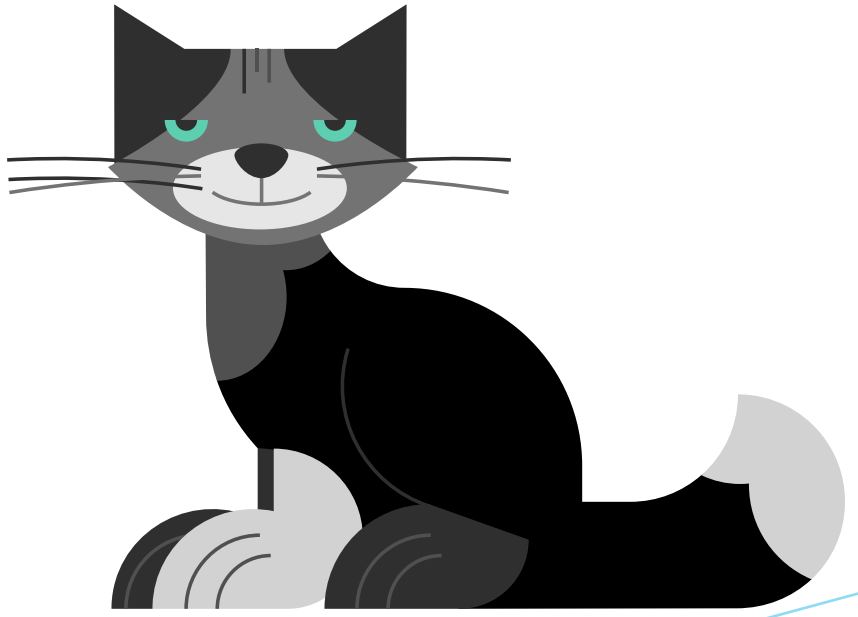
OUR JOURNEY SO FAR...

A review of the **vocabulary**



VECTOR DATABASES

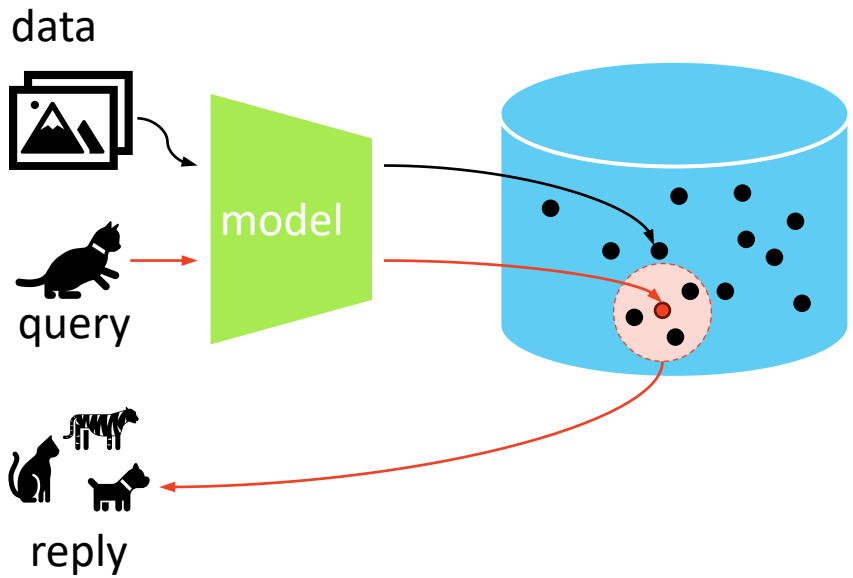
VECTOR STORAGE + SEMANTIC SEARCH

Insert lots of vectors:

1. Create embedding vector *x*
2. Store *x* in specialized DB
   (with associated attributes)

Query data quickly:

3. Embed query as vector *q*
4. Find nearest neighbours to *q*



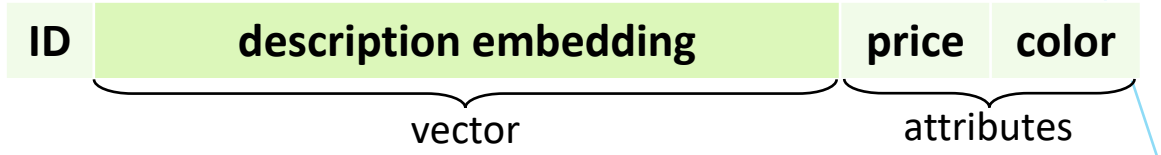
MAIN OPERATIONS

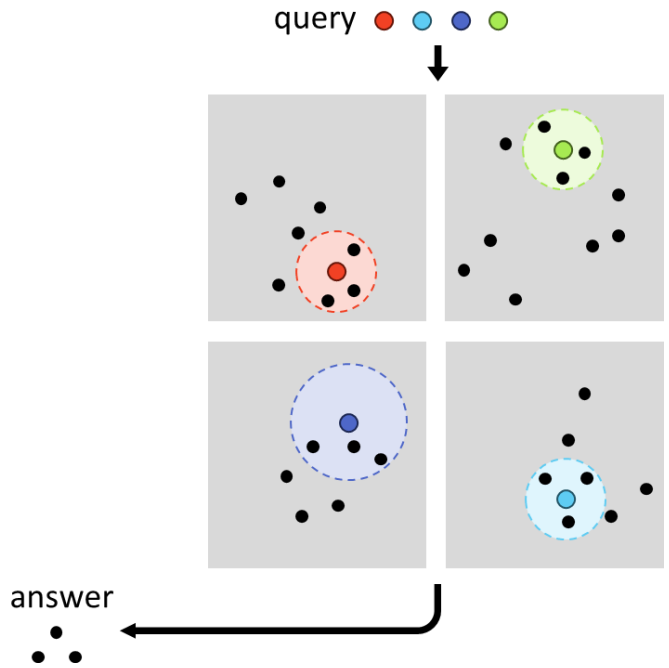
Insert

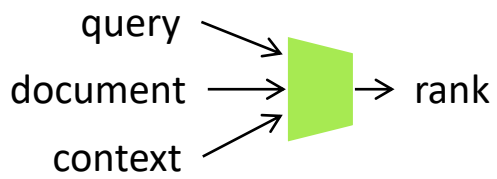
- **Store** id/key + vector + attributes
- Use key to later **delete/update** entire vector
  - Or just get vector by key

Query

- **kNN**: get k-nearest neighbours of *q*
  - Exact or approximate, range
- **Filtered:** kNN + filtering attributes
- **Multi-vector**: search multiple spaces
- Sometimes followed by **re-rank**
- Facilitated by **ANN index**







ANNS INDEX

- Trades performance: **recall** ↔ latency ↔ throughput ↔ memory
  - Generally: slow updates, require rebuilds
- 3 basic structures: **table/cluster**, **tree**, **graph**

flat, LSH, IVF, HNSW

  - Add **quantization**

SQ, PQ

  - Mix and match (**composition**)

IVFPQ, IVFHNSW, …

- Saw **disk-resident** indexes

DiskANN, ANNOY

- To provide fresh answers:
  - Store updates in **freshness layer** (secondary index)
  - **Blue-green** indexing to **rebuild** index

Neos

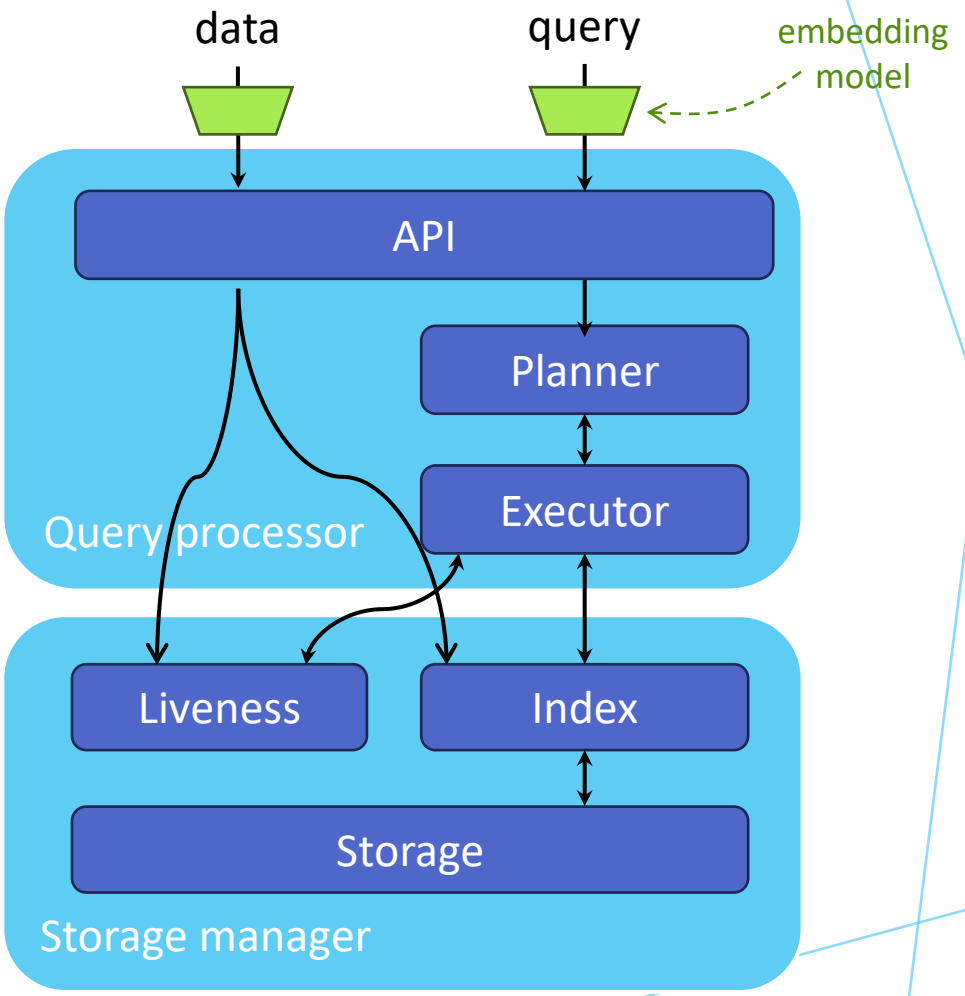
- To avoid rebuilds use:
  - **Updatable** index

