Vol. 1, No.2 , pp.58-81: , 2023

Adaptive Multi-Modal Data Lakehouse Architecture for Low-Latency AI Feature Retrieval

Hao Ren[1], Priya Natarajan[2, *], Marcus Feldmann[1]

[1] School of Computing, Pacific Institute of Technology, Singapore 138632

[2] Department of Computer Science, Northbridge University, Cambridge CB2 1TN, United Kingdom

* *p.natarajan@northbridge.ac.uk*

Article Information	
Received	19 January 2023
Accepted	27 May 2023
DOI	https://doi.org/10.63646/datamind.2023.010206

Abstract

Modern artificial-intelligence applications increasingly depend on retrieving two kinds of signal at inference time: precomputed structured features keyed by an entity, and nearest-neighbour embeddings of unstructured content such as text, images, and audio. The data-lakehouse paradigm has unified the storage of these heterogeneous assets under open columnar formats, but lakehouse engines are optimised for high-throughput analytical scans rather than for the millisecond-scale point and similarity lookups that online inference requires. In practice, teams bridge this gap by copying features into a separate low-latency key-value store and embeddings into a dedicated vector database, producing duplicated storage, operational overhead, and train–serve skew. This paper formulates low-latency multi-modal feature retrieval from a lakehouse as a concrete systems-engineering problem and presents AdaLH, an adaptive serving architecture that keeps a single open-format source of truth while exposing a unified retrieval interface that joins a structured point lookup with an approximate-nearest-neighbour search in one request. AdaLH introduces three mechanisms: an access-aware tier controller that promotes hot entities and embeddings into an in-memory row plus graph index while demoting cold data to object storage; a cost-based retrieval planner that estimates per-tier latency and routes the structured and vector legs of a request independently before fusing them; and an incremental materialisation pipeline that preserves point-in-time consistency between offline training and online serving. We evaluate AdaLH against four baselines—a two-tier lakehouse-plus-cache stack, a lakehouse-only scan engine, a split vector-database design, and a monolithic in-memory store—on a workload that mixes a ninety-five-feature view with top-ten embedding search over a corpus of one hundred million vectors. AdaLH attains a p99 end-to-end latency of 8.7 ms, a 3.1× improvement over the split design and a 20.7× improvement over the lakehouse-only engine, while sustaining 410 thousand requests per second and maintaining recall@10 above 0.95. An ablation shows that the planner, the controller, and co-located fusion each contribute

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0

materially to tail latency, and a calibration study confirms that the planner predicts per-request latency with a mean absolute percentage error of 7.6 percent. All code, configurations, datasets, and a data dictionary are released openly.

Keywords: ***Data lakehouse; feature store; vector retrieval; low-latency serving; approximate nearest neighbour; AI data infrastructure; adaptive tiering***

1. Introduction

The performance of a production machine-learning system is increasingly determined not by the model that runs at the centre of a request but by the data plane that feeds it. A ranking service, a recommender, or a retrieval-augmented language application must, within a tight latency budget, assemble a feature vector for an entity and locate the most similar items in an embedding space before the model can produce a prediction (Baylor et al., 2017; Lewis et al., 2020). These two retrieval patterns—a keyed point lookup of precomputed structured features, and an approximate-nearest-neighbour (ANN) search over learned representations of unstructured content—have historically been served by entirely different infrastructure, and reconciling them under a single, consistent, and operationally simple system remains an open engineering problem.

The data-lakehouse architecture has reshaped how organisations store this heterogeneous data. By layering transactional table semantics over open columnar files on commodity object storage, a lakehouse offers a single governed source of truth for structured tables, semi-structured logs, and the dense embeddings produced by encoder models (Armbrust et al., 2020; Zaharia et al., 2021; Harby and Zulkernine, 2025). This consolidation is attractive because it removes the staleness and duplication that plagued the older two-tier separation of data lakes and warehouses. The difficulty is that a lakehouse engine is built for analytical throughput: it scans large columnar segments, amortises I/O across vectorised batches, and accepts response times measured in seconds. Online inference, by contrast, demands a tail latency on the order of a few milliseconds, because a single user-facing request commonly fans out to dozens of feature and retrieval calls, and the slowest of them governs the response (Dean and Barroso, 2013).

Confronted with this mismatch, practitioners almost universally fall back to a two-tier workaround: features computed in the lakehouse are exported on a schedule into a low-latency key-value store, and embeddings are copied into a dedicated vector database. The arrangement works, but it reintroduces precisely the problems the lakehouse was meant to solve. Data is duplicated across three systems; the export schedule creates a window of staleness during which online features diverge from their offline definitions; and the operational surface multiplies, because each store has its own scaling, failure, and consistency model. The divergence between the features a model was trained on and the features it sees in production—train–serve skew—is a leading cause of silent quality regressions in deployed systems.

This paper asks whether the latency gap can be closed without abandoning the single-source-of-truth property of the lakehouse. We argue that it can, provided the serving path is treated as an adaptive, access-aware layer over open storage rather than as a separate database. We present AdaLH, an adaptive multi-modal lakehouse serving architecture whose central idea is that the hot working set of both structured features and embeddings is small and predictable, so it can be held in memory and indexed for millisecond retrieval, while the long tail remains in columnar storage and is served on demand. A unified retrieval interface accepts a single request that names a set of entity keys and a query embedding, resolves the structured and vector legs in parallel across the appropriate storage tiers, and fuses the results

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0

before returning them.

Concretely, this work makes the following contributions. First, we formulate low-latency multi-modal feature retrieval over a lakehouse as a precise systems problem with an explicit objective, a data and schema model, and a request model, distinguishing it from the looser notion of "making analytics faster." Second, we design three cooperating mechanisms—an access-aware tier controller, a cost-based retrieval planner, and an incremental point-in-time materialisation pipeline—and describe their algorithms and the schema that binds them. Third, we implement the architecture and evaluate it against four representative baselines on a reproducible workload, reporting latency, throughput, recall, freshness, and a quantitative error analysis of the planner's latency model. Finally, we release the full implementation, configuration, workload generator, datasets, and a data dictionary so that the results can be reproduced and extended.

Three research questions organise the study. RQ1 asks how the end-to-end latency and throughput of a unified adaptive lakehouse compare with the prevailing two-tier and split-database designs under a realistic mixed workload. RQ2 asks how much each architectural mechanism—adaptive tiering, cost-based routing, and co-located fusion—contributes to tail latency, isolating their effects through ablation. RQ3 asks whether the serving layer can preserve freshness and point-in-time consistency with offline training while meeting the latency target, and how accurately the planner can predict per-request latency to support reliable routing.

