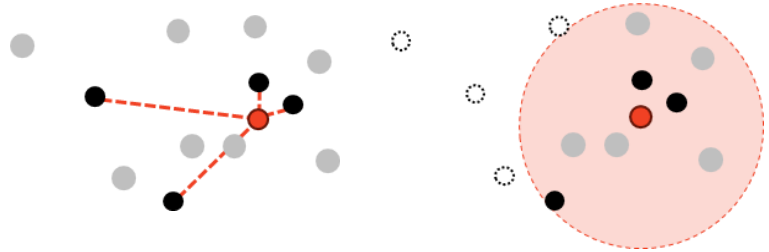
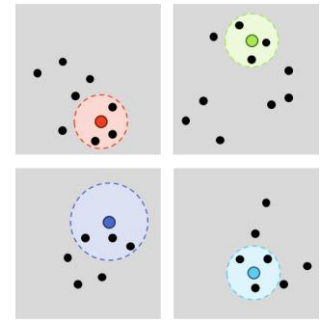
IV. ADVANCED INDEXING

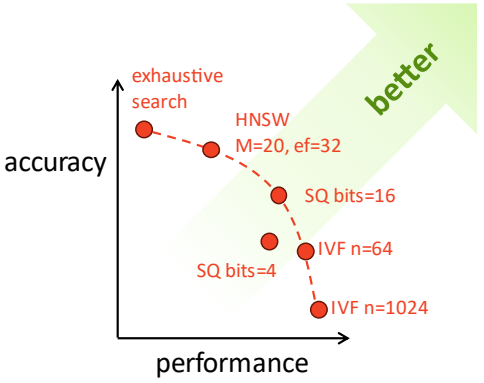
PREVIOUSLY, ON

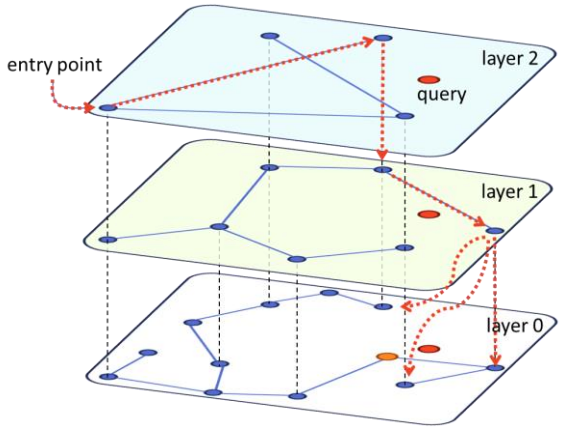
TOPICS IN VECTOR DATABASES

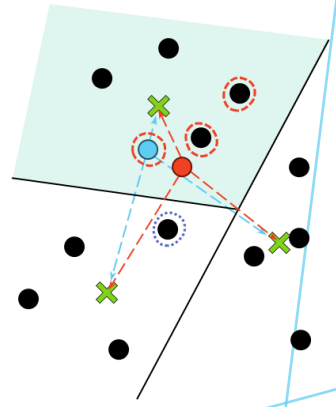
- Queries:
  - kNN
  - Filtered queries.
    - Prefilter / postfilter / single-stage
  - Multi-vector queries (and challenges)
  - Reranking
  - The need to index
- Index:
  - Tradeoffs and recall.
  - Flat index (for <100K vectors)
  - LSH (elegant but suboptimal)
  - IVF (cluster-based index)
  - HNSW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