SPFresh, FreshdiskANN

IV. ARCHITECTURE

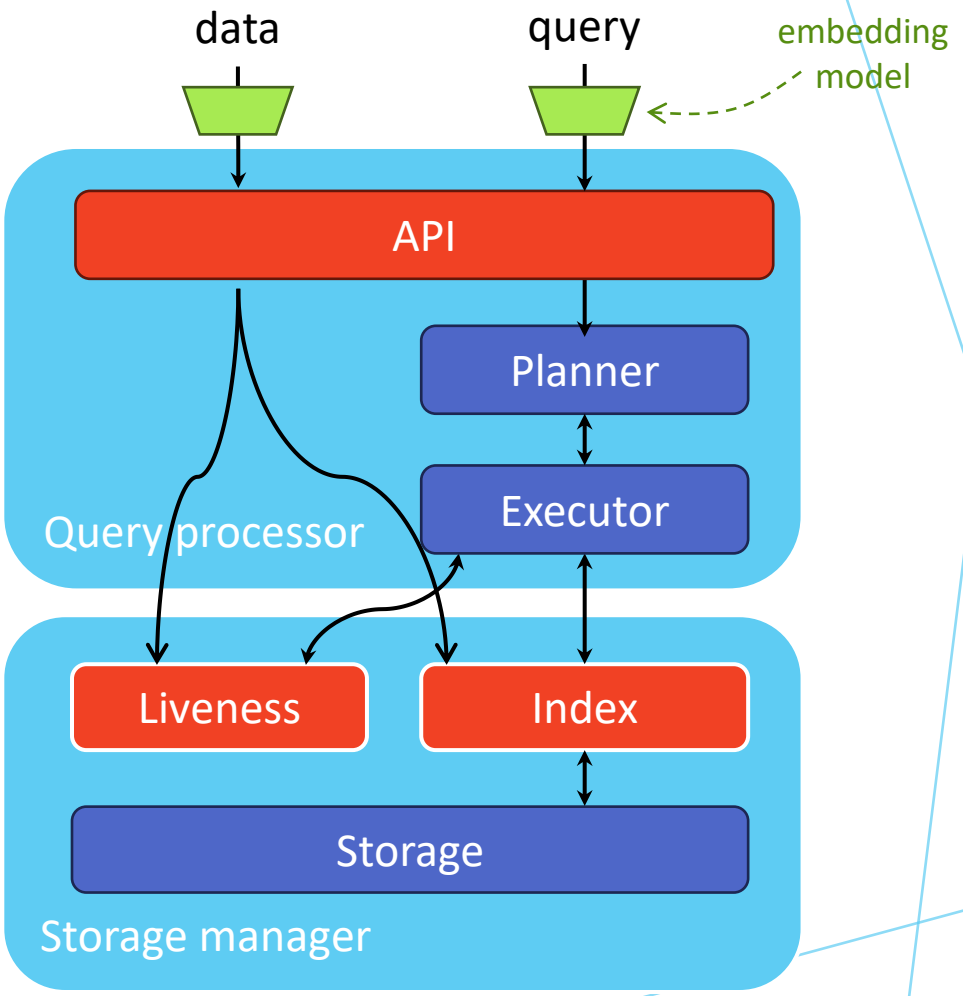
LOGICAL VDBMS COMPONENTS

- Conceptual architecture



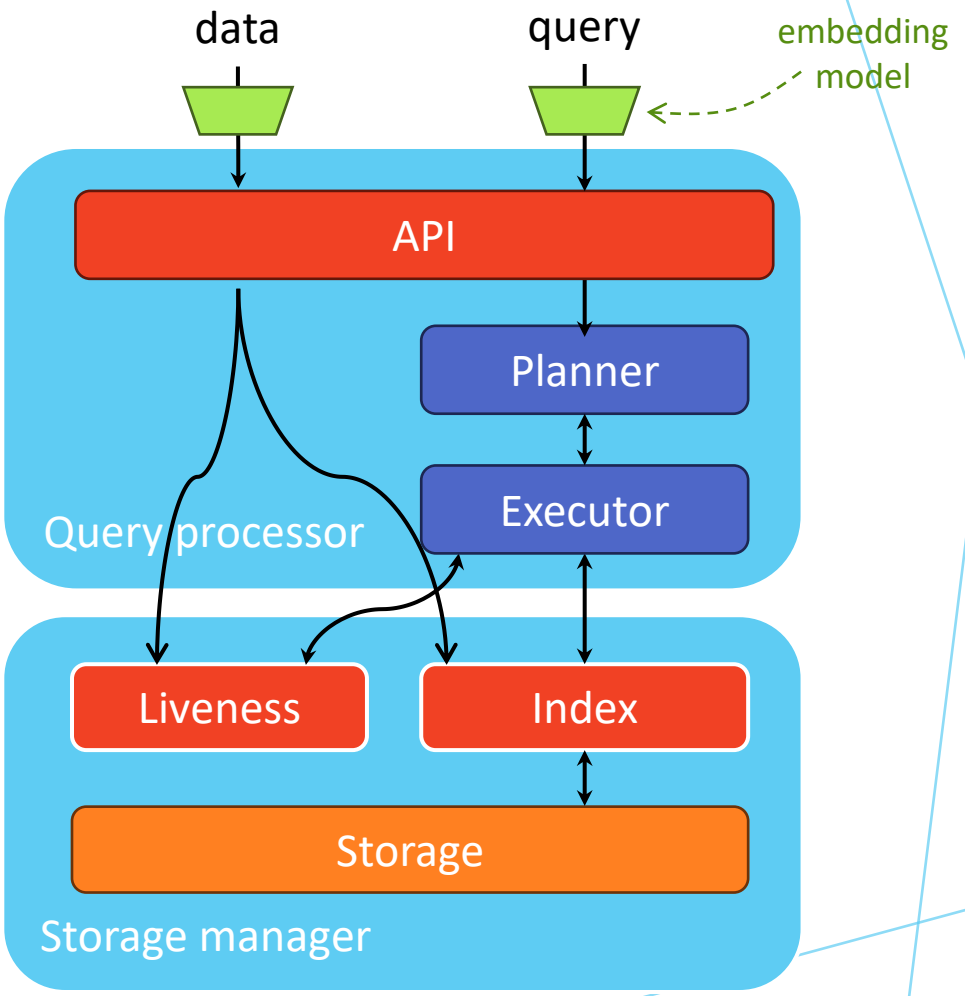
LOGICAL VDBMS COMPONENTS

- Conceptual architecture
  - Explained some parts



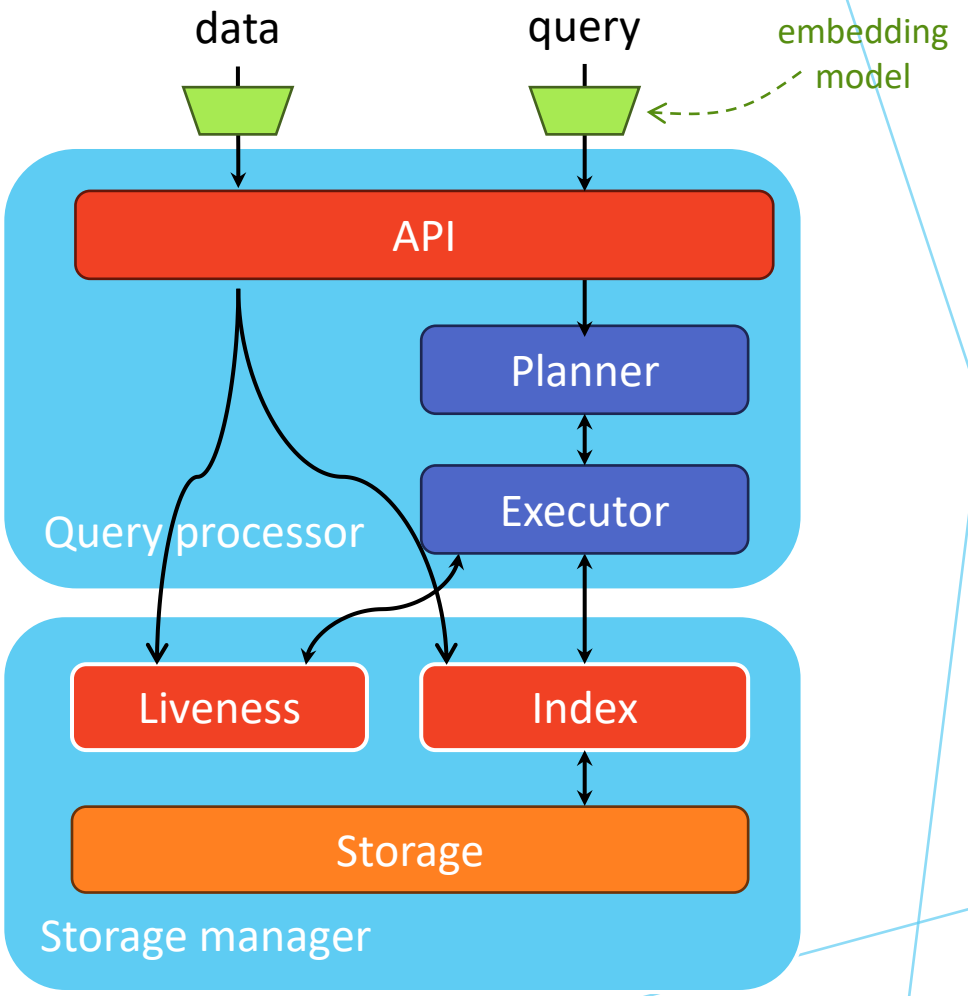
LOGICAL VDBMS COMPONENTS

- Conceptual architecture
  - Explained some parts
  - Briefly touched others



LOGICAL VDBMS COMPONENTS

- Conceptual architecture
  - Explained some parts
  - Briefly touched others
- Different VecDB → different architecture
- Production VecDB more complicated
  - Monitoring
  - Backup
  - Plugin storage/index
  - Security/compliance
  - Orchestration
  - Inference
  - Multi-tenancy
  - Coordination
- Let's look at some.

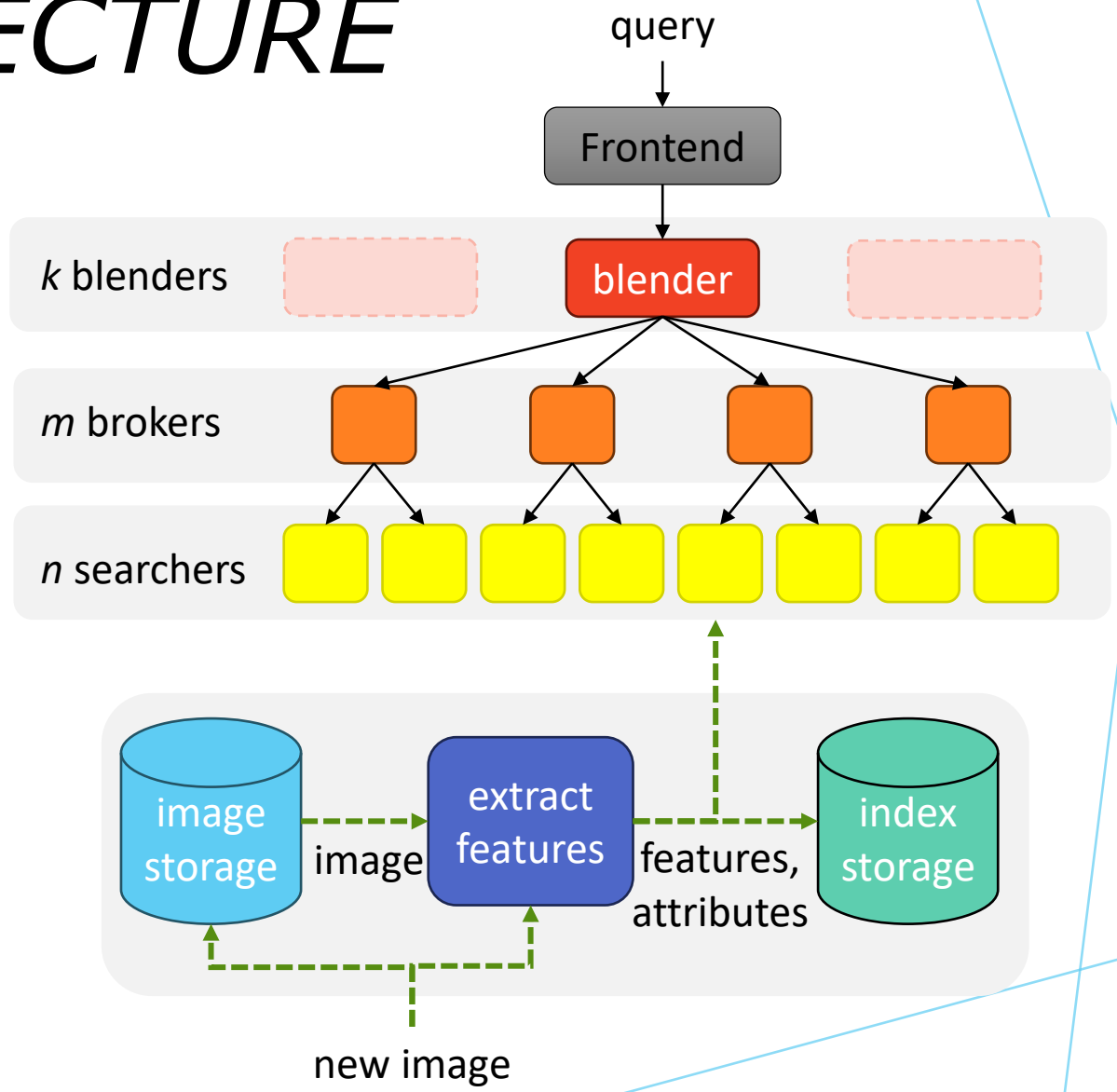


THE PAST: VEARCH [LI, MIDDLEWARE'18]

- Bespoke distributed system
  - Sharded.
  - Ingest queue
  - Data stored in files
- Powers production systems.
- Not perfect.
- Open source and maintained! https://vearch.github.io/home
  - Current design different from [Li, Middleware'18]

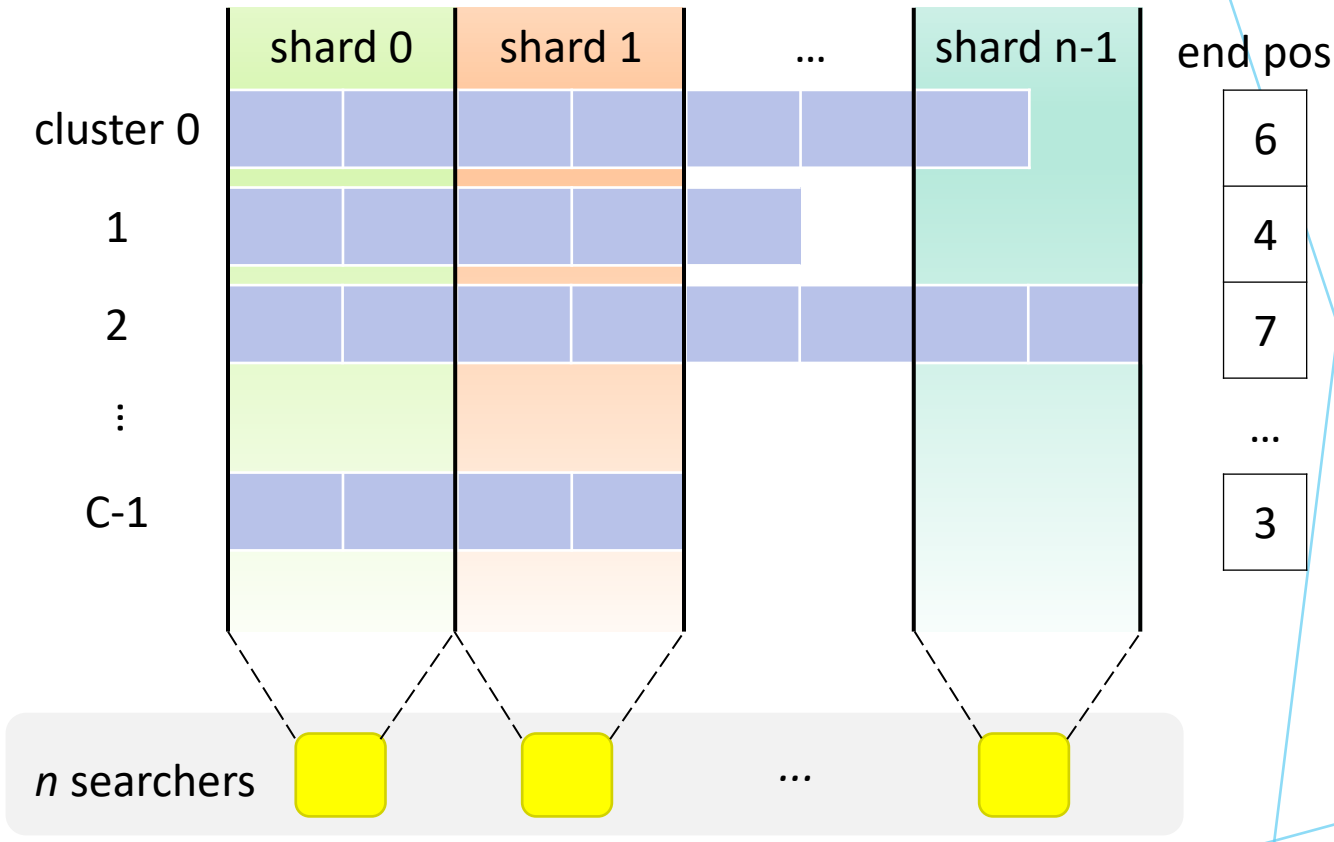
VEARCH ARCHITECTURE

- Hierarchical search subsystem
  - Blenders → brokers → searchers
  - Parallel search
  - Replication for performance/reliability
- Pipelined indexing subsystem
  - Ingest new images.
  - Extract features.
  - “Real-time index” for updates/deletes
  - Weekly rebuilds of full index



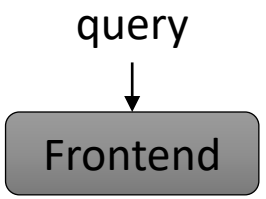
VEARCH INDEX

- IVF (cluster-based)
- Per-cluster list of vectors
- Table of end positions
  - For fast insert
- Shard (partition) index vertically
  - Keyed by image URL
  - Multiple copies per shard
- One shard per searcher
  - Index stored in searcher RAM
- Store product attributes in forward index



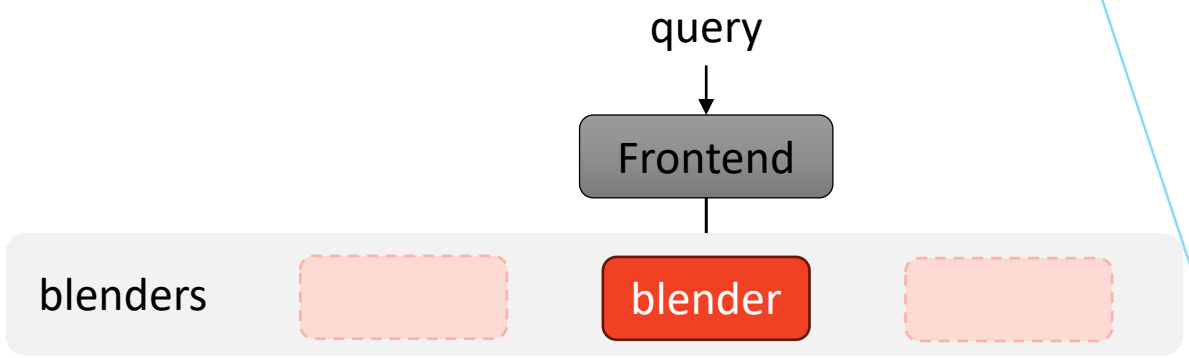
VEARCH SEARCH

- Query arrives at frontend



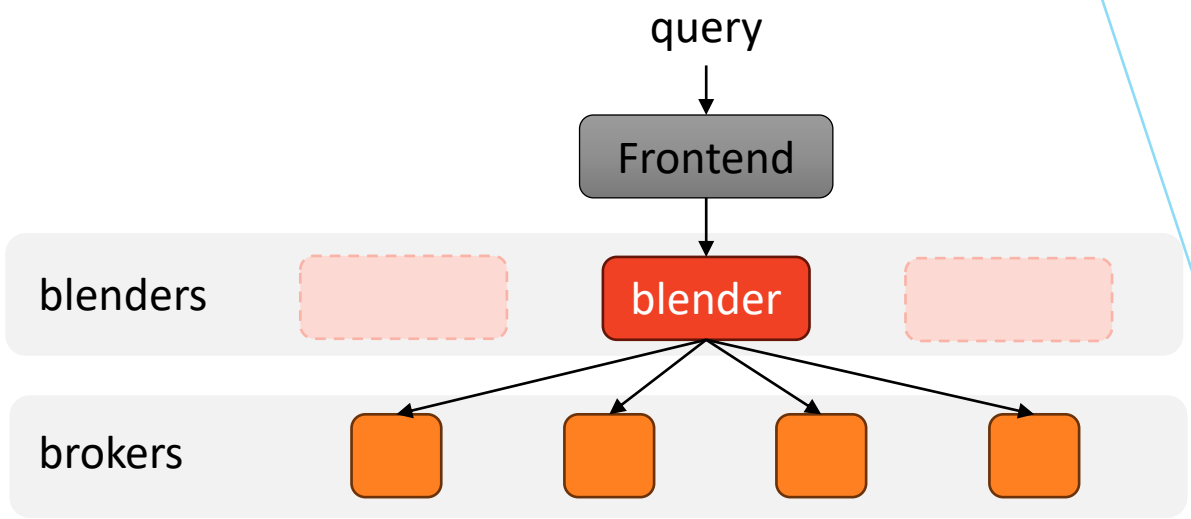
VEARCH SEARCH

- Query arrives at frontend
- Select blender for load balancing



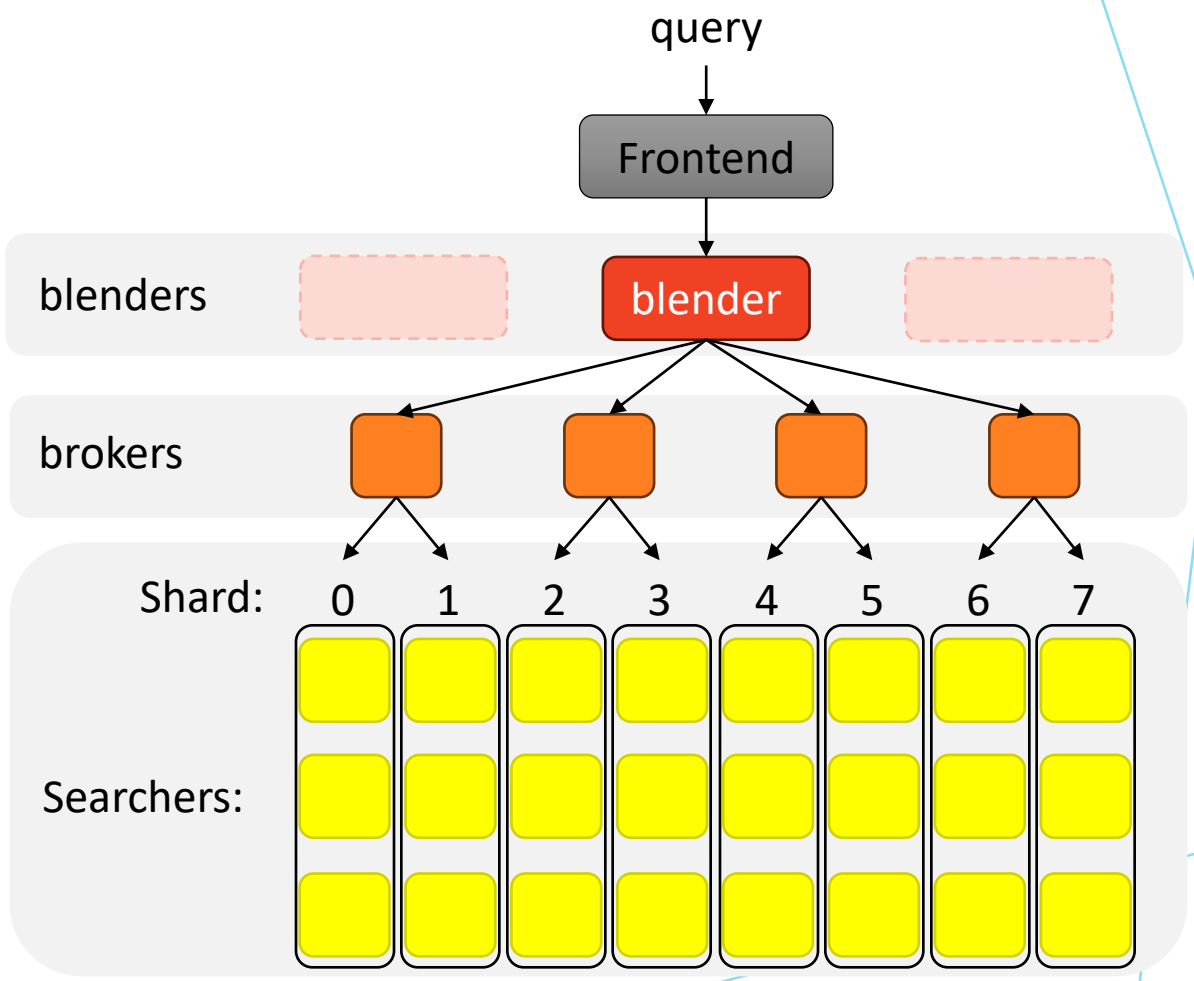
VEARCH SEARCH

- Query arrives at frontend
- Select blender for load balancing
- Blender:
  - Extract features
  - Contact all *m* brokers
  - Combine results