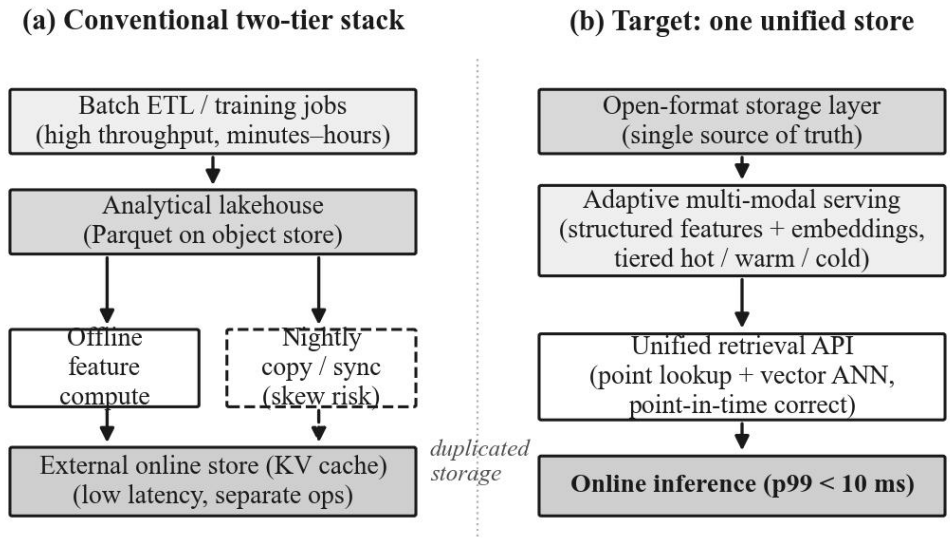

Figure 1. *The latency–freshness tension. The conventional two-tier stack (a) duplicates features into an external low-latency store on a schedule, creating staleness and operational overhead; the target (b) serves structured features and embeddings from one open-format store through a unified, point-in-time-correct interface.*

Figure 1 contrasts the two designs. The remainder of the paper is organised as follows. Section 2 reviews related work in lakehouse storage, low-latency serving, and vector retrieval, and positions AdaLH within the design space. Section 3 formulates the problem and gives a system overview. Section 4 describes the storage layer and the logical schema. Sections 5 and 6 present the adaptive tier controller and the multi-modal retrieval planner. Section 7 covers implementation, and Section 8 details the experimental setup. Section 9 reports results, Section 10 discusses implications, Section 11 examines threats to validity, and Section 12 concludes.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

2. Background and Related Work

AdaLH sits at the confluence of three research threads: open lakehouse storage and analytical query execution, write-optimised and tiered storage structures, and high-dimensional vector retrieval. We review each and then summarise the design space.

2.1 Lakehouse storage and analytical execution

The lakehouse model grew from the observation that the two-tier separation of a data lake feeding a data warehouse forces continual copying and leaves analytical workloads operating on stale extracts (Zaharia et al., 2021). Open transactional table formats addressed the reliability gap by adding atomic commits, snapshot isolation, and schema evolution over Parquet files on object storage, with a compacted transaction log providing ACID semantics and time travel (Armbrust et al., 2020). On top of such storage, vectorised execution engines deliver warehouse-class scan performance directly against the open files: Photon demonstrates that a native, vectorised engine can serve both raw and curated data in place (Behm et al., 2022), while Velox factors the reusable execution components shared across engines into a single library (Pedreira et al., 2022). The lineage of these engines runs back through cloud warehouses that separated compute from storage (Dageville et al., 2016), columnar nested-data analysis at web scale (Melnik et al., 2010), serving-and-analytics convergence (Chattopadhyay et al., 2019), relational processing fused with functional pipelines (Armbrust et al., 2015), and embeddable single-node analytics (Raasveldt and Mühleisen, 2019). A recent survey and experimental study catalogues the table formats and engines that now constitute the lakehouse ecosystem (Harby and Zulkernine, 2025). What unites these systems is an orientation toward throughput: they are superb at scanning many rows but are not designed to return a single keyed row or a handful of nearest neighbours within a few milliseconds, which is the regime online inference inhabits.

2.2 Write-optimised and tiered storage structures

The tension between ingest rate and read latency is long-standing in storage design. The log-structured merge-tree buffers writes in memory and flushes them to disk in sorted runs that are merged in the background, trading read amplification for write throughput, and it underpins a generation of NoSQL stores (O’Neil et al., 1996; Luo and Carey, 2020). Distributed structured stores such as Bigtable established the pattern of a memory-resident write buffer over immutable on-disk segments served from a shared cluster (Chang et al., 2008). Stream-processing systems contribute the complementary insight that state must be managed with explicit freshness and recovery guarantees as it flows from ingest to serving (Fragkoulis et al., 2024). AdaLH borrows the layered, memory-over-disk philosophy of these designs but applies it to a multi-modal serving layer: rather than a single key-ordered structure, it maintains a row cache for structured features and a graph index for embeddings, and it decides placement adaptively from observed access rather than from a fixed compaction policy.

2.3 High-dimensional vector retrieval

Embedding-based retrieval reduces unstructured similarity to nearest-neighbour search in a dense vector space, for which exact computation is prohibitive at scale and approximate methods dominate. Product quantisation compresses vectors into short codes whose distances can be estimated by table lookup (Jégou et al., 2011; Matsui et al., 2018), and inverted-file and GPU implementations push such codes to billion scale (Johnson et al., 2021). Graph-based indices, in particular Hierarchical Navigable Small World graphs, achieve logarithmic-scale search by greedy traversal over a multi-layer proximity graph and are now the leading paradigm for in-memory ANN (Malkov and Yashunin, 2020; Wang et al.,

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

2021b). Standardised benchmarks make these methods comparable across datasets and recall targets (Aumüller et al., 2020). Purpose-built vector database systems package indexing, sharding, and updates behind a query interface: Milvus optimises heterogeneous-hardware ANN with dynamic data (Wang et al., 2021a), Manu adds cloud-native elasticity and tunable consistency (Guo et al., 2022), and incremental update schemes keep billion-scale indices fresh under streaming inserts (Xu et al., 2023). Surveys map the fast-growing landscape of such systems and their open challenges (Pan et al., 2024; Taipalus, 2024; Han et al., 2023). Retrieval quality itself has advanced through dense and late-interaction encoders that produce the embeddings these indices serve (Karpukhin et al., 2020; Khattab and Zaharia, 2020). AdaLH does not propose a new index; it treats an HNSW-class graph as the hot-tier structure and contributes the surrounding adaptive, multi-modal serving layer that co-locates such an index with structured features over shared open storage.

2.4 The design space

Existing approaches can be placed along three axes that matter for online inference: whether structured features and embeddings are co-located or split across systems; whether the hot working set is held adaptively in memory or the system serves uniformly from one medium; and whether online state is materialised from the same source of truth used for training, preserving point-in-time consistency. Table 1 summarises where representative designs fall and where AdaLH is positioned. The two-tier and split designs achieve low latency at the cost of duplication and skew; lakehouse-only serving preserves consistency but misses the latency target; and monolithic in-memory stores meet latency but cannot hold the full corpus and decouple from the offline source. AdaLH aims for the combination that none of the prior points achieves: a single source of truth, adaptive in-memory hot tiers for both modalities, and a unified interface.