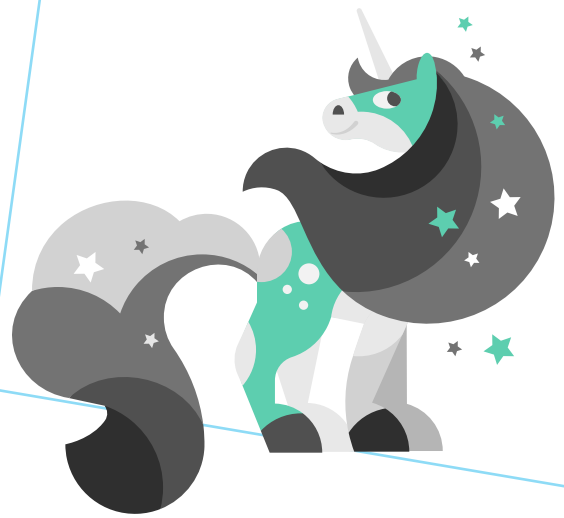






AND NOW…

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



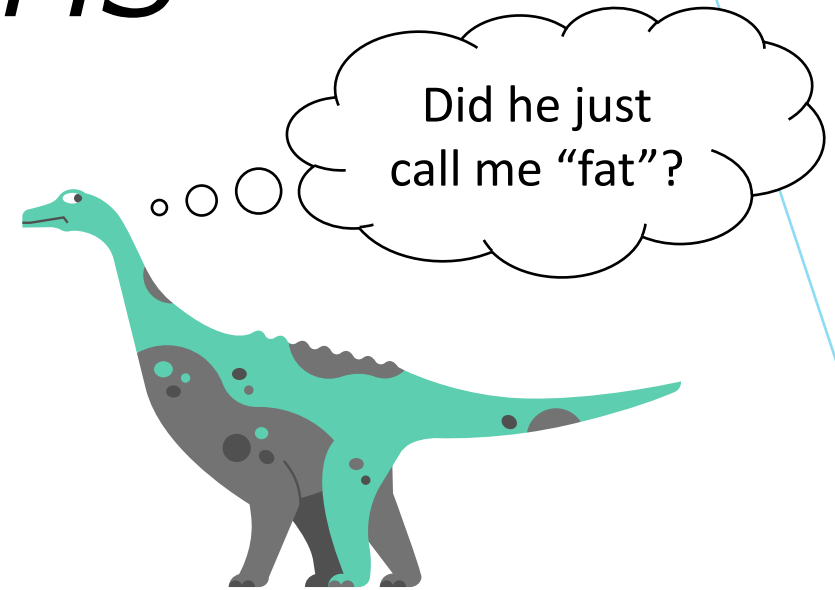
All this, today in…

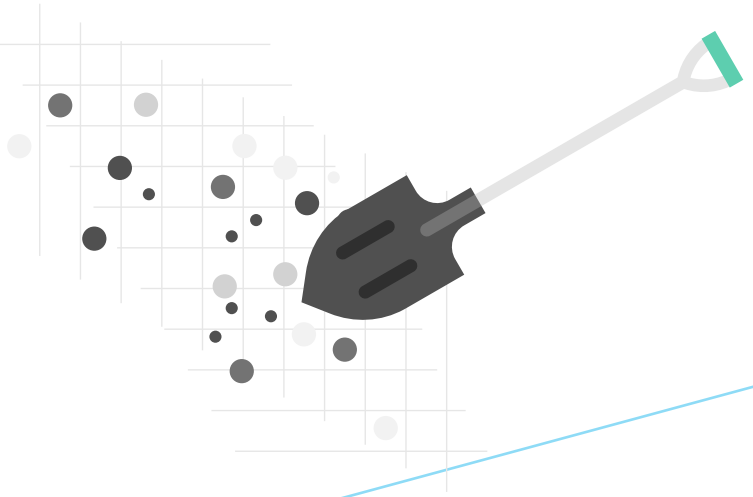
TOPICS IN VECTOR DATABASES!

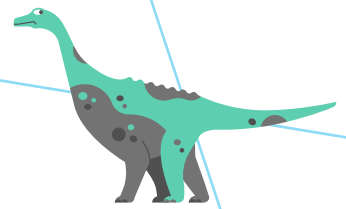
TWO COMMON PROBLEMS

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. Updatable index





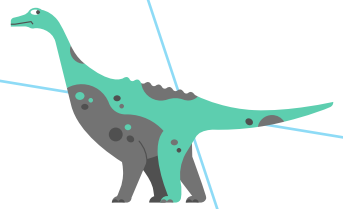


1. SHARDING

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

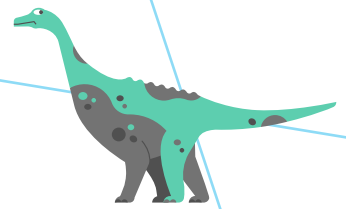
- Benefits:
  - *N/k* fits in machine RAM.
  - Search (and insert) in parallel.
- Downsides:
  - Need *k* machines, *k* can be large.
  - How to deal with edges?
  - Still lots of RAM.
  - Only delays the issue.

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



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]