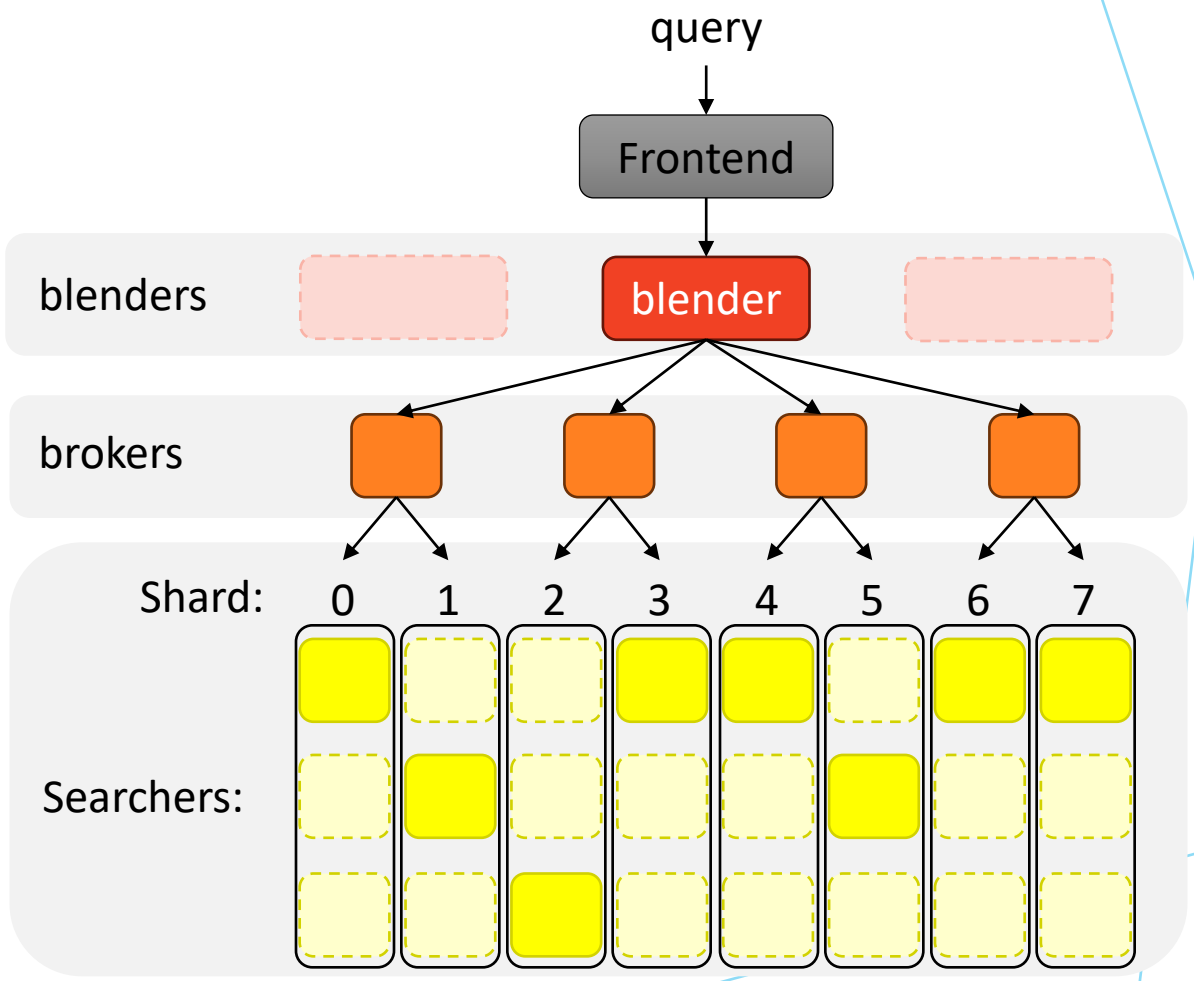
VEARCH SEARCH

- Query arrives at frontend
- Select blender for load balancing
- Blender:
  - Extract features
  - Contact all *m* brokers
  - Combine results
- Broker:



VEARCH SEARCH

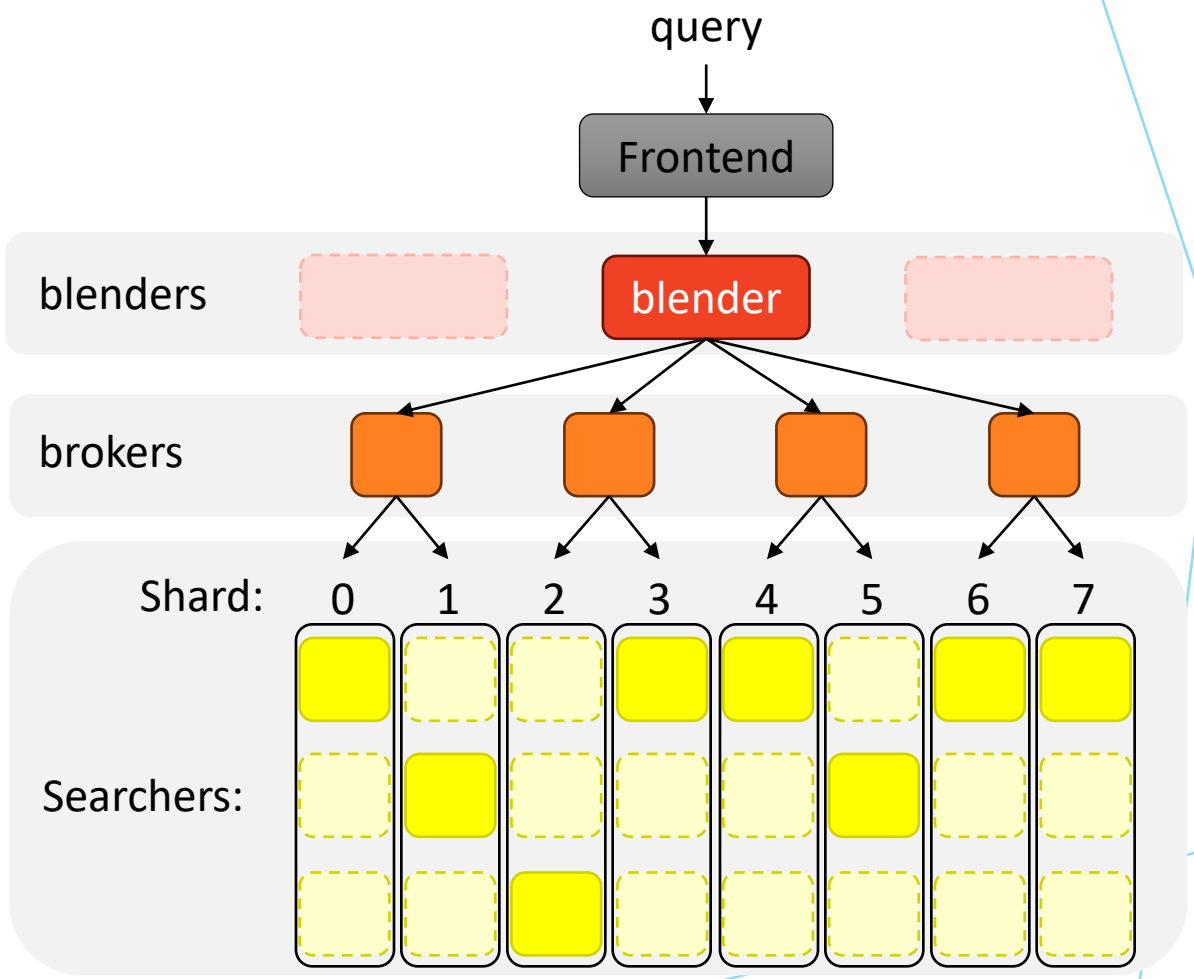
- Query arrives at frontend
- Select blender for load balancing
- Blender:
  - Extract features
  - Contact all *m* brokers
  - Combine results
- Broker:
  - Contact one searcher per shard
  - Combine results



VEARCH SEARCH

- Query arrives at frontend
- Select blender for load balancing
- Blender:
  - Extract features
  - Contact all *m* brokers
  - Combine results
- Broker:
  - Contact one searcher per shard
  - Combine results
- Searcher:
  - Identify cluster
  - Flat search cluster list

} Standard IVF



VEARCH INDEXING

FULL INDEX

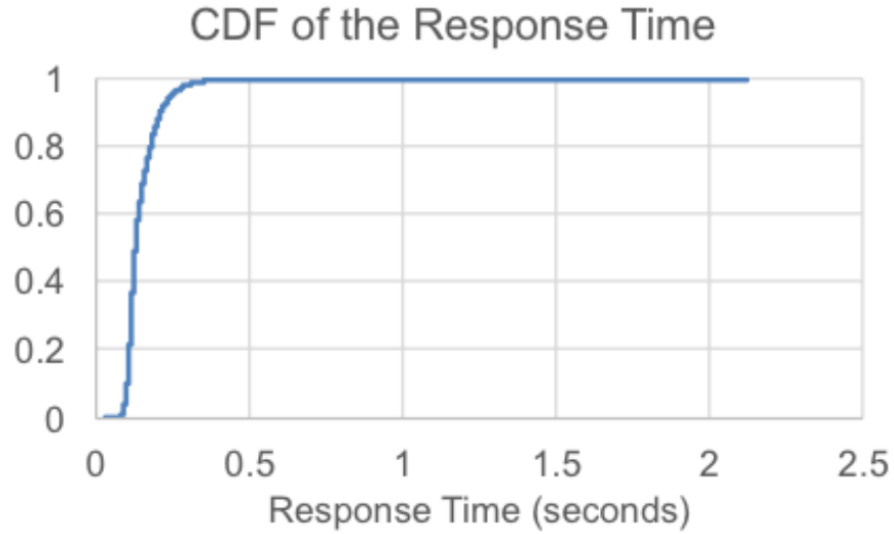
1. Log updates in buffer
2. Ingest buffer nightly:
   - Extract features
   - Store vector + attributes in KV store
   - Mark product as “valid” in bitmap
3. Weekly index rebuild:
   - Rebuild from valid images in store
   - Also build attribute DB (“forward index”)

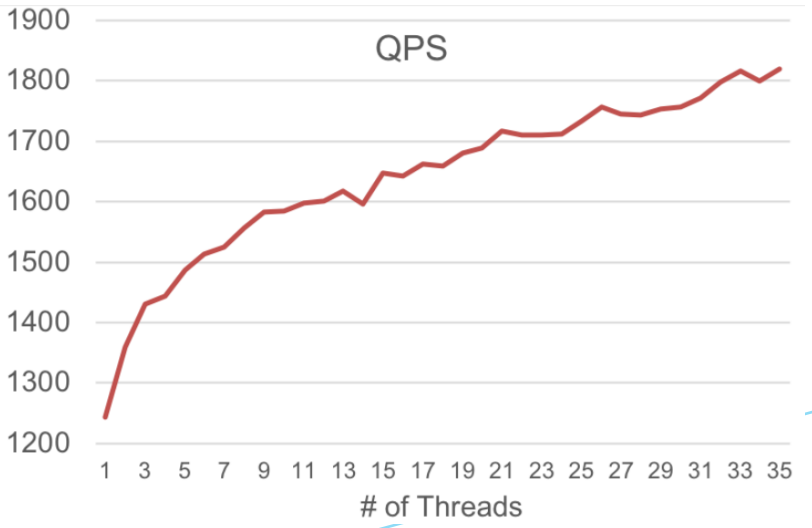
REALTIME INDEX

- Insert:
  - Extracts features
  - Searcher appends vector to nearest cluster
  - Append to attribute DB
- Update:
  - Update product attributes atomically
  - Mark images as valid/invalid
- Delete:
  - Mark invalid in bitmap

VEARCH SEARCH PERF

- N = 1B
- 20 searchers + 6 blenders/brokers
  - Each 24 cores, 256 GB
- Latency:
  - **Avg: ~100 ms**
  - **P99: 300+ ms**
  - **Max: 2.1 seconds**
- Throughput:
  - 1800 queries/sec





VEARCH PROBLEMS

- Slow (in modern terms):
  - High latency: 100s to 1000s of milliseconds
  - Low throughput: 1.8K QPS
- Inefficient:
  - 624 cores, 6.6TB RAM
- RT index interferes with throughput
- Weekly rebuilds
- Inflexible architecture
  - Nodes assigned role, data → hard to repurpose, scale

THE PRESENT: MANU [GUO, VLDB'22]

- Commercial product
  - Used in production.
  - Open source.
- Evolved from Milvus [Wang, SIGMOD'21]
  - Known as Milvus 2.0
- Milvus followed RDBMS design:
  - Lots of features.
  - Fast execution.
  - Monolithic design.
  - Caused trouble.

Manu:

- Service-oriented / disaggregated design:
  - Separate workers for each function.
  - Connect with distributed log.
  - Coordinators orchestrate work.
- Scale functionality independently.
- Support variety of indexes
  - FAISS, HSNWlib, ANNOY,…

REQUIREMENTS → FEATURES

Clients say:

- Row-level ACID is enough.
  - No need for multi-row/table transactions.
- Need fine-grained consistency
  - Strong vs. eventual too coarse.

    0 staleness ⟵ ? ⟶ ∞ staleness
  - App requirements differ.
- Need fine-grained elasticity:
  - GPUs are expensive.

Manu provides:

→ Insert/update/delete/query
- (We know this by now).