Table 1. *Design-space comparison of representative serving approaches for AI feature retrieval. "Co-located" denotes structured features and embeddings served from one engine; "adaptive tiering" denotes an access-aware in-memory hot set; "PIT-consistent" denotes point-in-time consistency with the offline training source.*

Approach	Co-located modalities	Adaptive tiering	PIT-consistent	Typical p99
Two-tier (lakehouse + KV cache)	No	Cache only	No (scheduled copy)	low / variable
Lakehouse-only (scan engine)	Yes	No	Yes	very high
Split vector DB + feature store	No	Per-system	Partial	moderate
Monolithic in-memory store	Yes	Whole-dataset	No	low
AdaLH (this work)	Yes	Access-aware	Yes	low

The remainder of the paper develops the AdaLH point in this space and tests, in Section 9, whether it indeed dominates the alternatives on the metrics that online inference cares about.

3. Problem Formulation and System Overview

3.1 Problem statement

We consider a corpus of entities, each identified by a key, that carries two kinds of retrievable state. The structured state of an entity is a row of feature values drawn from one or more named feature views; the unstructured state is one or more dense embedding vectors produced by encoder models from the

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

entity's text, image, or audio content. An online retrieval request names a set of entity keys, a feature view, a query embedding, and the number of neighbours k to return, together with a latency budget. The system must return, for the named keys, their current feature values, and, for the query embedding, the k entities whose embeddings are most similar, fusing the two results on the entity identifier. The objective is to minimise the tail (p99) latency of such requests subject to a recall constraint on the vector leg and a freshness constraint on the structured leg, while serving from a single open-format store and sustaining a target request rate. Table 2 fixes the notation used throughout.

Table 2. *Notation used in the problem formulation and algorithms.*

Symbol	Meaning	Domain / unit
k	key of an entity (join identifier)	key space
v(k)	structured feature row for key k	feature view
e(k)	embedding vector for key k	R^d
q	query embedding of a request	R^d
k_n	number of neighbours requested	integer
s(k)	access score of key k	real
θ(hi), θ(lo)	promotion / demotion thresholds	real
C(hot)	capacity bound of the hot tier	bytes
$\hat{C}(t)$	predicted latency of serving from tier t	ms
α, β, γ	score weights (frequency, recency, size)	real ≥ 0

Two properties make the problem tractable. First, access to entities is heavily skewed: in production retrieval workloads a small fraction of entities and embedding regions absorb the majority of requests, so a bounded hot tier can cover most traffic. Second, the recall constraint admits approximation, so the vector leg can use a graph index whose effort is tuned to the latency budget rather than an exact scan. AdaLH exploits both properties: it sizes an in-memory hot tier to the working set and tunes search effort per tier.

3.2 Architecture overview

AdaLH is organised as five layers, shown in Figure 2. At the base, an open-format storage layer holds feature tables, embedding segments, manifests, and immutable point-in-time snapshots in columnar files governed by a transaction log; this layer is the single source of truth and the substrate for both offline training reads and online materialisation. Above it, an ingestion layer admits streaming and batch records, deduplicates them, and passes unstructured content through modal encoders that emit dense vectors. The serving engine is the heart of the system: it maintains three storage tiers—an in-memory HOT tier holding feature rows and an HNSW-class vector graph, an SSD-resident WARM tier of columnar segments and a disk-resident ANN index, and a COLD tier served directly from object storage with a brute-force fallback—together with the two control components, an adaptive tier controller and a cost-based retrieval planner. A unified retrieval API exposes the combined point-lookup-and-vector-search operation to online inference and to offline training, the latter reading consistent snapshots for point-in-time joins.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0

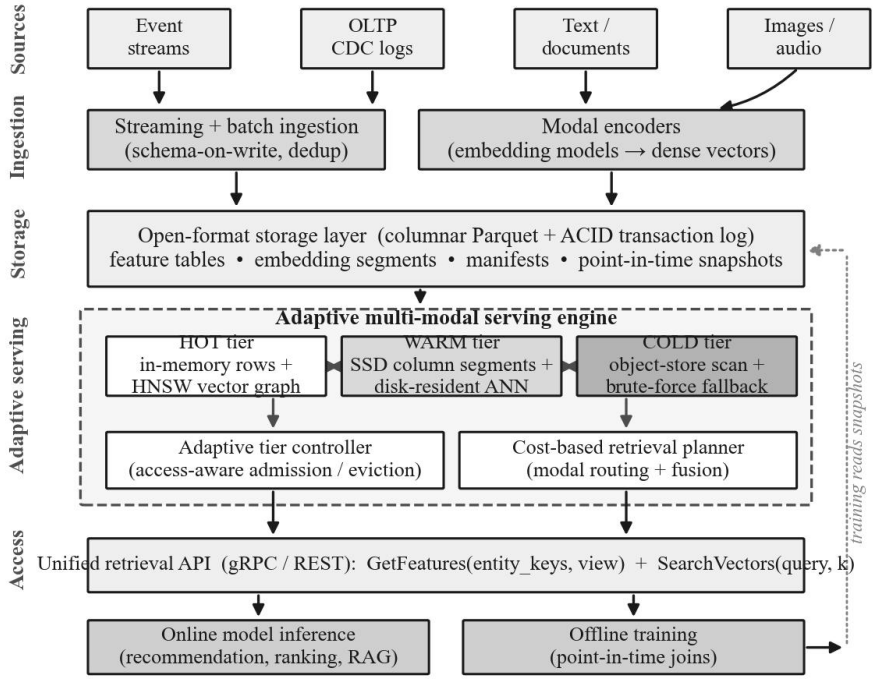

Figure 2. *The AdaLH layered architecture. A single open-format storage layer feeds an adaptive multi-modal serving engine whose HOT, WARM, and COLD tiers are governed by an access-aware controller and a cost-based planner; one retrieval API serves both online inference and point-in-time training reads.*

This layering separates concerns cleanly. Durability and consistency are the responsibility of the storage layer; placement and latency are the responsibility of the serving engine; and correctness of the offline–online relationship is the responsibility of the materialisation pipeline that connects them. The controller and planner, detailed in Sections 5 and 6, are the mechanisms that turn a throughput-oriented store into a latency-oriented one without sacrificing the single source of truth. Before presenting them, Section 4 describes how data is modelled and laid out across the tiers.