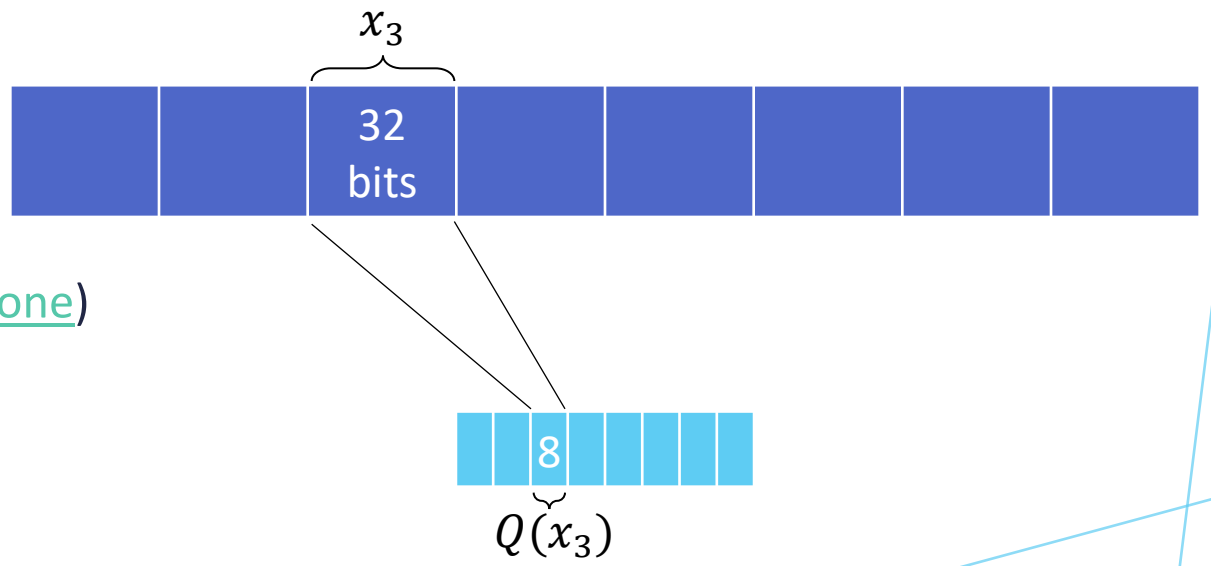


SQn: SCALAR QUANTIZATION

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

$$\text{binsize}_i = \frac{\max(x_i) - \min(x_i)}{2^n} \qquad q(x_i) \cong \frac{x_i - \min(x_i)}{\text{binsize}_i}$$

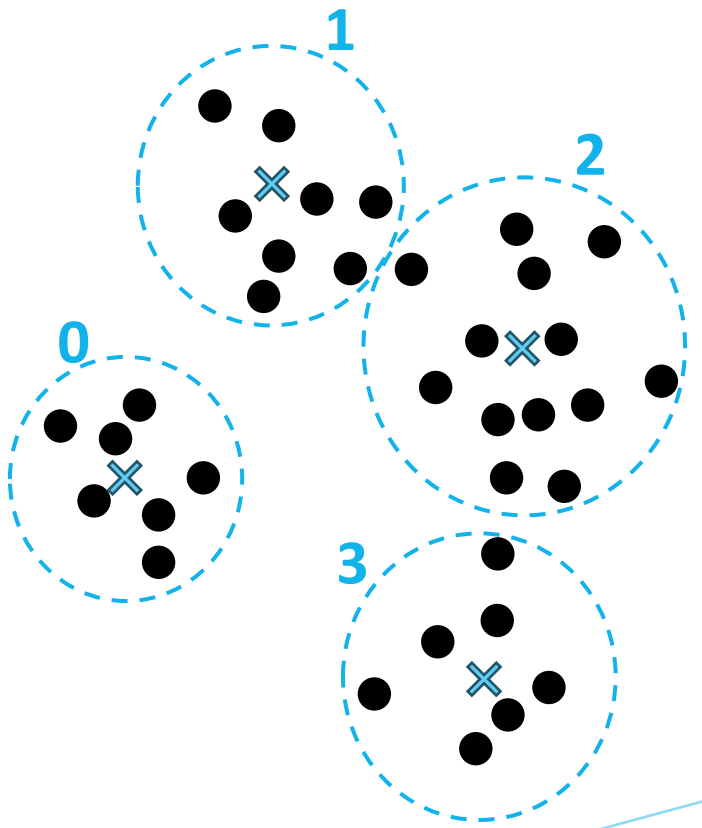
- Example: 32 → 8 (“SQ8”)
  - x4 less memory
  - x2 faster comparison [Qdrant, 2024]
  - ~1% recall loss [Qdrant, 2024]
  - Commonly used with other indices (Pinecone)
- *n* < 8 not common, inaccurate
  - SBQ @ Timescale uses 2 bits

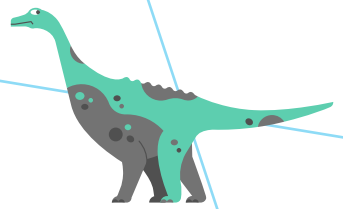


VQ: VECTOR QUANTIZATION

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

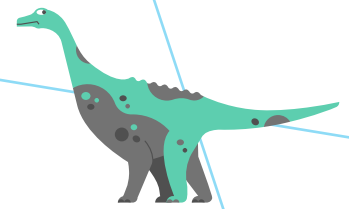
1. Cluster vectors.
   - Codebook = set of centroids.
2. Assign code word to each cluster.
3. $q(x)$ maps x to nearest cluster:
   - Encode x as cluster code word
   - Use centroid in distance computation

- $O(DNk)$ 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]