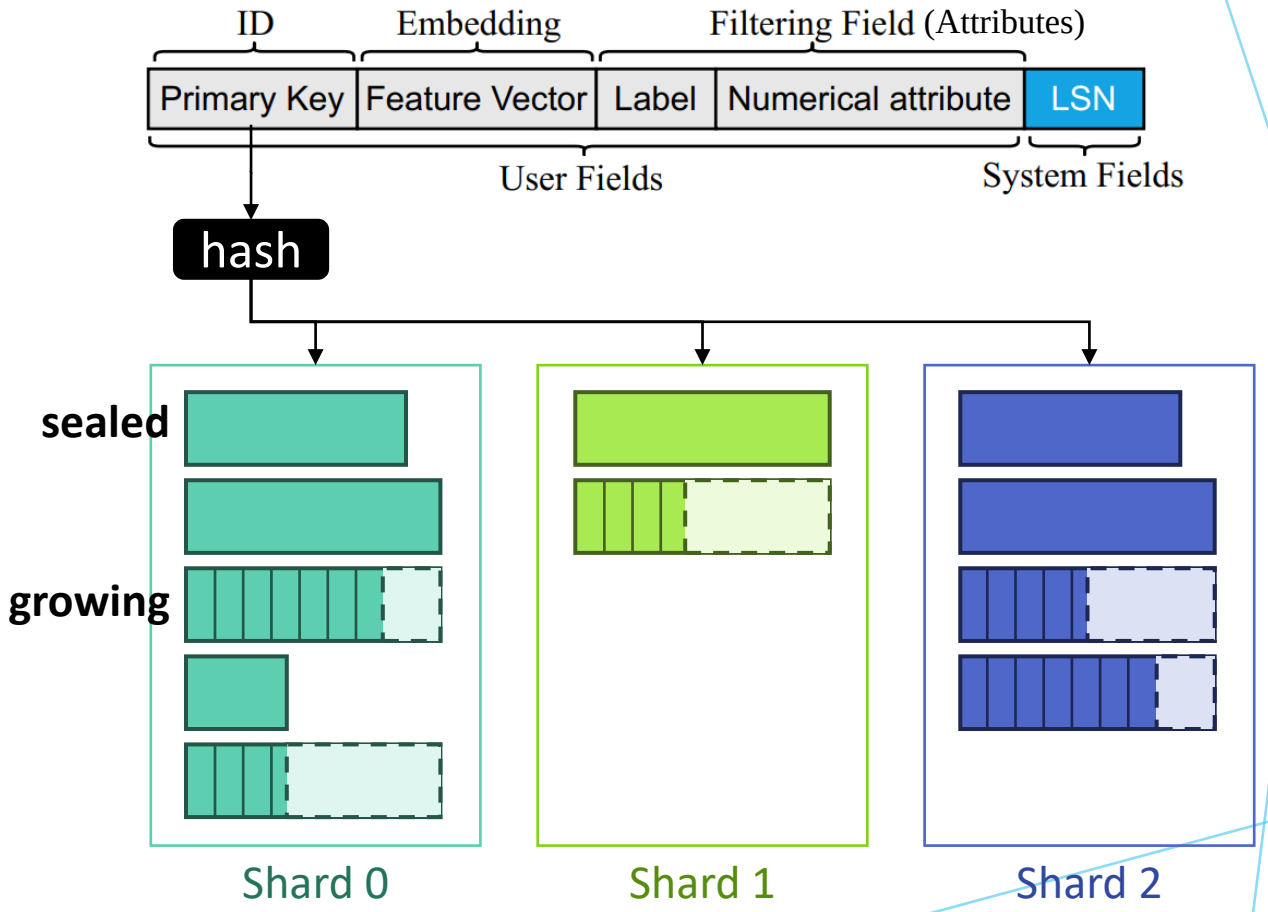
→ Bounded staleness
- "Delta consistency"

  0 staleness  ∞ staleness
- Aka Laxity, Δ-atomicity [Mortazavi, SEC'20]

→ Disaggregate functionalities
- Scale index, search, storage independently.

DATA ORGANIZATION

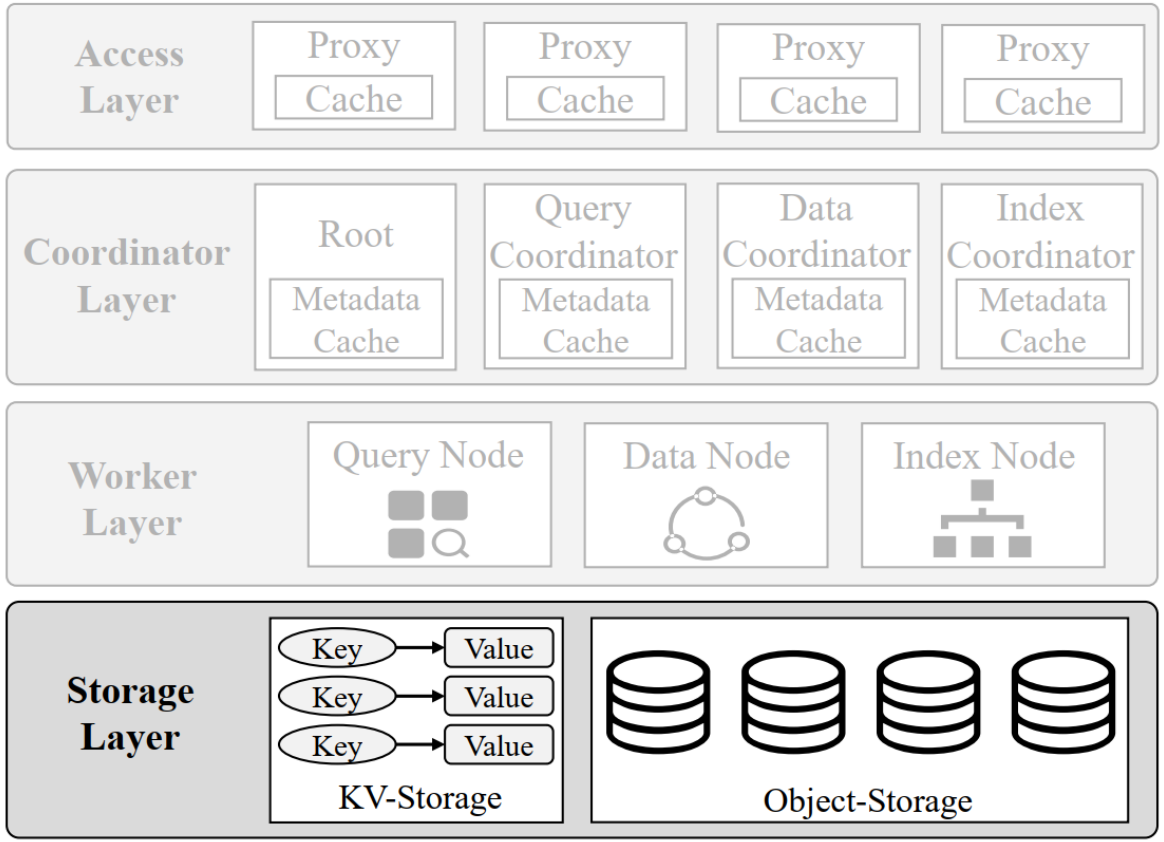
- Store
  - User data: key (ID) + vector + attributes
  - Metadata: sequence number
- Data partitioned to **shards**
  - Statically, by primary key.
  - Shard = write channel in write-ahead-log
- Shard divided to **segments**
  - Number grows with data
  - **Growing**: append only
  - **Sealed**: read only
  - Seal and index when 512MB / after 10 secs
- Growing segment divided to **slices**
  - Slice = 10K vectors
  - IVF in full slices (for searching)



ARCHITECTURE

Storage layer persists information.

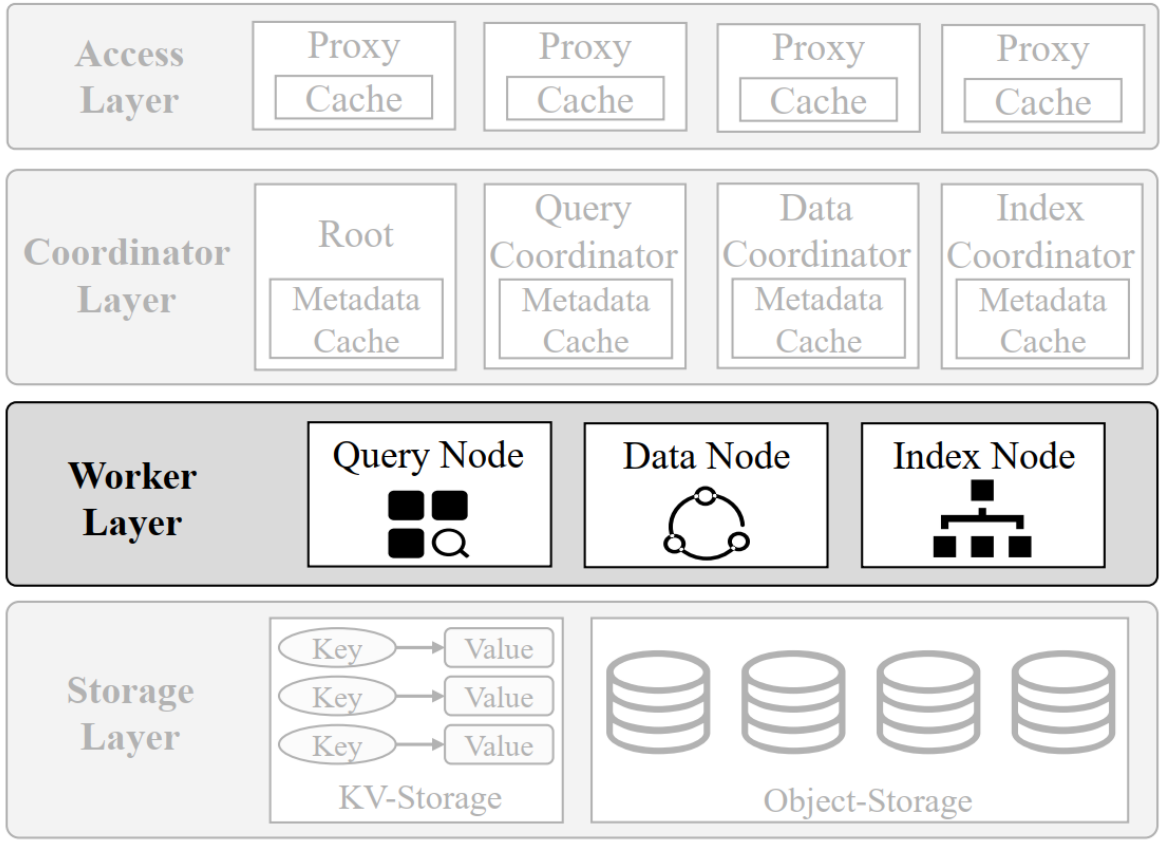
- **Object storage** (S3, MinIO, files)
  - Columnar Binlog (vectors + data)
  - Index
  - Segment mapping SSTables
- **KV store** (etcd)
  - Metadata
  - Status info.
  - Used by coordinators.



ARCHITECTURE

Workers run computational tasks.

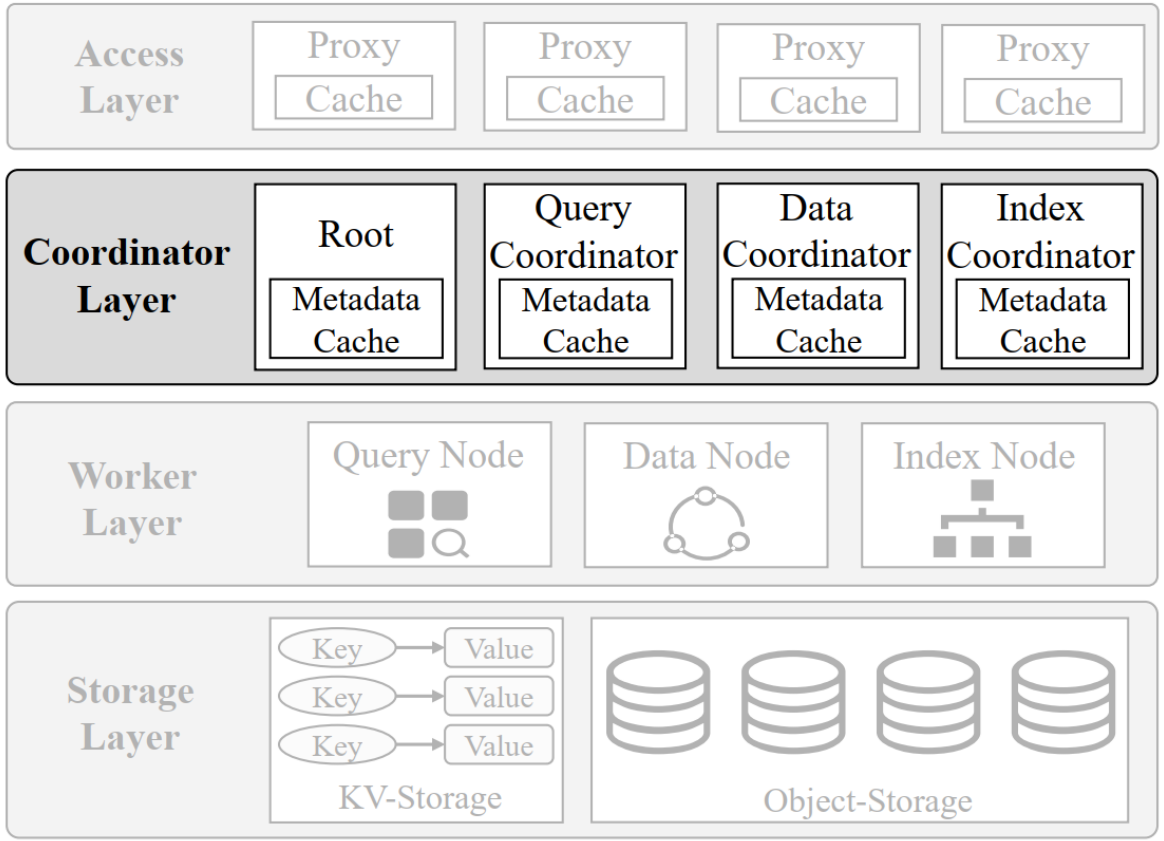
- Stateless.
- No inter-worker coordination.
- Can scale independently.
- **Query:**
  - Execute searches.
- **Data**:
  - Persist queued updates to binlog.
- **Index**:
  - Build index.



ARCHITECTURE

Coordinators manage metadata, nodes.

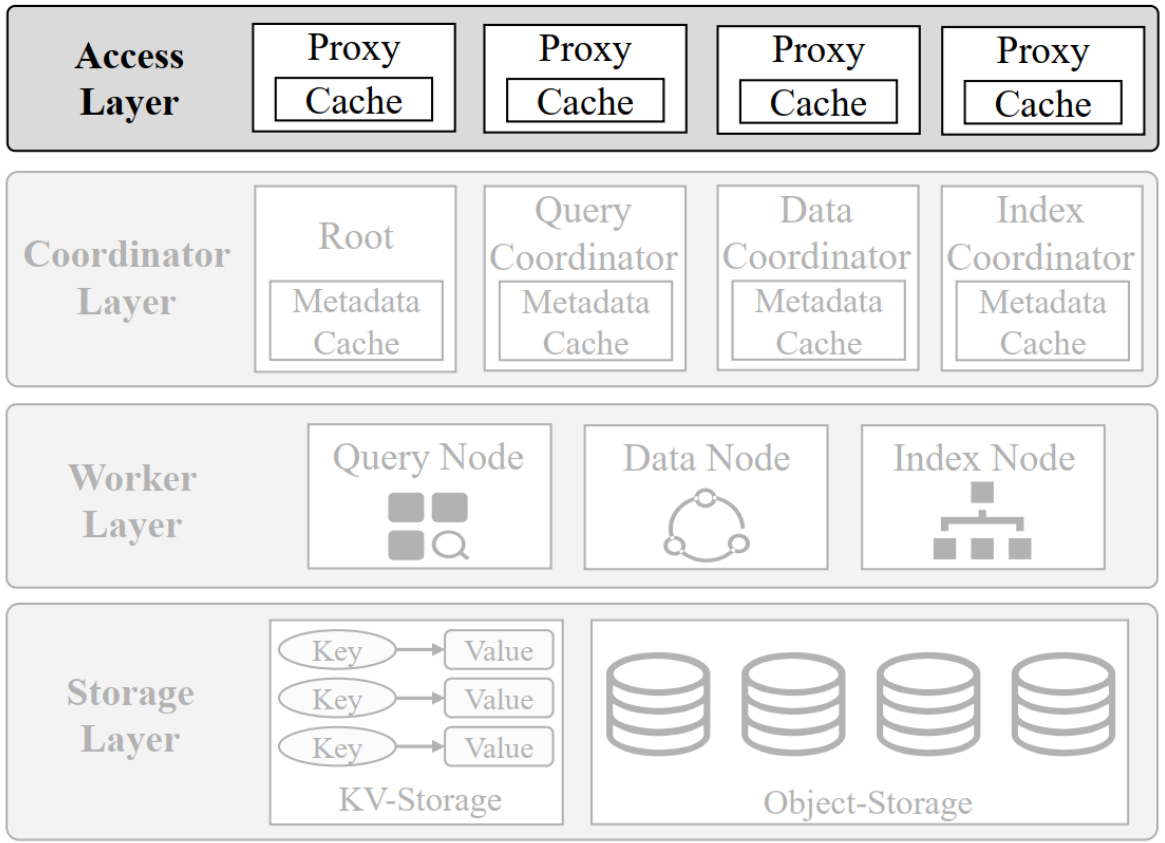
- Multiple hot backups
- Metadata caching for performance
- **Root coord** :
  - System, cluster, collections.
  - Provides timestamp oracle service (TSO)
- **Data coord** manages**:**
  - Data nodes
  - Segment route on storage
- **Query coord** manages**:**
  - Query nodes
  - Assigning segments to query nodes
- **Index coord** manages:
  - Index nodes
  - Index metadata



ARCHITECTURE

Access layer: deal with clients

- Stateless proxy
- Metadata cache
  - Reduces hops, load for validation
- Handle requests:
  1. Get request
  2. Verify search validity
  3. Distribute request to workers
  4. Aggregate results