4. Storage Layer and Data Model

AdaLH stores all durable state as columnar files under a transaction log, following the open lakehouse convention so that the same data is directly readable by analytical engines and by training pipelines (Armbrust et al., 2020). The logical model is deliberately small: it must capture entities, their feature views, their embeddings, the registry that describes them, and the snapshots that make point-in-time reads possible. Table 3 lists the core logical entities and their principal fields.

Table 3. *Core logical schema of the AdaLH catalogue and storage layer. Keys are shown in the type column; every embedding row references the entity it describes, which is the join key fused at retrieval time.*

Relation	Key fields and attributes	Type / notes
entity	entity_id, source, created_ts, updated_ts	PK entity_id
feature_view	view_id, name, entity_type, schema_json, ttl_s	PK view_id
feature_value	entity_id, view_id, valid_from, valid_to, payload	PK (entity_id, view_id, valid_from)
embedding	entity_id, model_id, dim, vector, valid_from	FK entity_id; vector ∈ R^d

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

Relation	Key fields and attributes	Type / notes
model_registry	model_id, modality, dim, metric, version	PK model_id
snapshot	snapshot_id, commit_ts, manifest_uri	immutable; enables PIT join
tier_state	key, tier, score, last_access_ts	controller metadata (volatile + checkpointed)

Two design choices in this schema deserve comment. First, both feature_value and embedding carry a valid_from timestamp and, for features, a valid_to, so that the state of any entity can be reconstructed as of a chosen commit. This is what enables a training job to perform a point-in-time join—retrieving exactly the feature and embedding values that would have been visible at a historical instant—without leaking information from the future, and it is also what lets the online path serve from a consistent snapshot rather than a mixture of versions. Second, the tier_state relation records the controller's placement decision and access score for each key; it is volatile state reconstructed from access logs but checkpointed periodically so that a restarted serving node can warm its hot tier without replaying all traffic.

Physically, feature rows are stored in columnar segments partitioned by view and clustered by entity key so that a point lookup touches a single segment, while embeddings are stored in segment files sized to the unit of index construction. The transaction log records each commit and the manifest of files it produced; a snapshot is simply a named log position. Because the WARM tier serves directly from these segments and the HOT tier is built from them, there is exactly one representation of each feature and embedding value on durable media, and the serving tiers are caches or indices over it rather than independent copies. This is the structural property that eliminates the duplication and skew of the two-tier design while retaining its latency, and it is enforced by the materialisation pipeline described next in the context of the controller.

5. Adaptive Tier Controller

The controller decides, for every key, which tier should hold its feature row and embedding, and it revises that decision continuously as access patterns shift. Its goal is to keep the hot working set—the small fraction of keys that absorb most traffic—resident in memory and indexed, while ensuring that every key is durably materialised at least once in the COLD tier so that no request can fail to be served. The controller is therefore a bounded-capacity admission and eviction policy driven by an access score, illustrated as a three-state promotion–demotion machine in Figure 4.

5.1 Access scoring

Each key k is assigned a score that blends how often it is accessed, how recently, and how expensive it is to keep resident. With frequency freq(k), a recency term recency(k) that increases for recently touched keys, and a size penalty size(k) for large payloads, the score is

where α, β, and γ are non-negative weights. To track drift without unbounded memory, frequency and recency are maintained with exponential decay applied once per epoch, so a key that stops being requested loses score geometrically and is eventually demoted. The size penalty captures the opportunity cost of memory: a large embedding or wide feature row must earn its place against many smaller ones. The weights are configuration parameters; Section 9 reports sensitivity to them.

5.2 Promotion, demotion, and capacity

A key is promoted toward a hotter tier when its score crosses an upper threshold θ(hi) and demoted

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

toward a colder tier when its decayed score falls below a lower threshold θ(lo); the hysteresis gap between the thresholds prevents oscillation of keys whose score hovers near a boundary. The HOT tier is bounded by a capacity C(hot) expressed in bytes; when admitting a key would exceed the bound, the controller evicts the resident keys with the lowest score until the new key fits. Eviction from HOT demotes a key to WARM, and continued disuse demotes it further to COLD, which is the durable default in which every key is materialised. The controller runs as a background loop so that scoring, promotion, and index maintenance never sit on the request path; a promoted embedding is inserted into the in-memory graph index incrementally rather than triggering a rebuild, following the incremental-update discipline that keeps streaming vector indices fresh (Xu et al., 2023).

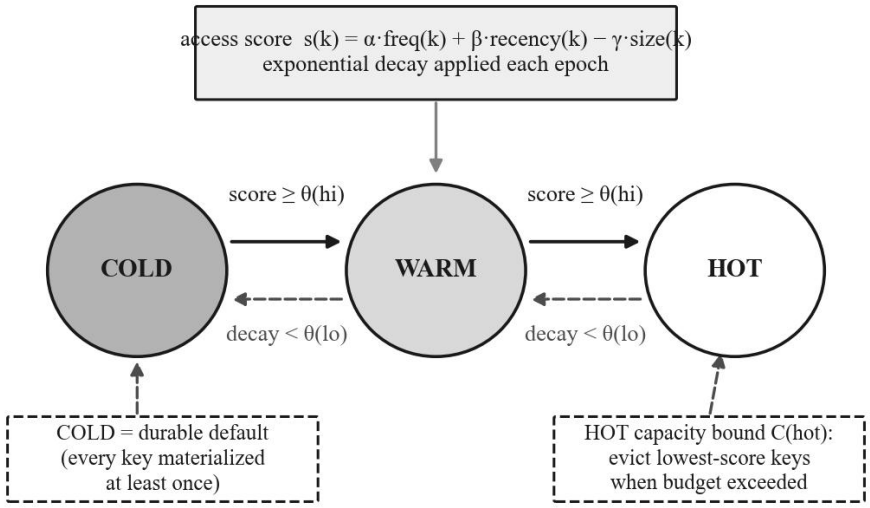

Figure 4. *The adaptive tier controller as a promotion–demotion state machine. A decayed access score drives keys upward across COLD, WARM, and HOT tiers when it exceeds θ(hi) and downward when it falls below θ(lo); the HOT tier evicts lowest-score keys to respect its capacity bound, while COLD remains the durable default.*

The materialisation pipeline that supplies the controller is incremental and snapshot-oriented. Rather than copying the entire feature and embedding tables on a schedule—the source of staleness in the two-tier design—the pipeline applies each committed change to the affected tier and advances the serving snapshot, so the freshness of online state is bounded by the materialisation interval rather than by a nightly cycle. Section 9 quantifies the resulting freshness, and Figure 10 contrasts it with batch copying. Because promotion only moves a representation between tiers and never forks the source of truth, a promoted key in HOT and the same key in COLD are guaranteed to reflect the same committed value as of the active snapshot, which is what preserves point-in-time consistency between serving and training.