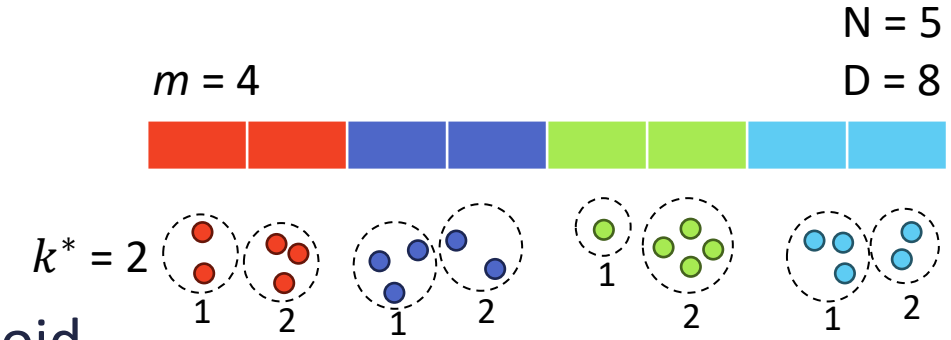
→ **Too slow!**

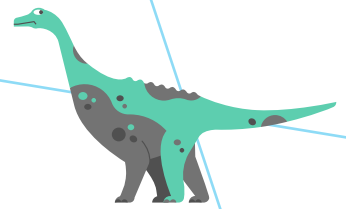
→ **Too big!**



PQ: PRODUCT QUANTIZATION

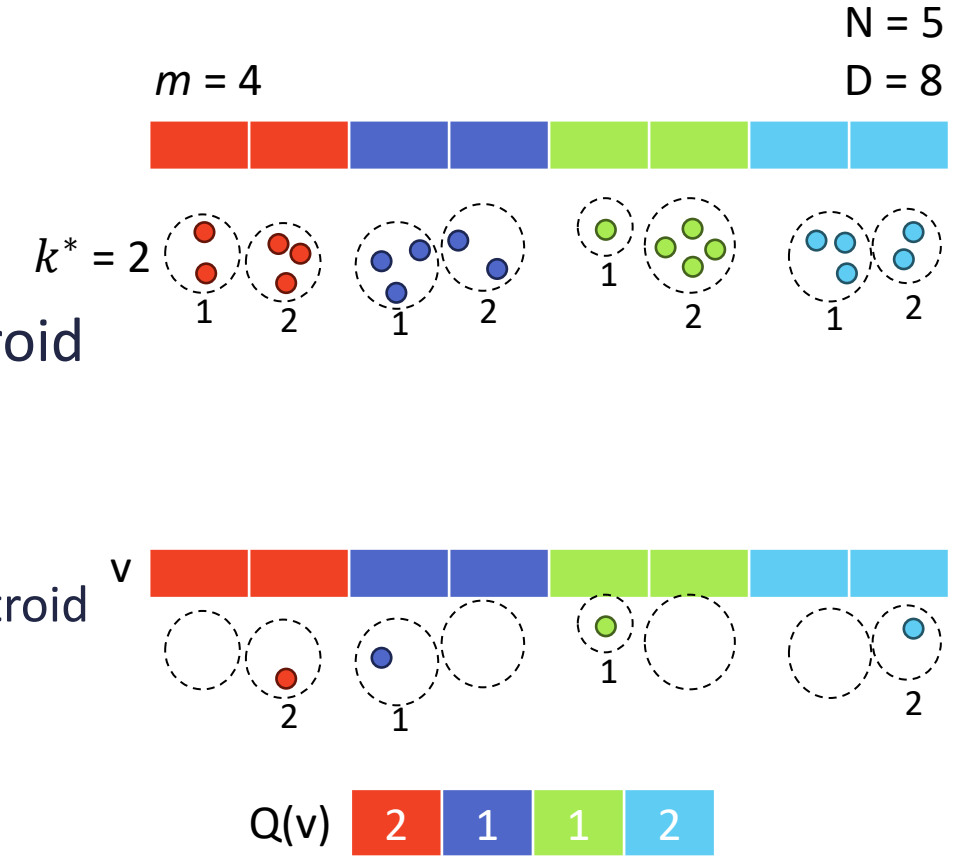
- Split space to *m* chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id $1 \ldots k^*$ to each subspace centroid