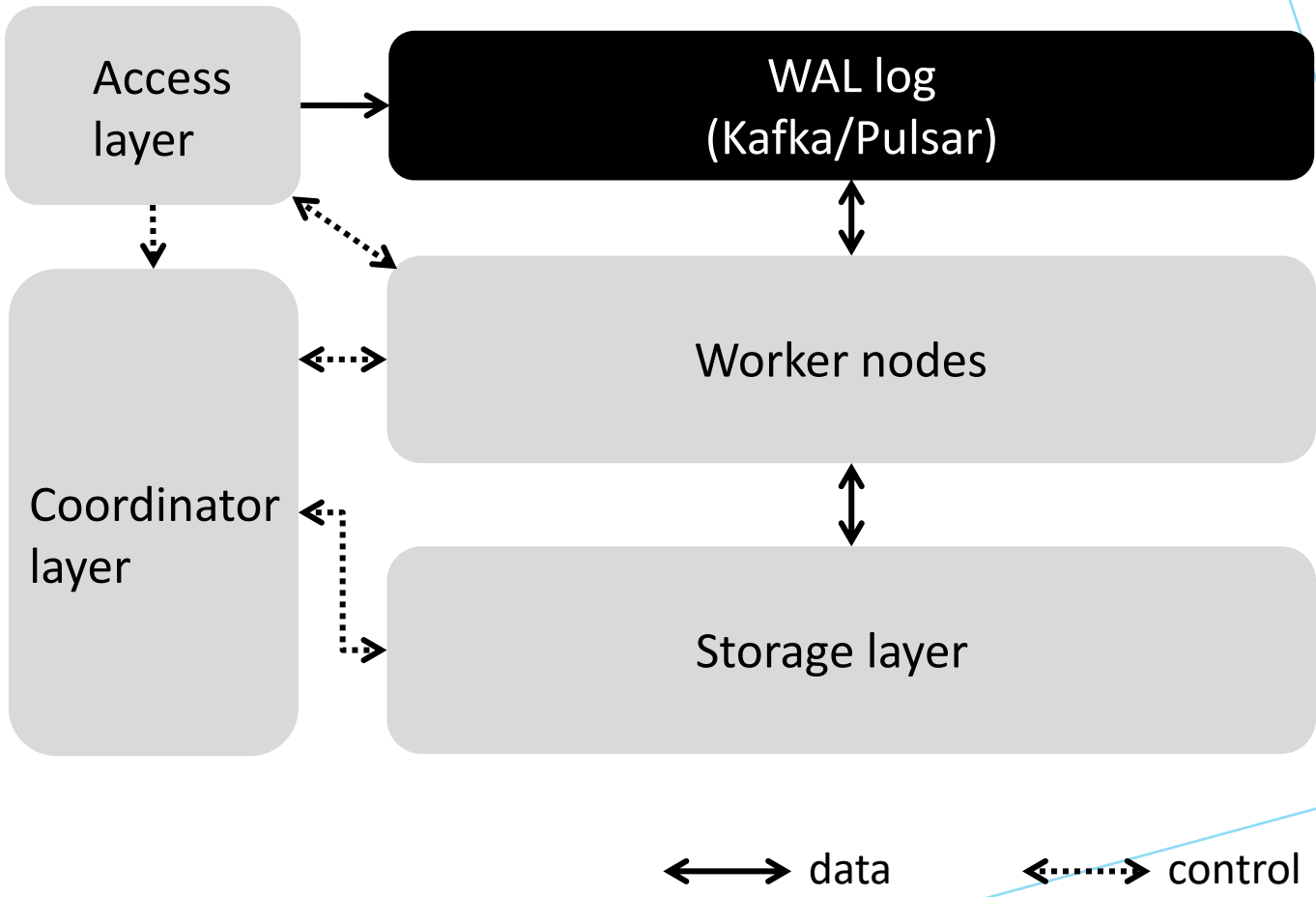


THE LOG BACKBONE

Write-ahead-log (**WAL**) connects components

- Changes published to WAL
  - Insert/delete vector.
  - Create/delete collection.
  - … any change to system state.
- Also used for control plane.
- Nodes subscribe to updates
- Queries bypass WAL

→ decoupled architecture:
scale components independently

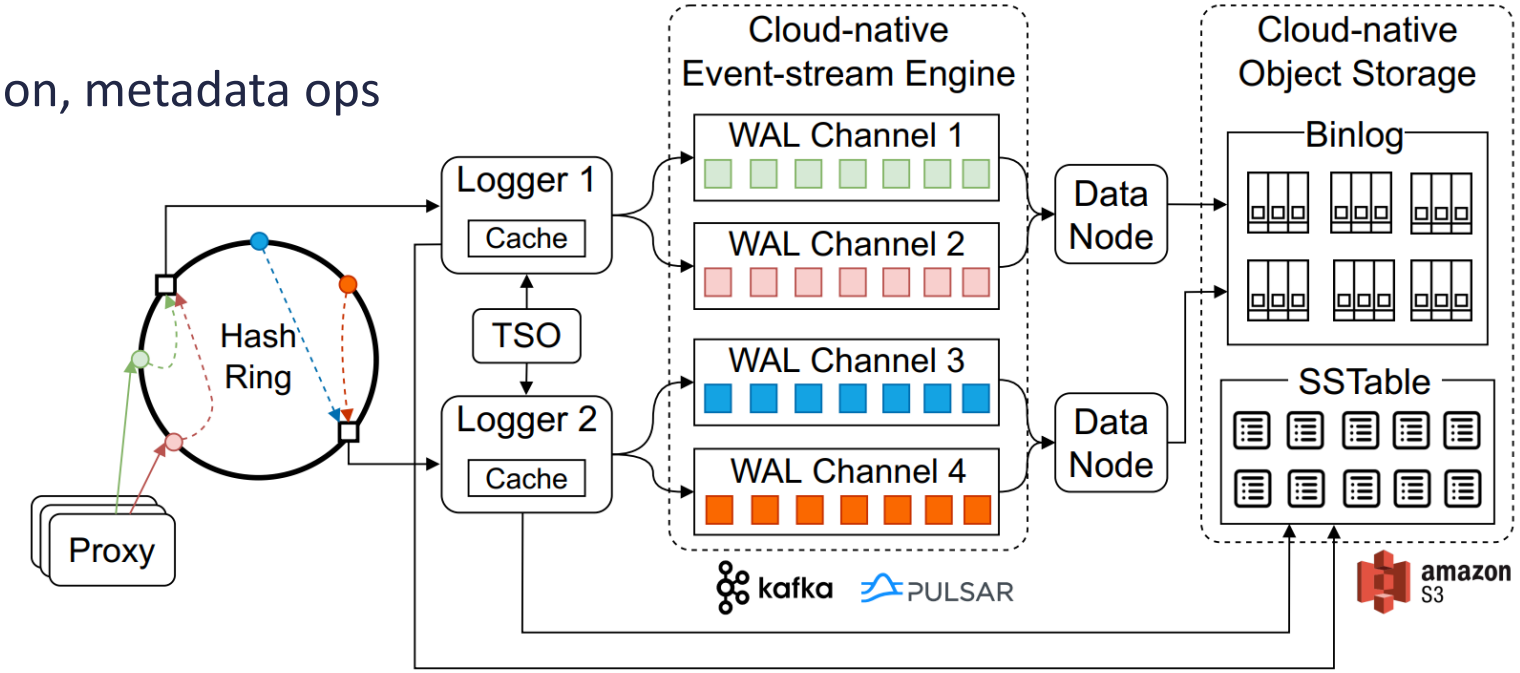


INSERTING A VECTOR

- In a world…
- where vectors are many…
- and queries are fast…
- …one vector makes the journey…
- **…from insert to index.**

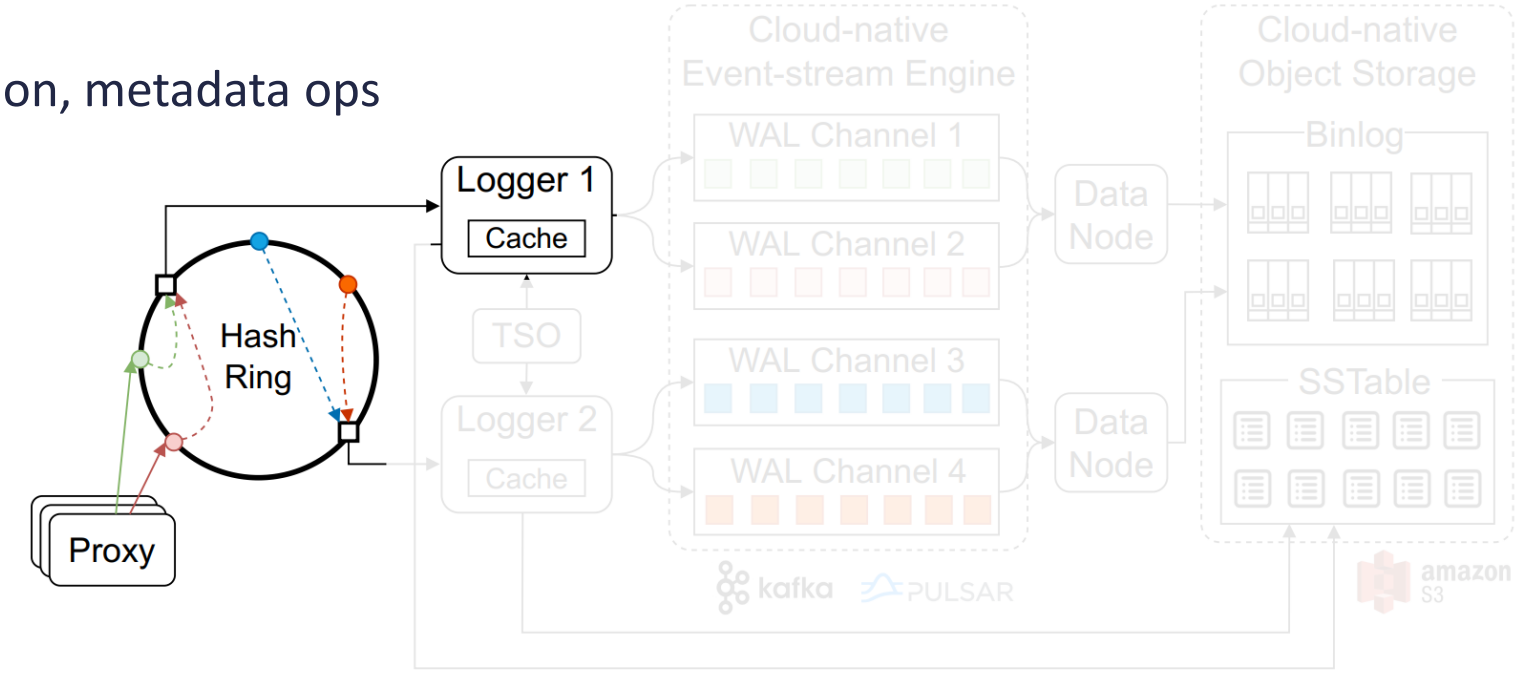
FIRST STOP: WRITE-AHEAD-LOG

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:



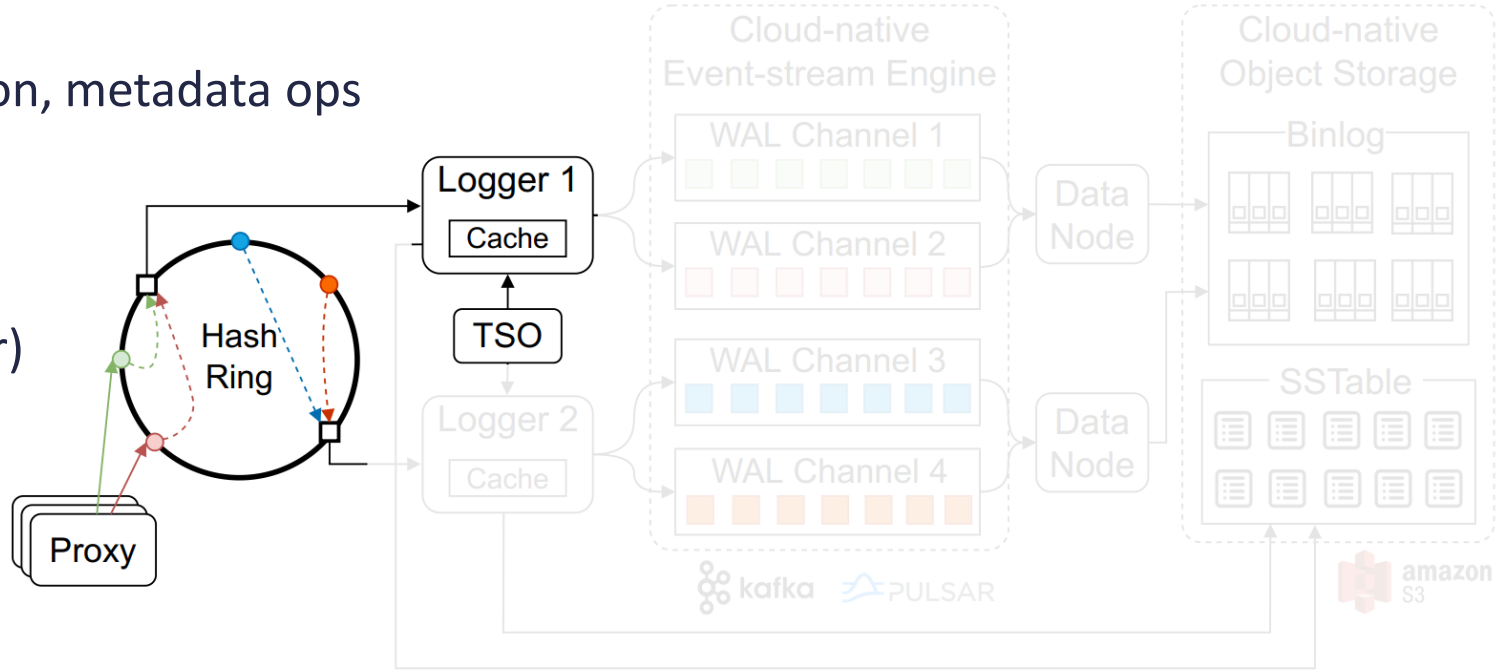
LOG WRITING

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request



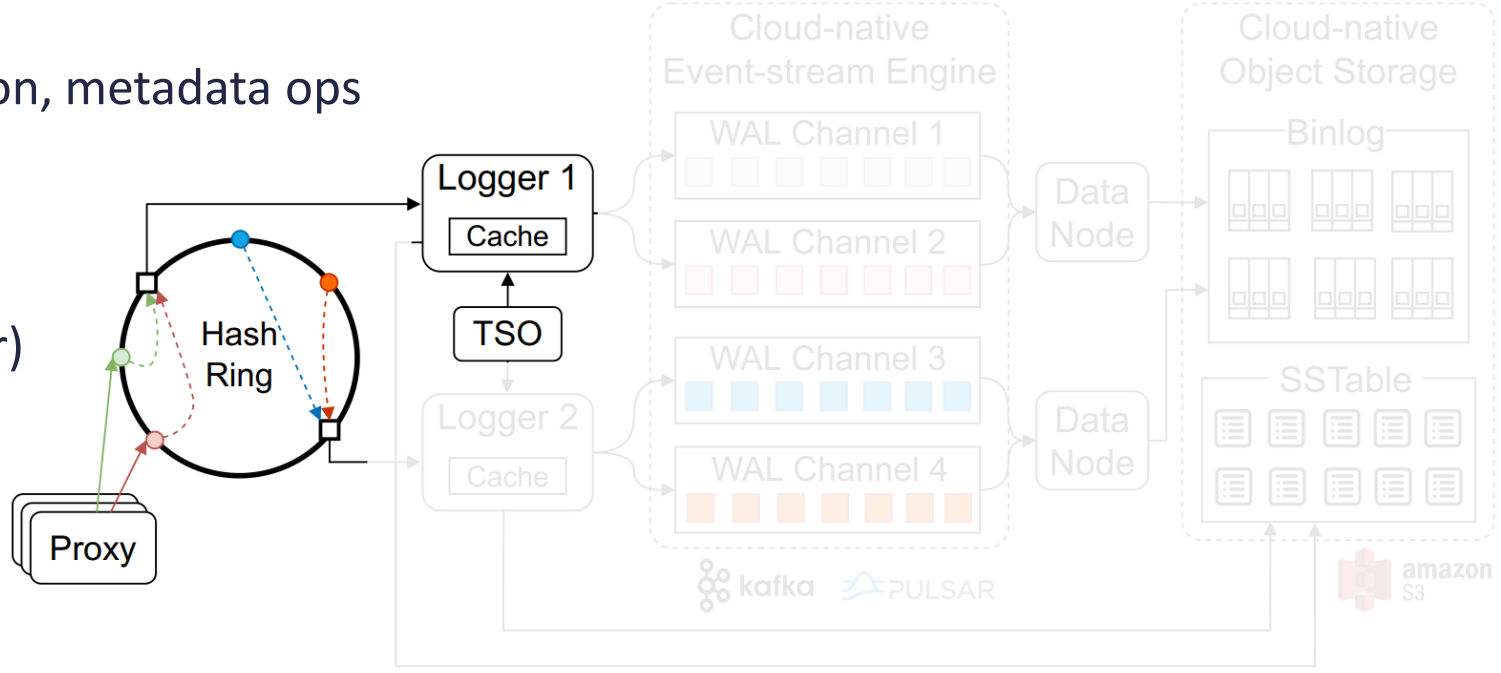
LOG WRITING

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request
  2. Assign LSN (sequence number)
     - Uses timestamp oracle server