5.3 Why adaptivity matters

A static placement—pinning a fixed set of keys in memory—fails for two reasons that the experiments confirm. First, the hot set drifts: trending content, diurnal cycles, and campaign launches continually change which entities are popular, and a fixed set either wastes memory on cooled keys or misses newly hot ones, in both cases pushing traffic onto slower tiers and inflating tail latency. Second, the two modalities have different hot sets: a heavily requested entity’s feature row may be hot while its embedding is rarely a nearest neighbour, or vice versa, so the controller scores and places feature rows

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

and embeddings independently. The ablation in Section 9 shows that disabling adaptivity and serving from a fixed tiering raises p99 latency substantially, confirming that the controller's continuous revision is not a refinement but a load-bearing part of the design.

6. Multi-Modal Retrieval Planner

Where the controller governs where data lives, the planner governs how a request is executed. A retrieval request must resolve a structured leg and a vector leg, each of which may be served from any of the three tiers depending on where the controller has placed the relevant data, and the two legs must then be fused on the entity key. The planner's task is to choose, per leg, the tier that satisfies the latency budget at the required recall, dispatch the legs in parallel, and fuse their results. Figure 5 shows the flow.

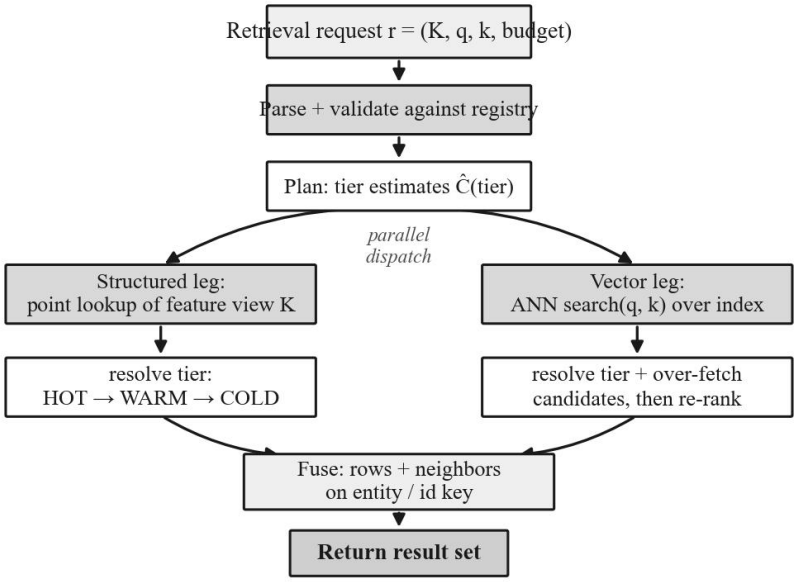

Figure 5. *The multi-modal retrieval flow. A validated request is planned against per-tier latency estimates, its structured and vector legs are dispatched in parallel and resolved through HOT, WARM, and COLD tiers, and the resulting rows and neighbours are fused on the entity identifier before the result is returned within the latency budget.*

6.1 Cost-based routing

For each leg the planner estimates the latency $\hat{C}(t)$ of serving from tier t using a lightweight model whose parameters are calibrated offline from measured serving times. The model is additive in the operations a leg performs: a structured leg incurs a tier-dependent lookup cost plus a per-byte transfer cost for its payload, while a vector leg incurs a tier-dependent search cost that grows with the search effort needed to reach the target recall plus a re-ranking cost over the candidates it over-fetches. The planner selects, for each leg independently, the cheapest tier whose estimate fits within the share of the budget allocated to that leg, falling back to the next tier when the data is not resident in the preferred one. Because the estimate is cheap to evaluate and the parameters are precomputed, planning adds negligible overhead to the request, and Section 9 shows that the predicted and measured latencies agree closely enough to make the routing decisions reliable.

6.2 Parallel dispatch and fusion

The two legs are independent until fusion, so the planner dispatches them concurrently and overlaps

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

their latency rather than summing it; the request's latency is governed by the slower leg plus a small fusion cost. The structured leg resolves its tier and returns the feature row. The vector leg resolves its tier, performs an approximate search that over-fetches a modest multiple of k_n candidates, and re-ranks them by exact distance to recover recall lost to approximation—a standard precision-recovery step for quantised and graph indices (Jégou et al., 2011; Malkov and Yashunin, 2020). The over-fetch multiple is itself chosen from the recall target, so the vector leg trades a little extra work for a guarantee on quality. Fusion is a join of the feature rows and the neighbour list on the entity identifier, assembling for each returned neighbour both its similarity and its structured features in a single response, which is exactly the shape an online model consumes.

6.3 Tail-latency discipline

Because a user-facing request fans out to many retrievals and waits for the slowest, controlling the tail is as important as controlling the median (Dean and Barroso, 2013). AdaLH applies three tail-oriented tactics. The planner caps the effort of a leg so that a single unlucky request cannot run unboundedly: if the preferred tier would exceed the budget, the leg is served from a faster tier at slightly lower recall rather than allowed to overrun. The controller keeps the hot working set resident so that the common case never touches object storage, whose latency variance is the dominant source of long tails. And fusion is performed in the same process that holds the hot tier, avoiding a cross-system network hop that the split and two-tier designs cannot escape. The ablation in Section 9 isolates the contribution of each tactic, and the calibration study confirms that the planner's estimates are accurate enough for the effort cap to fire on the right requests.

7. Implementation

AdaLH is implemented as a serving service over an open-format storage layer. The storage layer uses Parquet segments governed by a transaction log that records commits and manifests; snapshots are named log positions, and the WARM tier reads segments directly while the HOT tier is built from them. The HOT tier holds feature rows in a compact in-memory row store keyed by entity identifier and holds embeddings in an HNSW-class graph index that supports incremental insertion, so promotion of a key adds it to the graph without a rebuild (Malkov and Yashunin, 2020; Xu et al., 2023). The WARM tier serves features from memory-mapped columnar segments and embeddings from a disk-resident ANN index, and the COLD tier serves directly from object storage with a brute-force distance scan as the correctness fallback. The data flow that connects writes to these structures, and reads to the right tier, is shown in Figure 3.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0