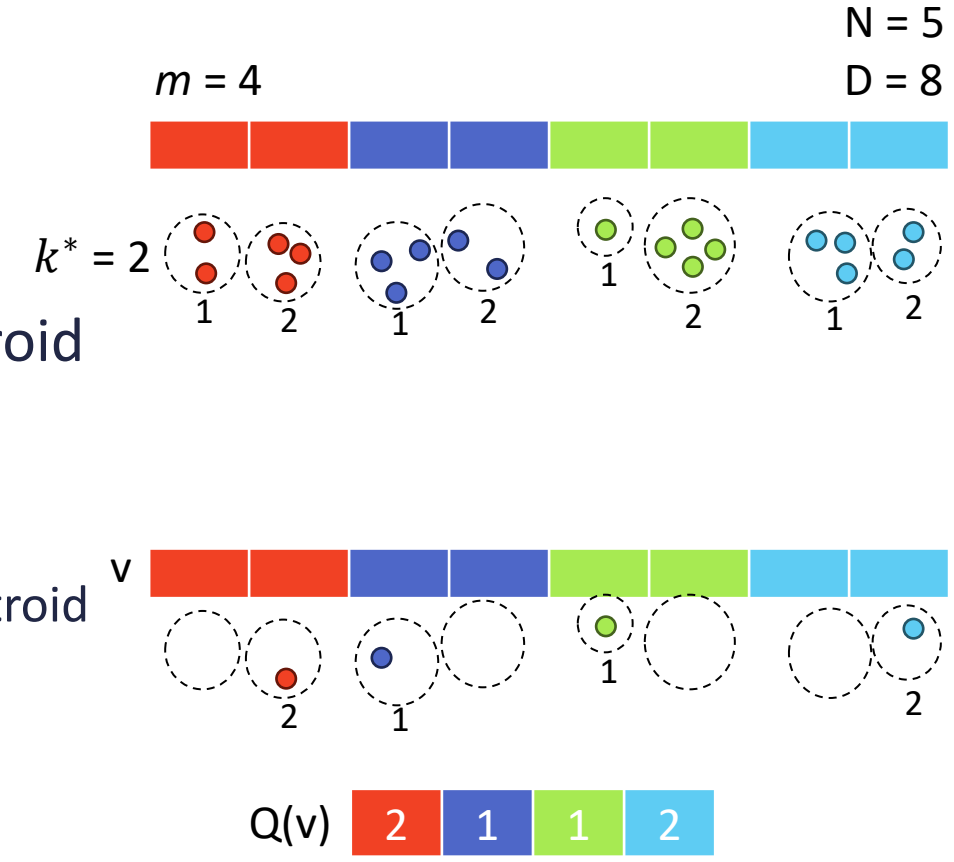
PQ: PRODUCT QUANTIZATION

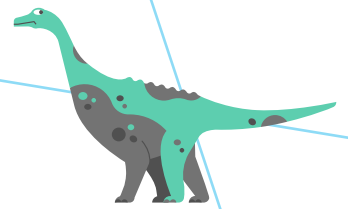
- Split space to *m* chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id $1 \ldots k^*$ to each subspace centroid
- To quantize vector v:
  - Split
  - Replace each chunk with id of nearest centroid
  - Concatenate



PQ: PRODUCT QUANTIZATION

- Split space to *m* chunks (subspaces)
- Cluster each subspace to k^* clusters
- Assign id $1 \ldots k^*$ 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





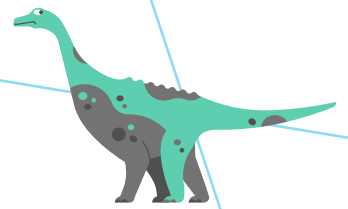
BENEFITS OVER VQ

- 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
    + $32k\frac{\boldsymbol{D}}{\boldsymbol{m}}$ bits for codebook



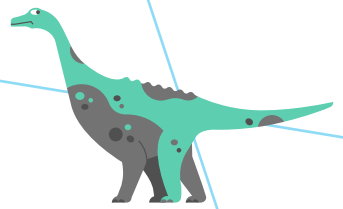
BENEFITS OVER VQ

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



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%**

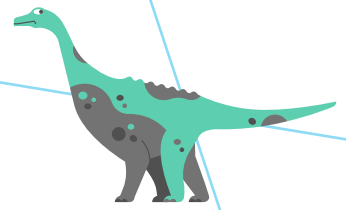
[Briggs, 2024]

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

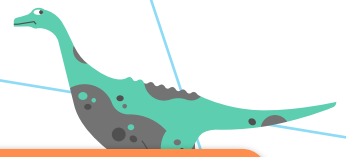


3. COMPOSITE INDEX

- 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 ☺

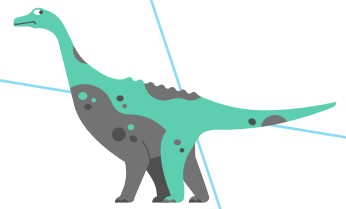
Very fast!

	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

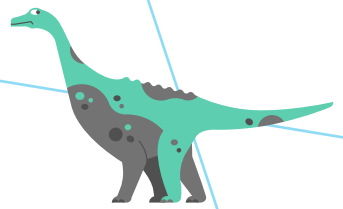
	Flat	IVF256 PQ32x8	IVF4096 HNSW32	IVF4096 HNSW32 PQ32
Memory	512 MB	40MB	523MB	43MB
Query latency	8.26 ms	0.73 ms	0.55 ms	0.55 ms
Recall	100%	74%	90%	69%

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

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

THE QUANTIZATION/COMPOSITE INDEX CINEMATIC UNIVERSE

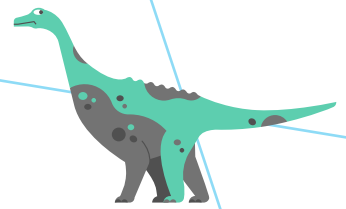
- Lots of research
  - Relevant research – techniques used in practice! **(we shall see a few)**
- OPQ: rotate vectors for optimal PQ [Ge et al., TPAMI 2013]
- IVFADC-R: Three-level quantization + re-ranking [Jégou et al., ICASSP'11]
- IVFOADC+G+P: Near SotA composite index, very fast [Baranchuk, ECCV'18]
- 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]