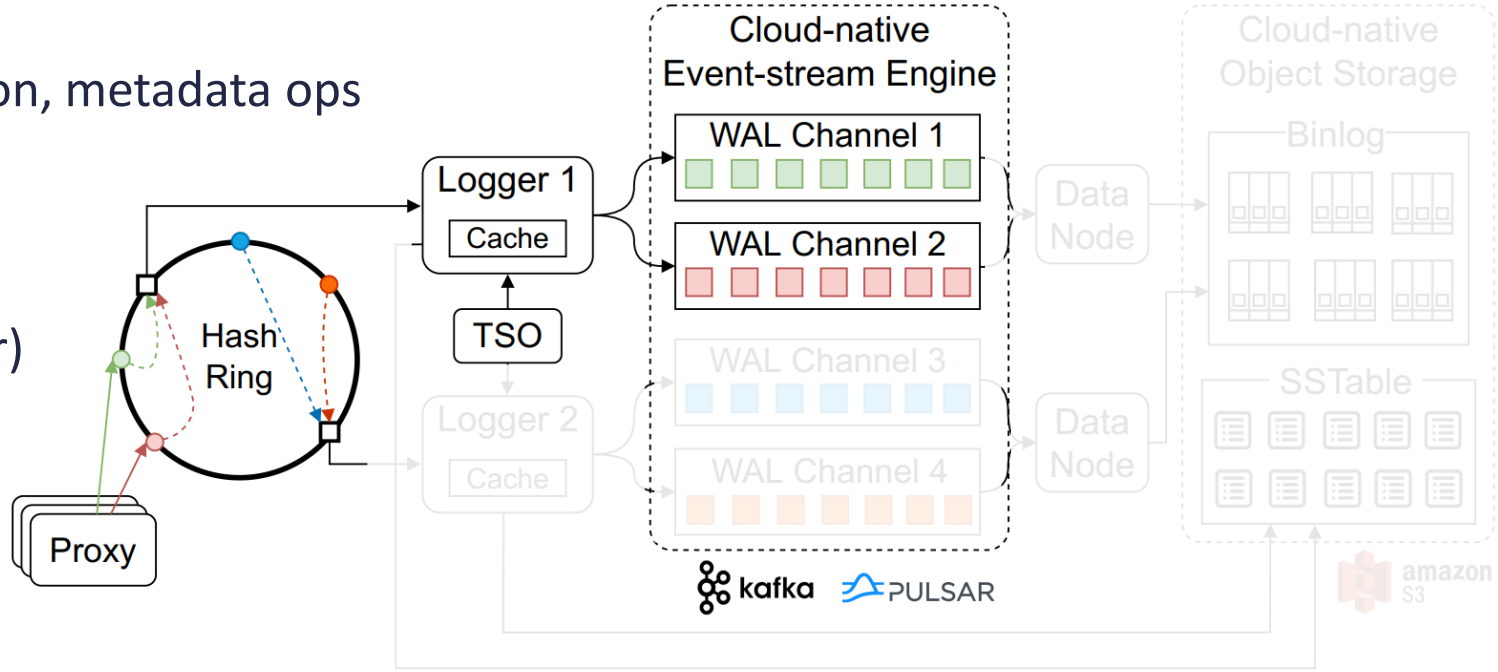
LOG WRITING

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request
  2. Assign LSN (sequence number)
     - Uses timestamp oracle server
  3. Assign key to a segment



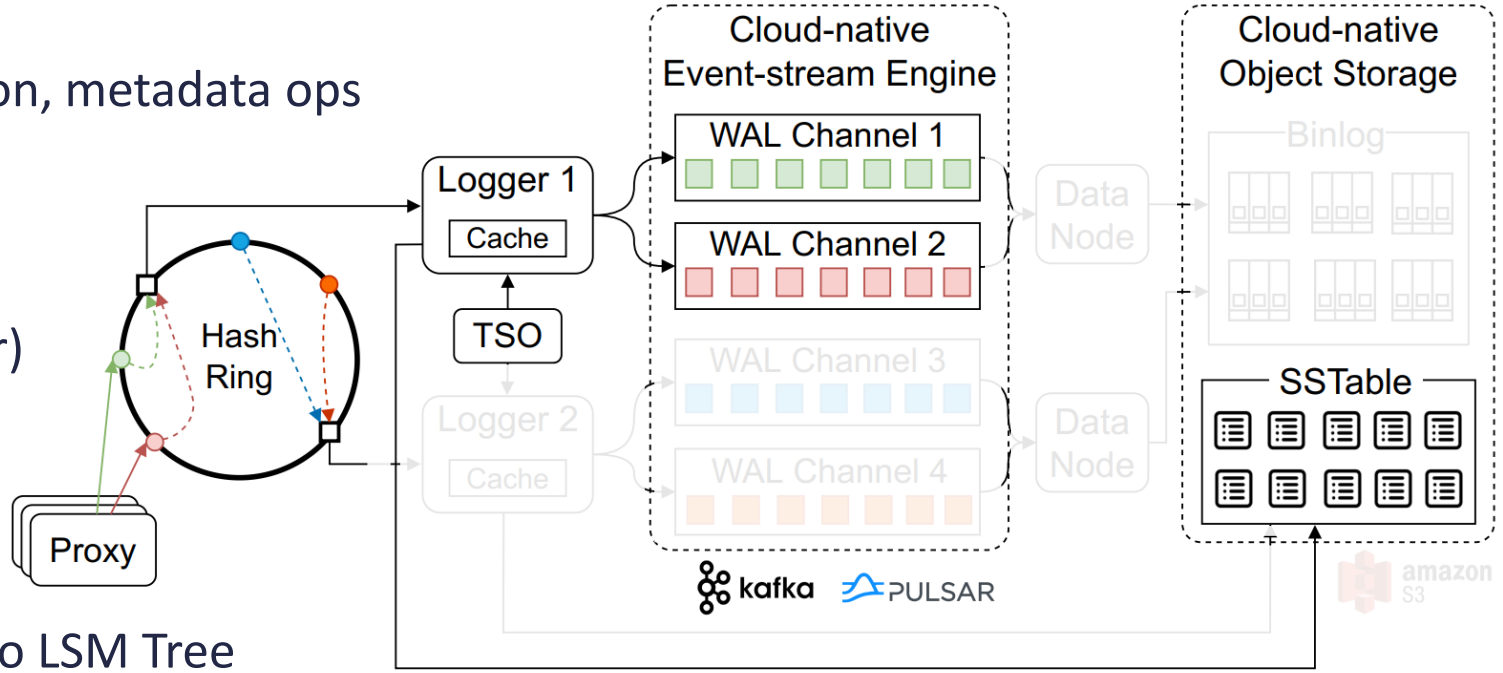
LOG WRITING

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request
  2. Assign LSN (sequence number)
     - Uses timestamp oracle server
  3. Assign key to a segment
  4. Write to WAL



LOG WRITING

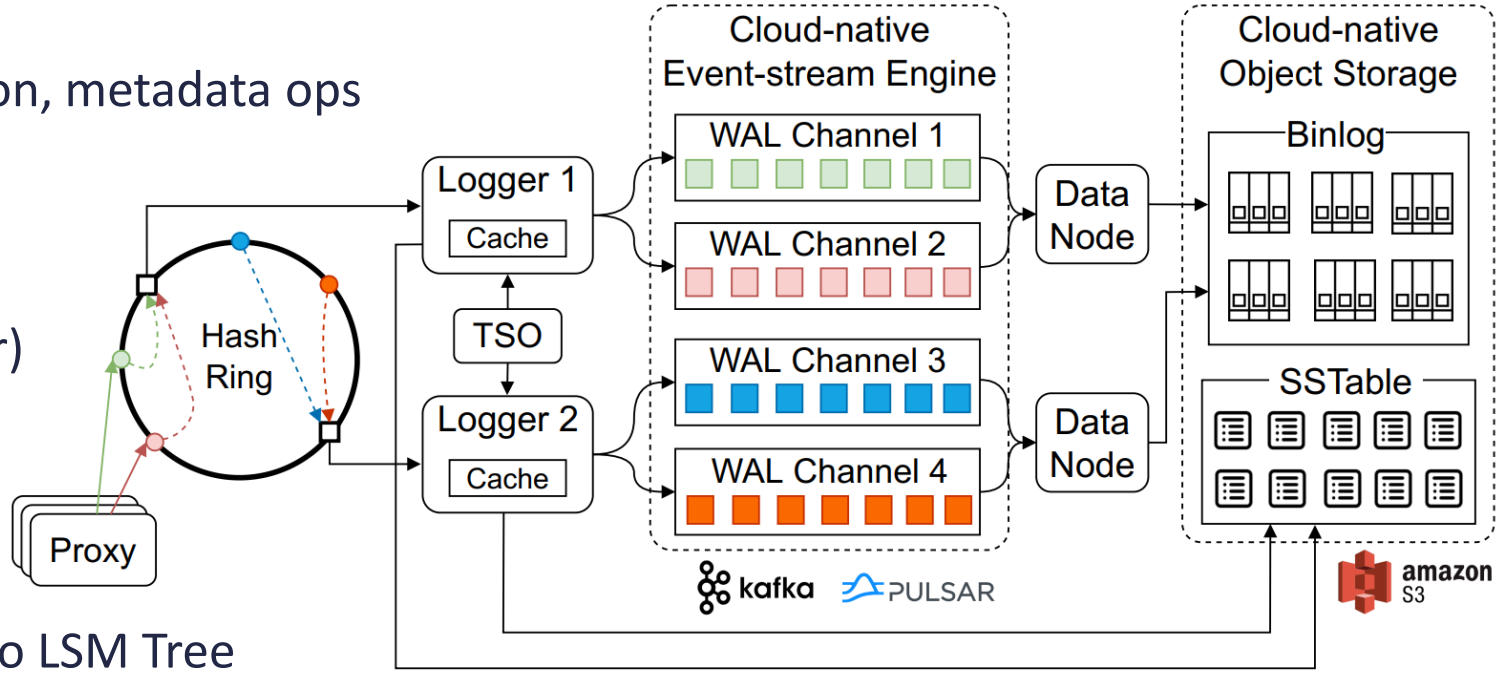
- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request
  2. Assign LSN (sequence number)
     - Uses timestamp oracle server
  3. Assign key to a segment
  4. Write to WAL
  5. Write key-segment mapping to LSM Tree



LOG WRITING

- Hash ring of channels
  - Channel per shard = hash(key)
  - Special channels for configuration, metadata ops
- Insert handling:
  1. Verify request
  2. Assign LSN (sequence number)
     - Uses timestamp oracle server
  3. Assign key to a segment
  4. Write to WAL
  5. Write key-segment mapping to LSM Tree

→ Insert can be acknowledged... **but journey is not over**



INSERT PATH

Proxy:

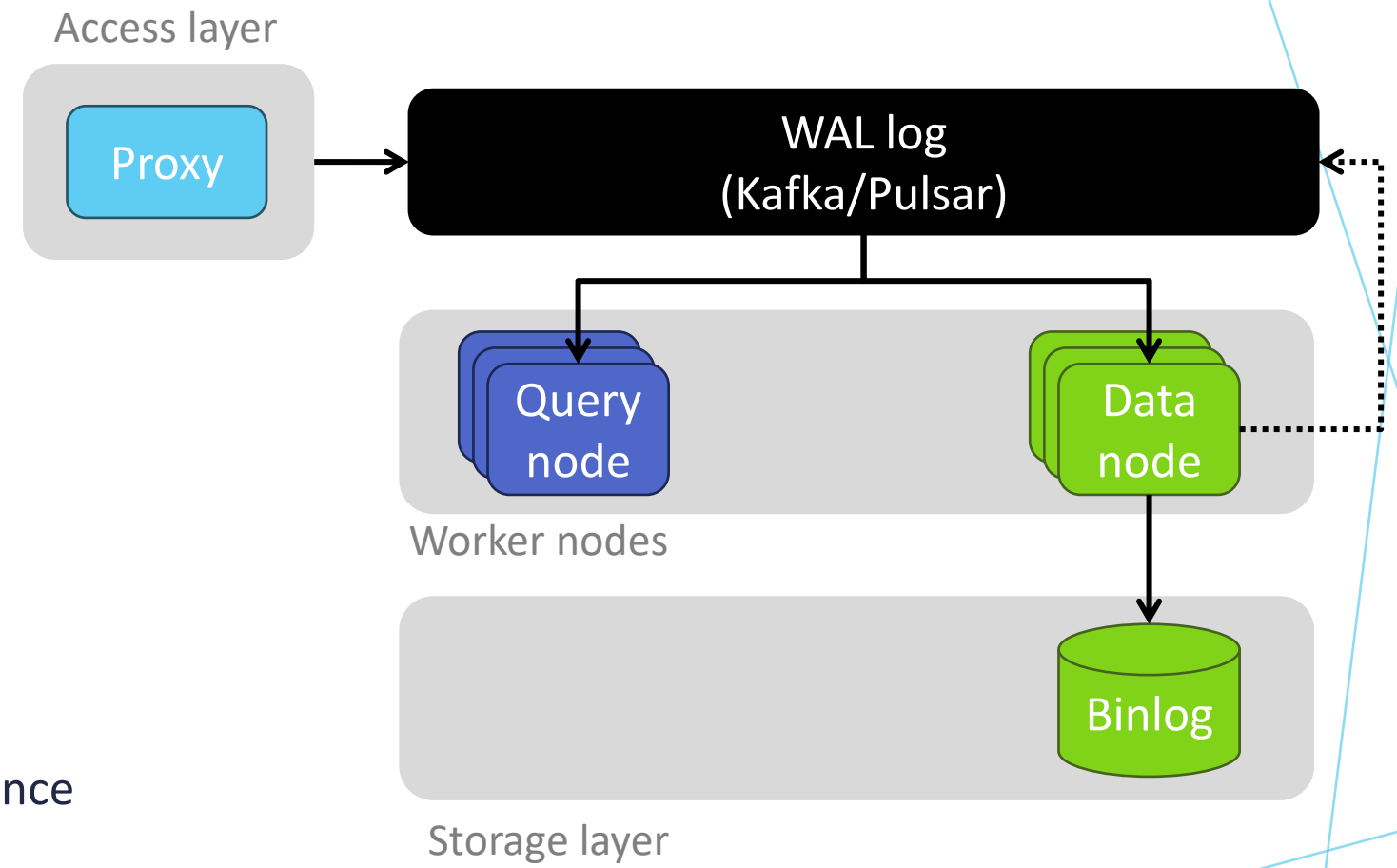
1. Insert to WAL

Query node:

2. Receive from channel
3. Update growing segment in RAM
   - Build slice index as needed

Data node:

2. Receive from channel
3. Convert to columnar binlog
4. Write to object store
5. If needed: seal segment and announce



INCREMENTAL INDEXING PATH

Data coordinator:

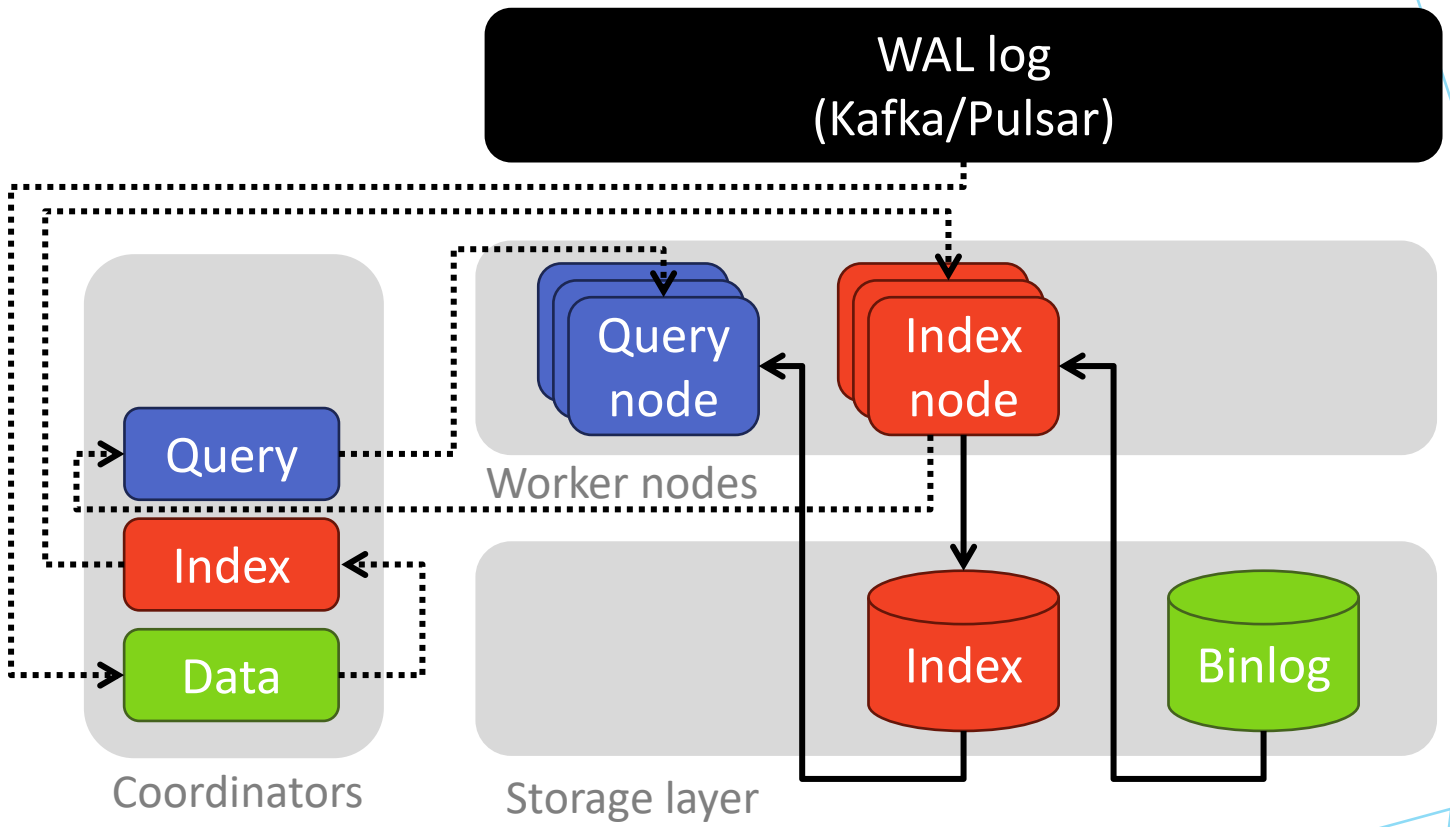
1. Receive seal announcement
2. Tell index coordinator to build

Index:

3. Coordinator assigns index node
4. Loads segment from binlog
5. Build index and store
6. Announce to query coordinator

Query:

7. Coordinator updates node
8. Load index to RAM



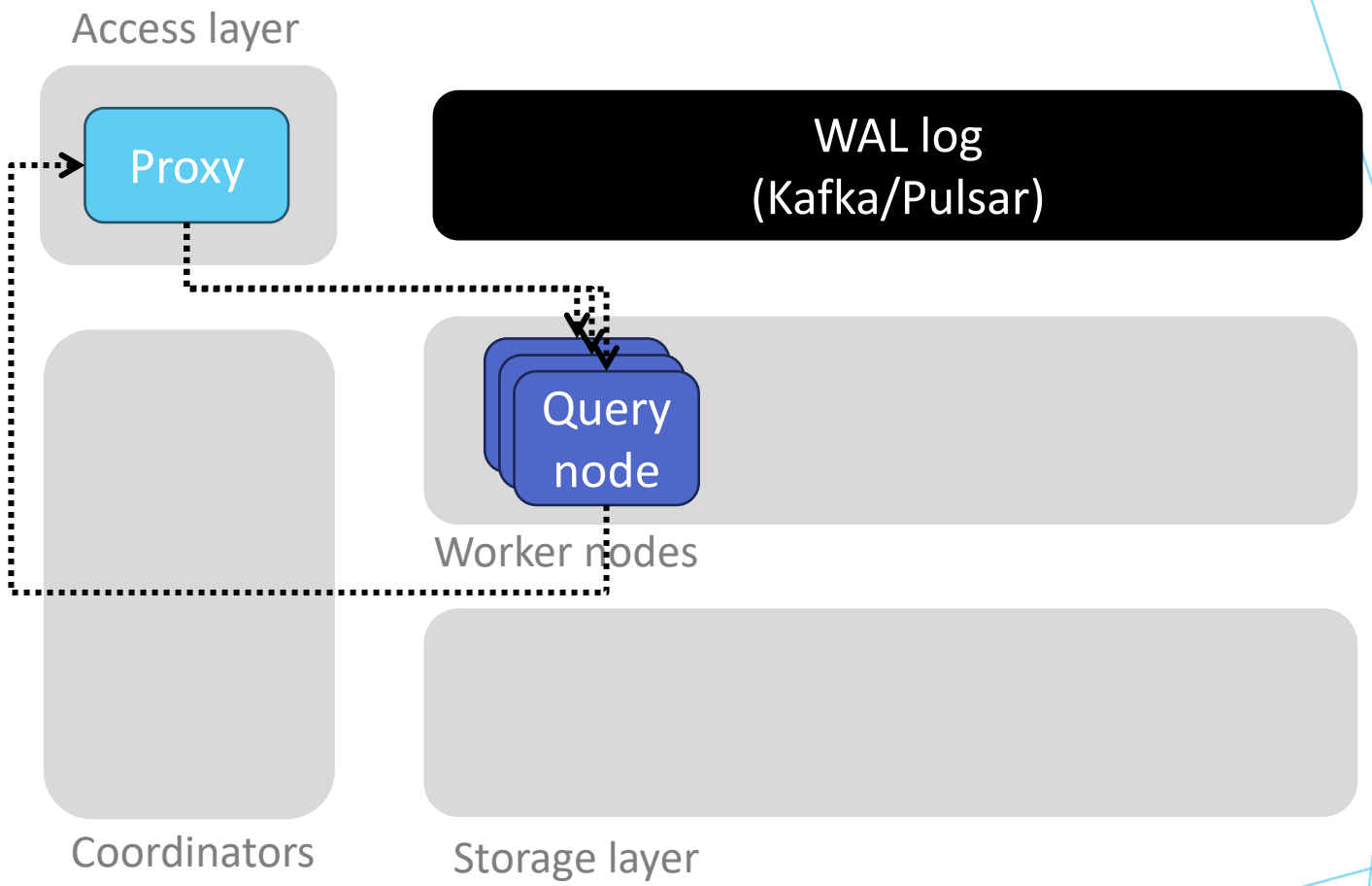
…SO COMPLICATED!

… But there are benefits:

- Insert acknowledged quickly (after WAL)
- Reads proceed independently (next slide)
- Liveness layer = WAL + growing segments (in query node RAM & binlog)
- Fault tolerance (persistent WAL & binlog).
- Log provides coordination (consistent time semantics).
- Scale each functionality/component independently.
- Can add new analytics (just subscribe to binglog/WAL).

SIMPLE QUERY PATH

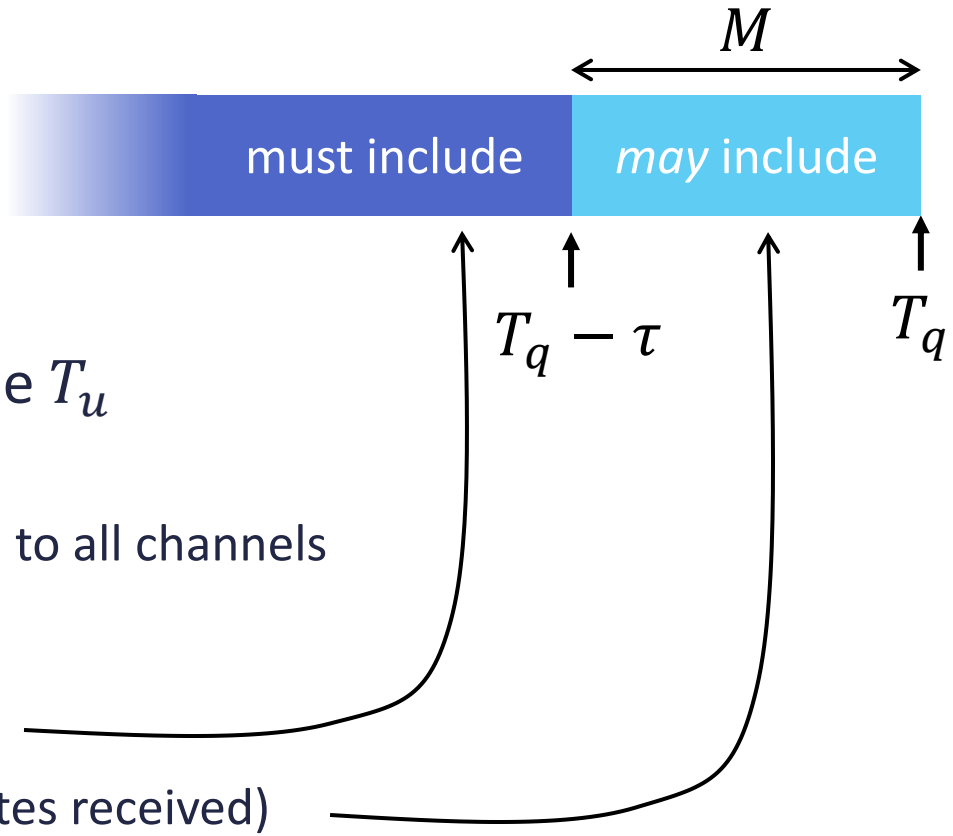
1. Proxy contacts query nodes
   - Use cached metadata
2. Nodes run kNN in parallel
   2. Run kNN on all segments
   3. Each returns top k to proxy
3. Proxy aggregates top k
4. Return result



Based on [Guo, VLDB '22], Milvus Docs

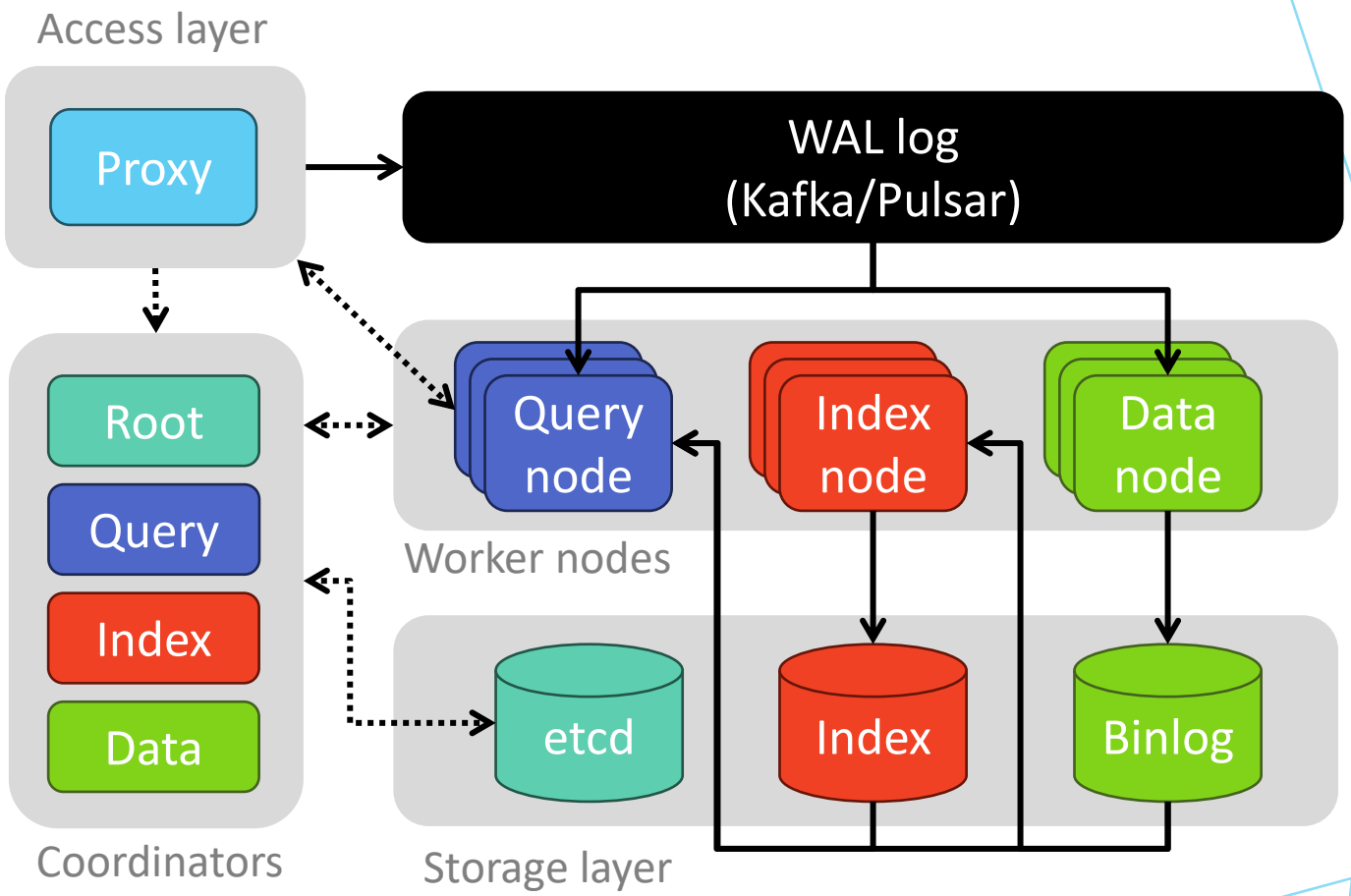
DELTA CONSISTENCY

- Query time T_q
- Users specify max staleness τ
  - Must include all updates **before** $T_q - \tau$
- Query nodes keep track of last update time T_u
  - Available from LSN on data
  - **Inject periodic ticks** (watermarks, heartbeats) to all channels
- Delay execution:
  - $T_u < T_q - \tau \Rightarrow$ must wait (updates missing)
  - $T_u > T_q - \tau \Rightarrow$ can execute (all relevant updates received)



OTHER CONSIDERATIONS

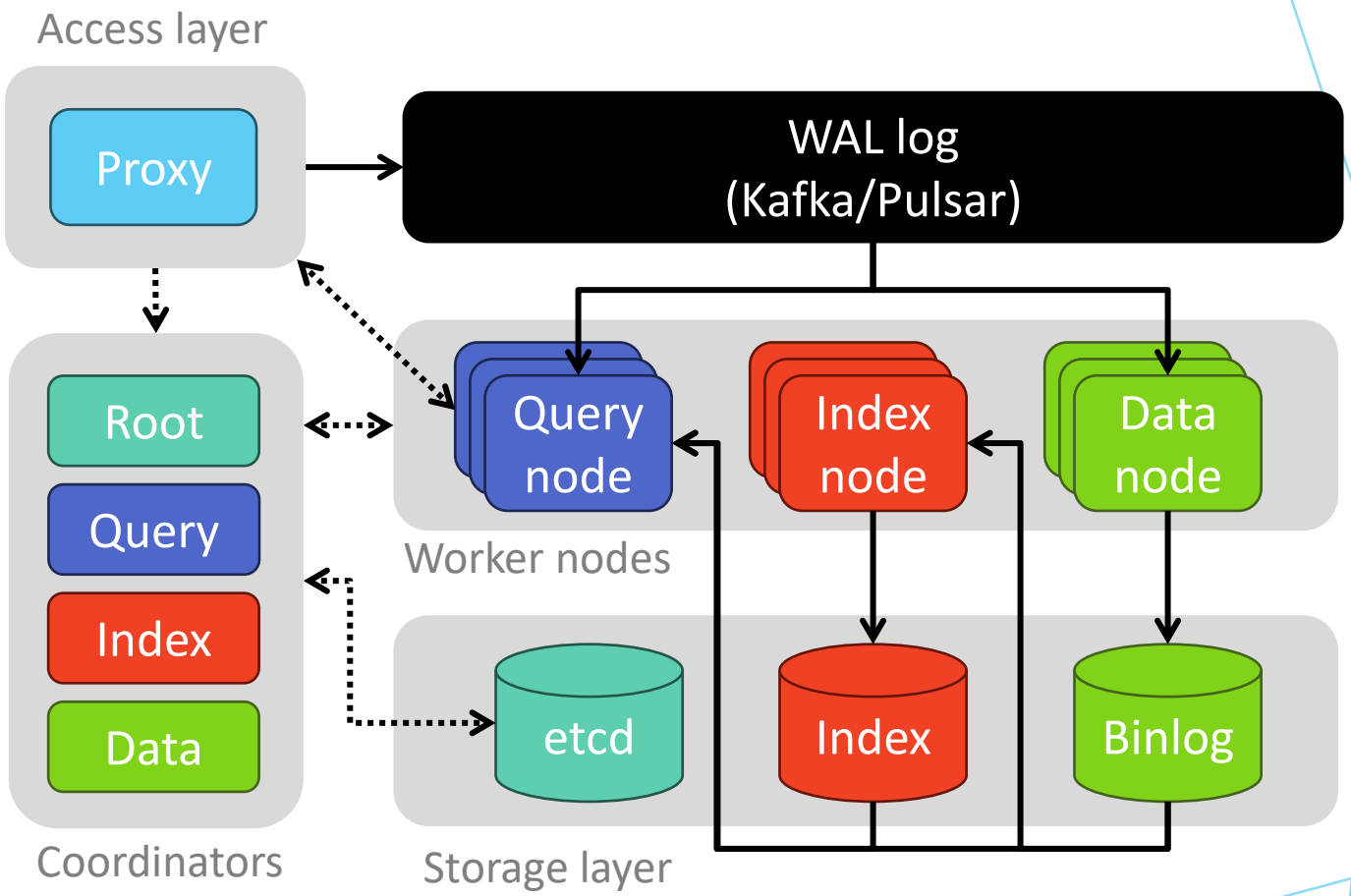
- Full index rebuild:
  - Get segment paths from data coord.
  - Assign index nodes.
- Delete/updates:
  - Per-segment bitmaps.
  - Rebuild segments as needed.
- Segment reassignment:
  - Scaling, load-balance, recovery.
  - Managed by query coordinator.