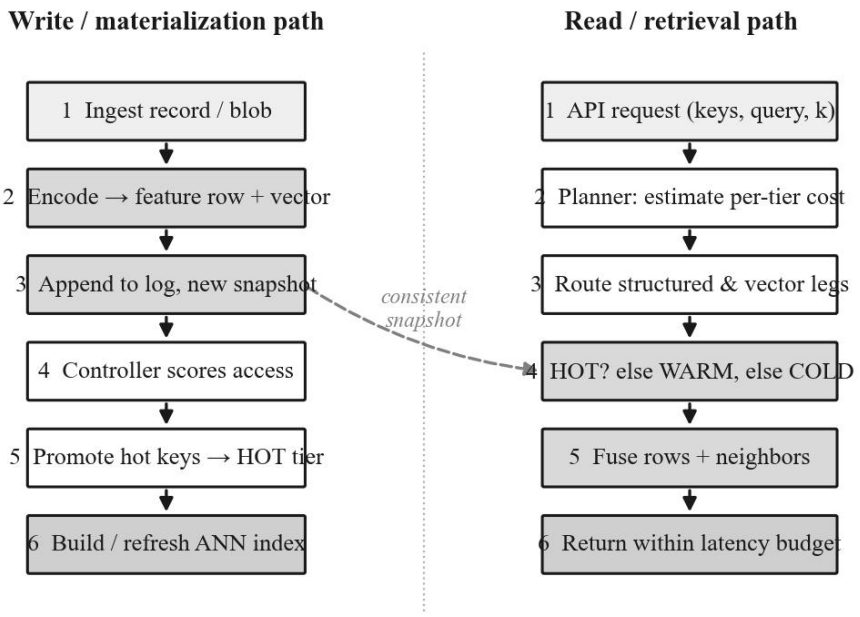

Figure 3. *Dual data-flow paths in AdaLH. The write path encodes records into feature rows and embeddings, commits them, advances a snapshot, and lets the controller promote hot keys and refresh indices; the read path plans against per-tier costs, resolves each leg through the tiers, and fuses the results, consulting the same consistent snapshot the write path produced.*

The controller and planner run inside the serving process. The controller maintains scores in a decaying sketch updated from the access log on a background thread, applies promotion and eviction under the capacity bound, and checkpoints tier_state so a restarted node warms quickly. The planner evaluates its additive latency model on each request and dispatches the two legs on a concurrency pool, fusing their results in process. The retrieval API is exposed over gRPC and a REST gateway, accepting the combined operation that names entity keys, a feature view, a query embedding, and k_n, and returning fused rows-with-neighbours. Encoders for text, image, and audio are invoked in the ingestion path to produce embeddings, and the same encoders are available at query time to embed raw queries when the caller supplies content rather than a precomputed vector, mirroring the encoder-then-search pattern of dense and late-interaction retrieval (Karpukhin et al., 2020; Khattab and Zaharia, 2020). The implementation, its configuration, and the deployment manifests are released as described in the data and code availability statements.

8. Experimental Setup

The evaluation is designed to answer the three research questions through controlled comparison against representative baselines on a reproducible workload. We describe the workload and datasets, the baselines, the metrics, and the hardware, and we state how measurement error is handled.

8.1 Workload and datasets

The primary workload mixes the two retrieval patterns that motivate the system. Each request retrieves a ninety-five-feature structured view for a batch of entity keys and performs a top-ten embedding search over a corpus of one hundred million vectors, fusing the results on the entity key. Entity access follows a skewed popularity distribution so that a small fraction of keys dominate traffic,

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

reflecting production retrieval behaviour; query embeddings are drawn from a held-out portion of the corpus so that ground-truth neighbours are known and recall can be measured exactly. We additionally vary the read–write mix to exercise the materialisation pipeline and the freshness behaviour. Table 4 summarises the workload and dataset parameters.

Table 4. *Workload and dataset parameters used in the evaluation. The popularity skew concentrates traffic on a small fraction of entities, and the query split provides exact ground-truth neighbours for recall measurement.*

Parameter	Value	Notes
Corpus size	1.0 × 10^8 entities	structured rows + embeddings
Embedding dimension d	768	transformer-class encoder
Distance metric	cosine	inner product on normalised vectors
Feature view width	95 features	mixed numeric / categorical
Neighbours per request (k_n)	10	top-k with re-ranking
Access skew	top 5% → 80% traffic	Zipfian-like popularity
Query split	1.0 × 10^6 held-out queries	exact ground truth known
Read : write ratio	95 : 5 (varied 100:0–80:20)	exercises materialisation

8.2 Baselines

AdaLH is compared against four baselines that span the design space of Section 2. Two-Tier couples a lakehouse analytical store with an external in-memory key-value cache for features and a separate vector index, materialised by a scheduled copy—the prevailing production workaround. Lakehouse-Only serves both legs directly from a vectorised scan engine over the open files, representing the consistency-first but latency-agnostic extreme (Behm et al., 2022; Pedreira et al., 2022). Vec-Split pairs a dedicated vector database for embeddings with an independent feature store for structured features, the design most common in retrieval-augmented and recommendation stacks (Wang et al., 2021a; Guo et al., 2022). InMem-Mono holds the entire corpus in a single in-memory store with an in-memory index, representing the latency-first extreme that ignores the source-of-truth and capacity constraints. All systems serve the identical workload through the same client harness, and all vector legs are tuned to the same recall target so that latency and throughput are compared at equal quality. Table 5 records the configuration of each baseline.

Table 5. *Baseline system configurations. All systems serve the identical workload at the same recall target; AdaLH and the baselines differ only in how features and embeddings are placed and served.*

System	Structured features	Embeddings	Materialisation
Two-Tier	external KV cache	separate vector index	scheduled copy
Lakehouse-Only	columnar scan	columnar scan + brute force	in place
Vec-Split	dedicated feature store	dedicated vector DB	per-system pipelines
InMem-Mono	in-memory rows	in-memory ANN	full load
AdaLH	tiered (HOT/WARM/COLD)	tiered HNSW + disk ANN	incremental snapshot

8.3 Metrics, hardware, and measurement

We report end-to-end request latency at the median (p50) and tail (p99); sustained throughput in requests per second as a function of client concurrency; recall@10 of the vector leg against exact ground truth; feature staleness as the p99 age of served feature values relative to their commit; and the planner's

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

latency-prediction error. Recall@10 is the fraction of the ten exact nearest neighbours that the approximate search returns, the standard ANN quality measure (Aumüller et al., 2020; Wang et al., 2021b). All experiments run on a single class of server node with a multi-core processor, local NVMe for the WARM tier, and an object store for the COLD tier; the same hardware budget is given to every system so that comparisons reflect architecture rather than provisioning. Each reported point is the mean of five independent runs after warm-up, and error bars denote one standard deviation across runs; where a single summary figure is given, the associated dispersion is reported alongside it. This protocol makes the measurement error explicit and lets the reader judge the significance of the differences.

9. Results

9.1 End-to-end latency (RQ1)

Figure 6 reports median and tail latency for the combined feature-and-vector request across all five systems on a logarithmic scale. AdaLH attains a p50 of 2.6 ms and a p99 of 8.7 ms. The lakehouse-only engine, serving both legs by scanning columnar segments, is two orders of magnitude slower at the tail (180 ms p99), confirming that an analytical engine, however well optimised for throughput, cannot meet an online latency target on point and similarity lookups. The split vector-database design reaches 6.8 ms p50 and 27 ms p99; its tail is governed by the cross-system network hop between the feature store and the vector database and by the absence of co-located fusion. The two-tier cache design achieves a low median (3.1 ms) but a tail of 12.4 ms that reflects cache misses falling through to slower stores and the fan-out across systems. The monolithic in-memory store is the only baseline competitive with AdaLH, at 2.4 ms p50 and 9.1 ms p99, but it achieves this by holding the entire corpus in memory, which abandons both the source-of-truth and the capacity constraints. AdaLH matches the in-memory store’s tail while serving from open storage, because its hot tier covers the skewed working set and its planner prevents any single leg from overrunning the budget.