4. DISK RESIDENT INDEXES

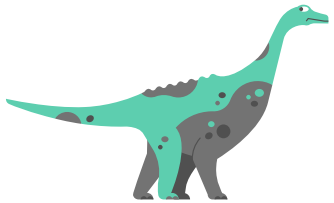
- What if N > 1B ?
  - Indexes are memory-intense
  - Quantization reduces recall
- Offload index to SSD
  - New index structures with careful IO optimizations
  - Updates, rebuilds now more expensive
- For static data:
  - ANNOY – tree-based [Bernhardsson ’20].
  - DiskANN – graph-based [Subramanya,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

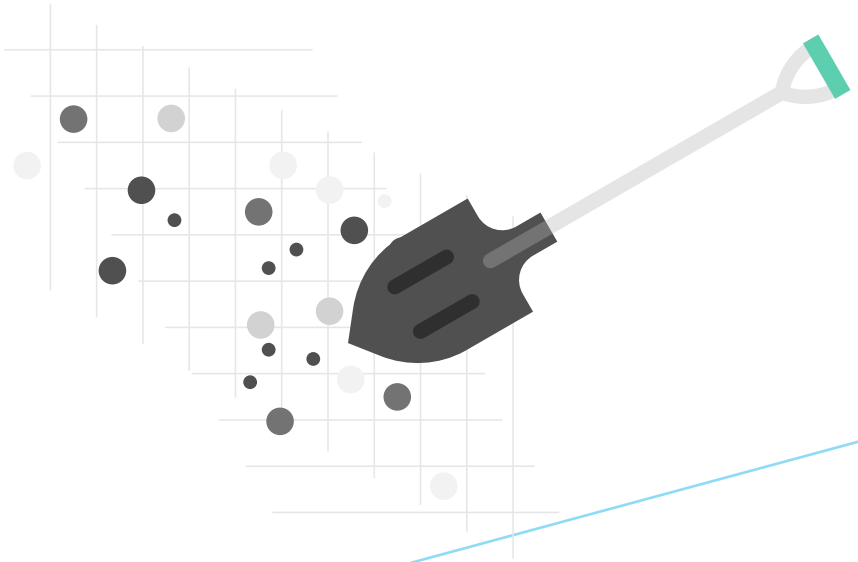
Active research area
We shall see several papers