Based on [Guo, VLDB '22], Milvus Docs

MORE IN THE PAPER

- Filtered queries.
- Multi-vector queries
- Bayesian optimization for index configuration.
- SIMD, GPU optimization.
- Time travel.
  - Restore database to time *T*.
- Fault tolerance.
- Monitoring, load balancing.



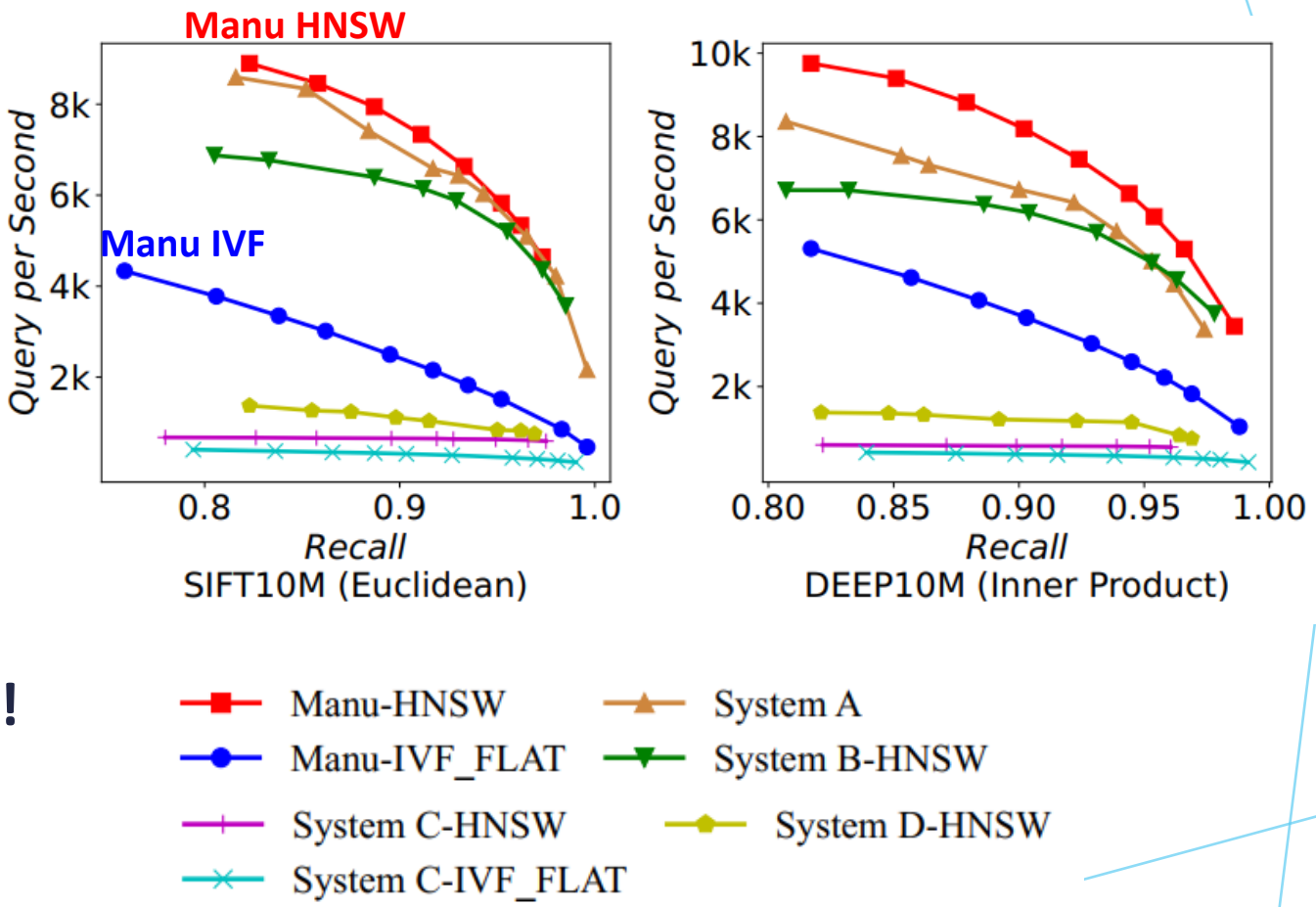
Based on [Guo, VLDB '22], Milvus Docs

RESULTS: SINGLE NODE PERFORMANCE

- Configuration:
  - Single node
  - EC2 m5.4xlarge (16 vCPU, 64 GB RAM)
  - **Recall@50**
  - SIFT 10M, DEEP 10M
- **Excellent single node perf.**
- Graph-based A, B slightly slower

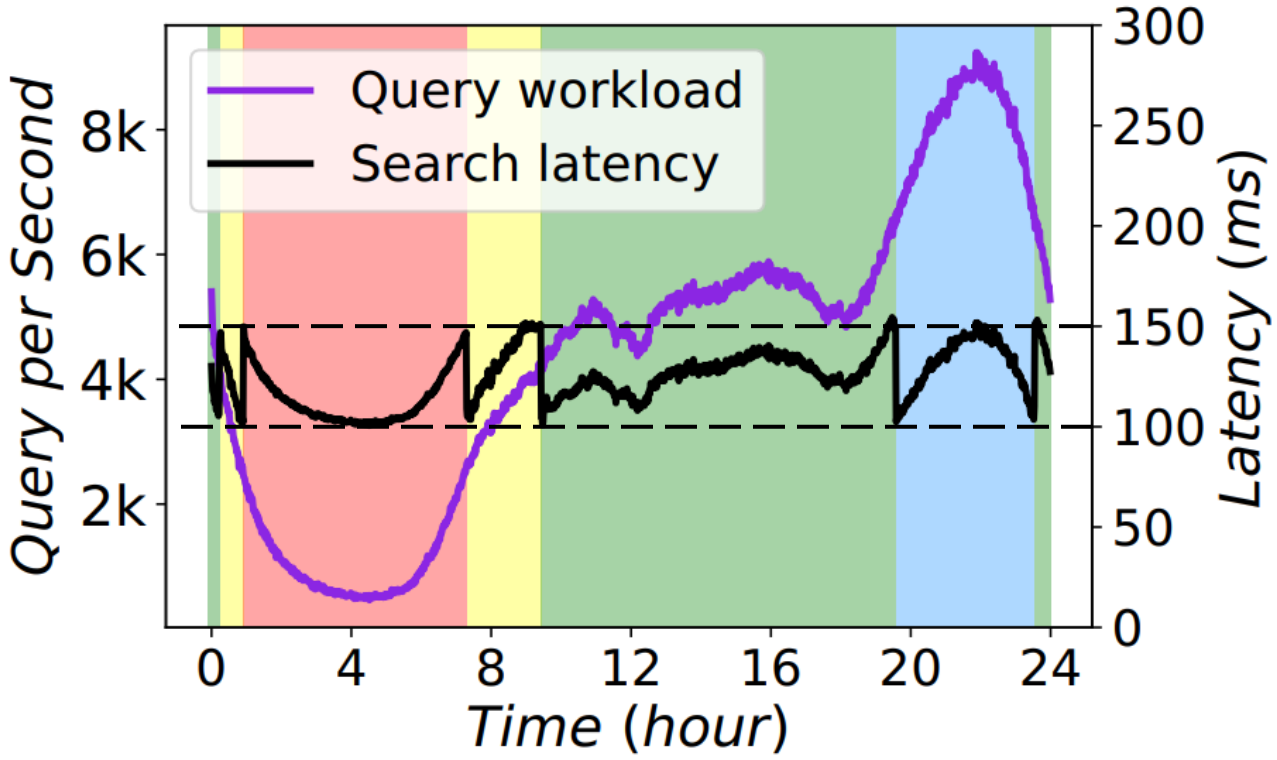
→ **SIMD, CPU optimizations matter!**

- Disk-based system D very slow



RESULTS: ELASTICITY

- Configuration:
  - 4 nodes: 2 data, 1 query, 1 index node.
  - EC2 m5.4xlarge (16 vCPU, 64 GB RAM)
  - Recall@50 > 80%
  - SIFT 100M
- Scale to maintain latency
  - < 100ms? Halve query nodes
  - > 150ms? Double query nodes
- 100ms seems high?
  - Recall@**50 ,** few nodes



SCALABILITY

- Segmenting allows linear scaling
  - With dataset size
  - With num query nodes
- Across datasets and indexes.
- Even with graph-based HNSW
- Want better? Increase segment size
  - Sub-linear complexity for graph index

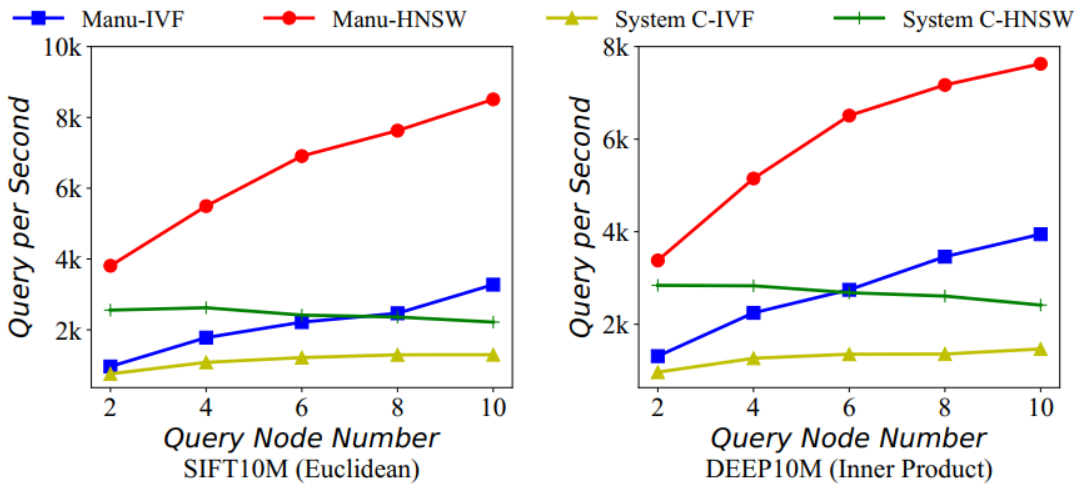


Figure 10: Scalability of Manu w.r.t. query nodes.

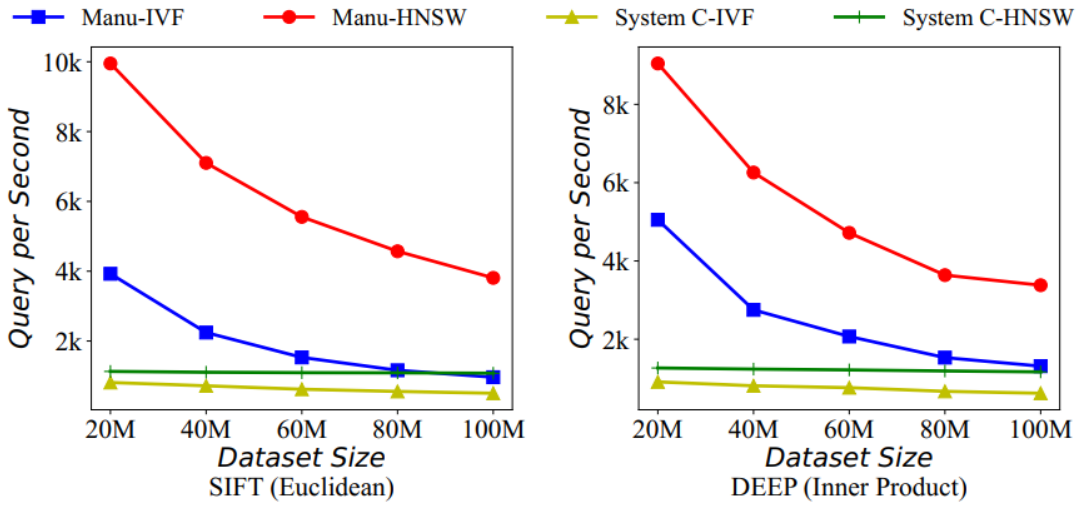
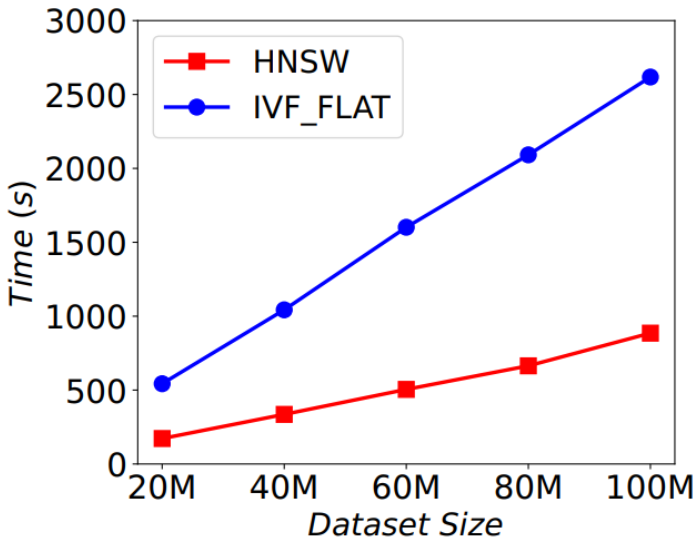


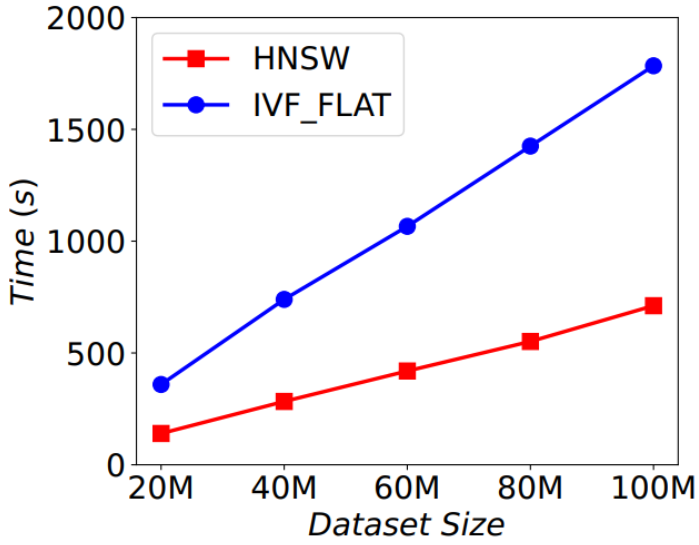
Figure 11: Scalability of Manu w.r.t. data volume.

INDEX BUILDING

- Configuration:
  - 1 index node.
  - EC2 m5.4xlarge
    (16 vCPU, 64 GB RAM)
- **< 1 hour for 100M vectors**.
- Effect on query?
  - Likely small.
  - Decoupled paths.
  - ***Possibly*** affects low τ
    (reading binlog delays inserts)



(a) SIFT (Euclidean)



(b) DEEP (Inner Product)

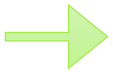
τ

THAT'S IT FOR TODAY!

- VecDBs are fascinating.
- Best parts of RDBMS +
  distributed systems +
  algorithms for high-dimensional data
- From next week: your presentations
- Lots more to research!
  - Very active area
  - Still young
  - Not (yet) crowded ☺

THAT'S IT FOR TODAY!

- VecDBs are fascinating.
- Best parts of RDBMS + distributed systems + algorithms for high-dimensional data
- From next week: your presentations
- Lots more to research!
  - Very active area
  - Still young
  - Not (yet) crowded ☺



- Incremental, disk-resident indexing
- Multitenancy and disaggregation
- Architectures
- HW assistance (GPU, SSDs)
- Filtered/hybrid queries
- Improving indexing
- Learning to index:
- Quantization