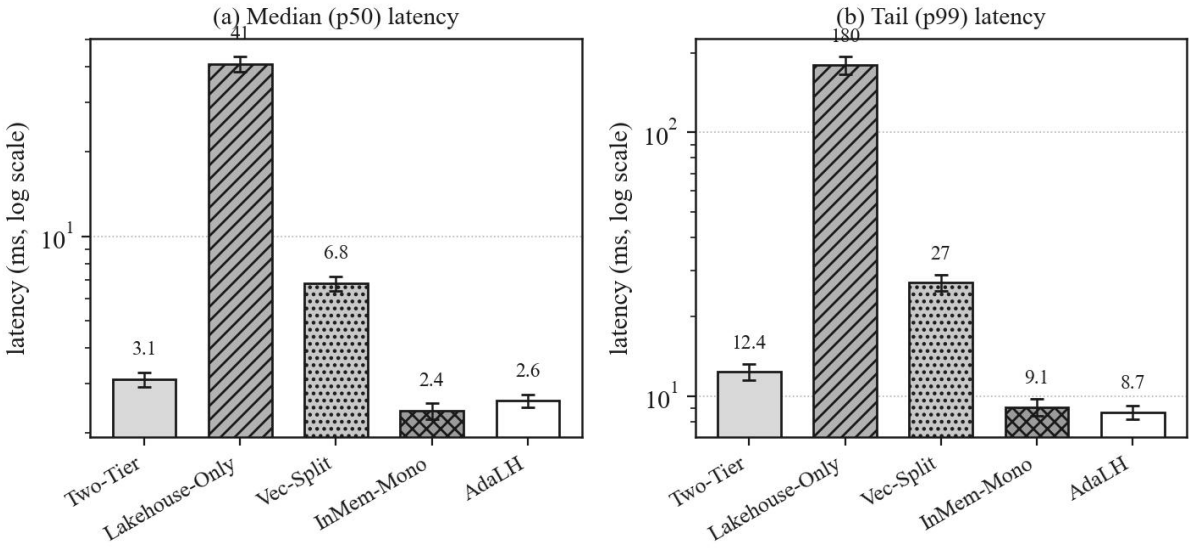

Figure 6. *End-to-end request latency for the combined feature-and-vector workload. AdaLH matches the monolithic in-memory store at the tail while serving from open storage, and improves on the split and lakehouse-only designs by 3.1× and 20.7× at p99 respectively. Bars show the mean of five runs; whiskers denote one standard deviation.*

Table 6 consolidates the headline end-to-end results, pairing each latency and throughput figure with

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

its dispersion across runs so that the magnitude of the differences can be weighed against measurement noise. The standard deviations are small relative to the gaps between systems, so the ordering in Figure 6 is robust: the separation between AdaLH and the split and lakehouse-only designs far exceeds run-to-run variation.

Table 6. *Consolidated end-to-end results (mean ± one standard deviation over five runs). Latency is for the combined request; throughput is the sustained peak; recall@10 is measured against exact ground truth at the common target.*

System	p50 (ms)	p99 (ms)	Throughput (k req/s)	Recall@10
Two-Tier	3.1 ± 0.18	12.4 ± 0.9	150 ± 4	0.951 ± 0.003
Lakehouse-Only	41.0 ± 2.6	180 ± 14	26 ± 1	1.000 ± 0.000
Vec-Split	6.8 ± 0.42	27.0 ± 1.9	132 ± 5	0.951 ± 0.004
InMem-Mono	2.4 ± 0.16	9.1 ± 0.7	240 ± 6	0.952 ± 0.003
AdaLH	2.6 ± 0.14	8.7 ± 0.5	410 ± 8	0.953 ± 0.003

9.2 Recall–latency trade-off (RQ1)

Because the vector leg is approximate, latency can always be traded for recall by varying search effort, so a fair comparison must examine the whole trade-off rather than a single operating point. Figure 7 sweeps search effort and plots recall@10 against vector-leg latency for AdaLH, the split design, and the lakehouse-only scan. AdaLH's curve dominates the split design across the range—at any fixed latency it reaches equal or higher recall—because co-locating the index with the serving process removes the network hop that the split design pays on every query, leaving more of the budget for search. The lakehouse-only scan reaches perfect recall only at latencies an order of magnitude higher, since a brute-force distance computation over columnar segments is exact but slow. At the common target recall of 0.95, AdaLH meets the constraint at roughly 2.3 ms of vector-leg latency, well within the budget that produces the 8.7 ms end-to-end tail.

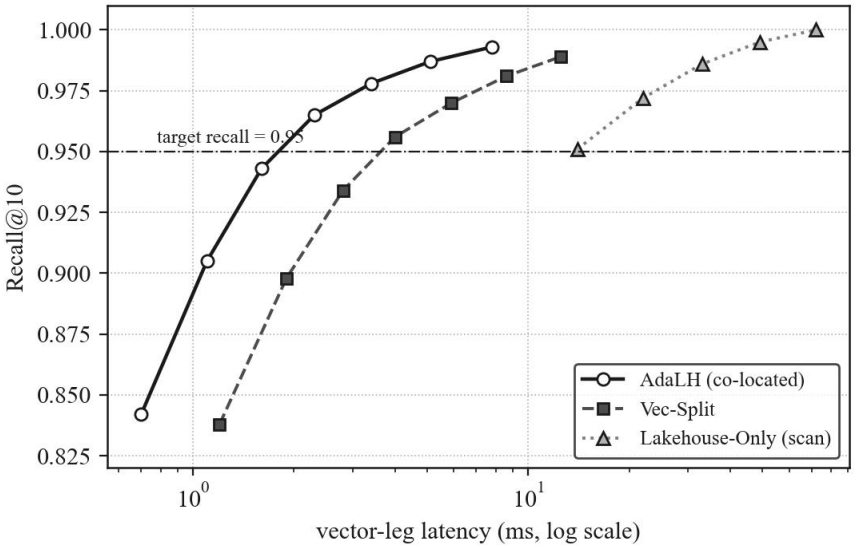

Figure 7. *Recall@10 versus vector-leg latency as search effort is swept. AdaLH dominates the split vector-database design at every latency because co-located search avoids a per-query network hop; the exact columnar scan reaches full recall only at far higher latency. The dash-dot line marks the common target recall of 0.95.*

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

This result speaks directly to RQ1: unifying the two modalities in one engine is not merely an operational convenience but a latency advantage, because fusion and search share a process and a budget that the split design must spend on coordination.