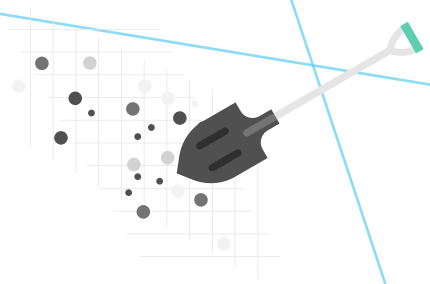
INTERIM SUMMARY

- 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

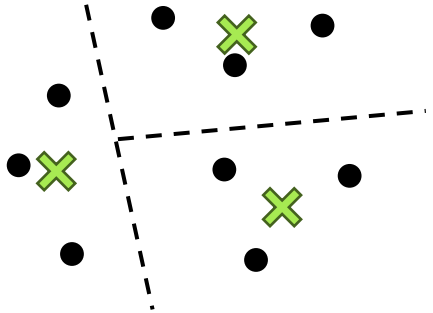




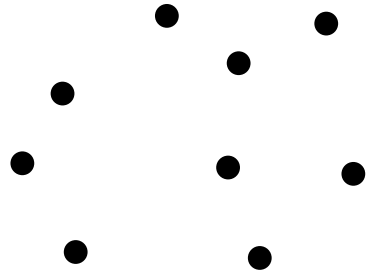
UPDATES DEGRADE INDEX



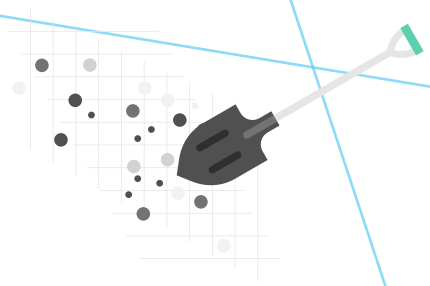
CLUSTER-BASED



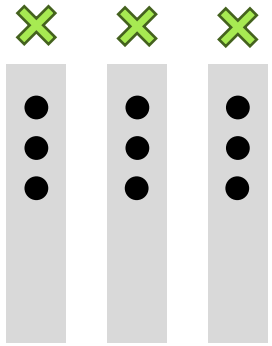
GRAPH-BASED



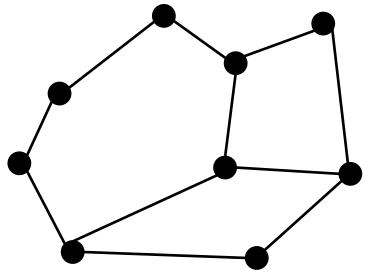
UPDATES DEGRADE INDEX



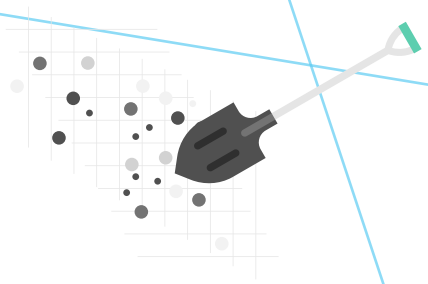
CLUSTER-BASED



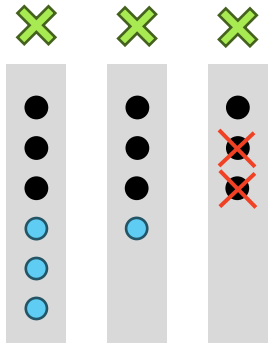
GRAPH-BASED



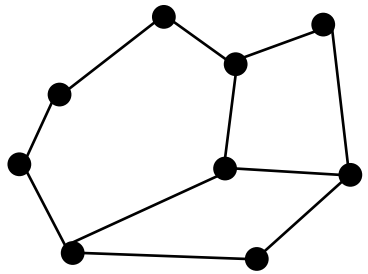
UPDATES DEGRADE INDEX



CLUSTER-BASED

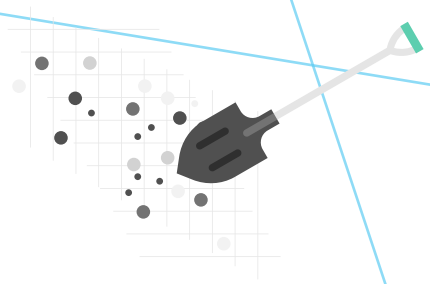


GRAPH-BASED

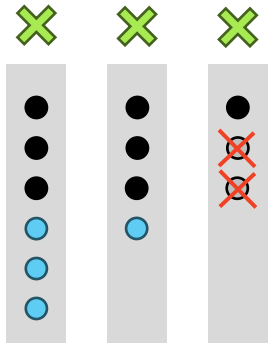


- Updates cause unbalanced partitions
- Large partitions → high latency
- Static centroids → low accuracy

UPDATES DEGRADE INDEX

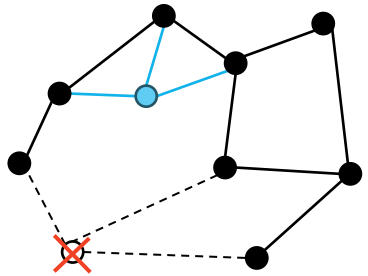


CLUSTER-BASED



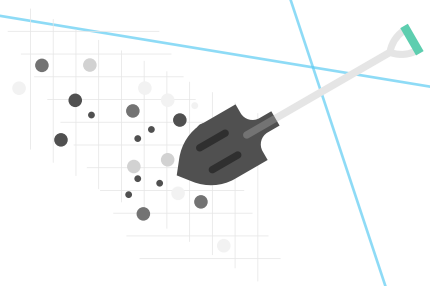
- Updates cause unbalanced partitions
- Large partitions → high latency
- Static centroids → low accuracy

GRAPH-BASED



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

WHAT TO DO?



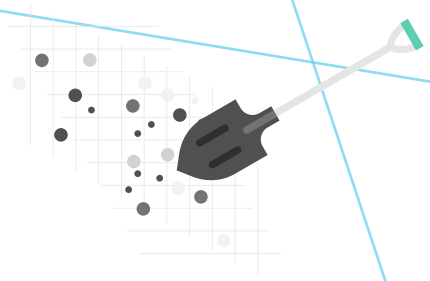
PROBLEMS

- 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**.
- Very 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 → **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:

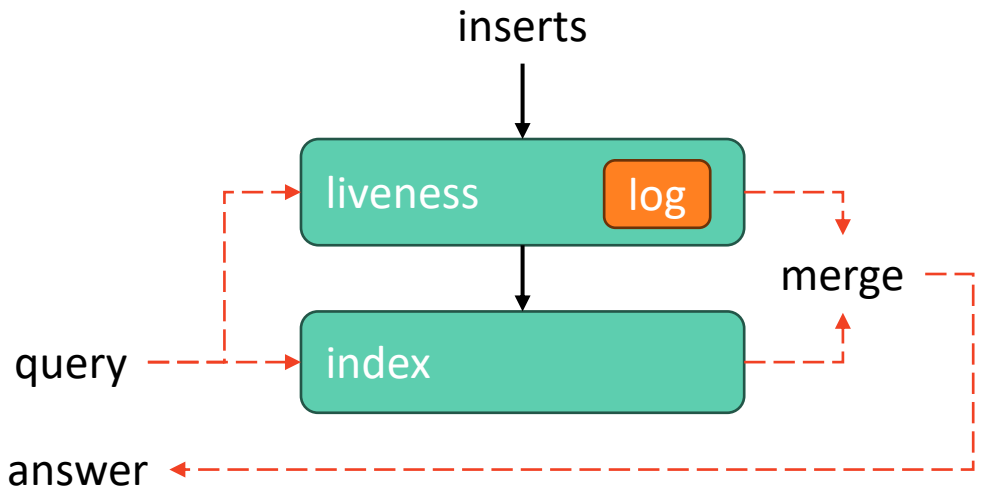
	Memory	CPU	Time
DiskANN	1100 GB 64GB	32 cores 16 cores	2 days 5 days
SPANN	260 GB	45 cores	4 days

[Xu, SOSP'23]

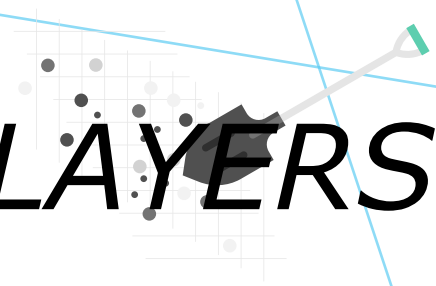
5. FRESHNESS LAYER

(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

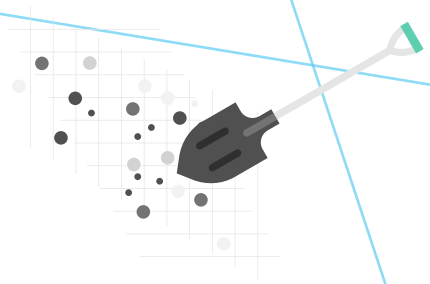


IMPLEMENTING FRESHNESS LAYERS



- 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

6. 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)

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

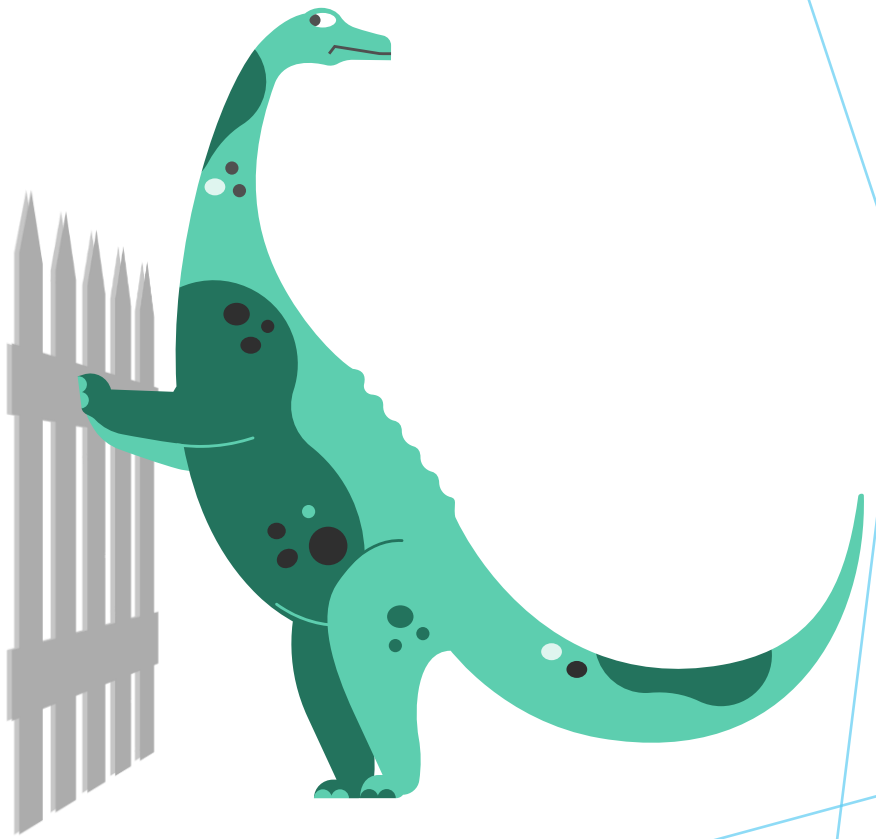
- Techniques
  - Quantization: SQ, PQ
  - Composite
  - Liveness layer
  - Sharding
- Most modern SotA indexes are **graph-based**
  - HNSW (custom variants, implementations)
  - Composite (graph + IVF + quantization)
  - Sometimes disk-resident

RESEARCHING INDEXES

- High-quality implementations:
  - FAISS –most indexes, composite, GPU
  - ANNOY
  - HNSWlib
  - DiskANN (incl. Fresh-…, Filtered-… variants)
  - …
- 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
- Common datasets:
  - SIFT1M, SIFT1B
  - GIST1M
  - DEEP1B
  - GloVe
  - …

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]