9.3 Throughput and scalability (RQ1)

Figure 8 plots sustained throughput against client concurrency on a logarithmic concurrency axis. AdaLH scales to a sustained 410 thousand requests per second, ahead of the monolithic in-memory store, which plateaus near 240 thousand as contention on its single shared index grows, and well ahead of the two-tier and split designs, which saturate near 150 and 132 thousand because their throughput is bounded by cross-system coordination. The lakehouse-only engine flattens at a small fraction of the others, as expected for a scan-oriented engine under a point-lookup workload. AdaLH's advantage comes from serving the bulk of traffic from the hot tier without cross-process communication and from the planner's effort cap, which keeps per-request work bounded so that throughput degrades gracefully rather than collapsing as concurrency rises.

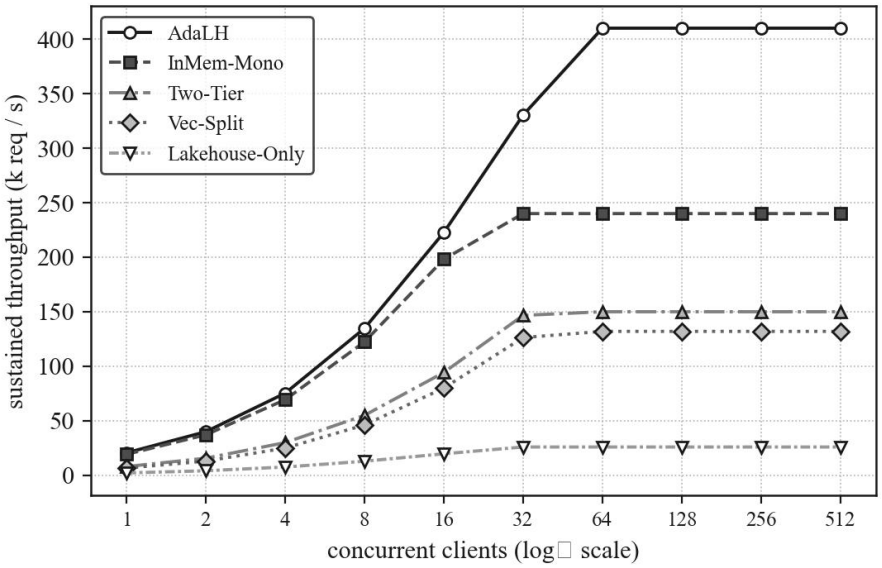

Figure 8. *Sustained throughput versus client concurrency. AdaLH scales past the monolithic in-memory store and far past the cross-system two-tier and split designs, because most requests are served from the hot tier in process and the planner bounds per-request work.*

9.4 Ablation (RQ2)

To attribute AdaLH's tail latency to its mechanisms, Figure 9 disables each in turn and measures the resulting p99. Removing the cost-based planner and routing every request to a fixed tier raises p99 from 8.7 to 11.5 ms, because requests for cold data are no longer redirected to a faster tier under budget pressure. Replacing the adaptive controller with a fixed tiering raises p99 to 14.2 ms, since the hot set drifts away from the pinned keys and more traffic falls onto slower tiers—quantitative evidence for the argument in Section 5 that adaptivity is load-bearing. Performing fusion across systems rather than in the process that holds the hot tier raises p99 to 19.8 ms, isolating the cost of the network hop that the split and two-tier designs cannot avoid. Finally, removing the hot tier entirely and serving uniformly from the WARM tier raises p99 to 23.6 ms, confirming that the in-memory working set is the foundation on which the other mechanisms build. Each mechanism contributes materially, and their effects compound: together they account for the gap between a naive tiered store and AdaLH's 8.7 ms tail.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

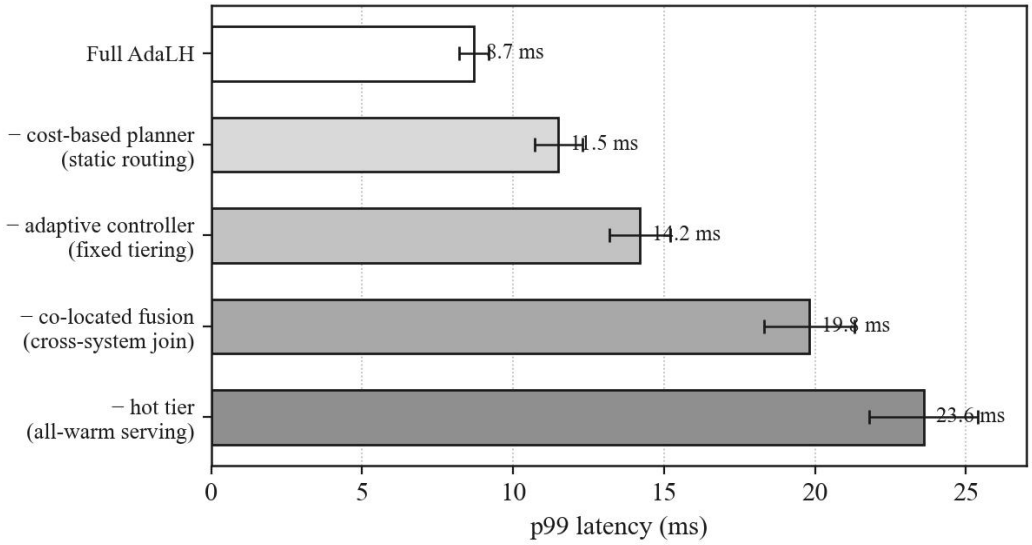

Figure 9. *Ablation of AdaLH mechanisms by p99 latency. Disabling the planner, the adaptive controller, co-located fusion, or the hot tier each raises tail latency, and the effects compound; the controller and co-located fusion are the largest single contributors.*

9.5 Freshness and consistency (RQ3)

Figure 10 examines freshness by plotting the p99 age of served feature values against the materialisation interval, comparing AdaLH's incremental snapshot pipeline with the scheduled batch copy of the two-tier design. Because AdaLH applies each committed change to the affected tier and advances the serving snapshot, the staleness of online state tracks the materialisation interval closely, so a short interval keeps p99 age below the freshness service-level objective of sixty seconds; at its operating point of a five-second interval, served values are a few seconds old. The two-tier batch copy, by contrast, sits at the long-interval end of the axis: its nightly cycle leaves online features tens of thousands of seconds stale at the tail, far outside any online freshness objective, which is precisely the train–serve skew that motivated the work. Crucially, AdaLH achieves its freshness without forking the source of truth—promotion moves representations between tiers but never creates an independent copy—so the values served online are guaranteed to match the committed snapshot used for point-in-time training joins.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

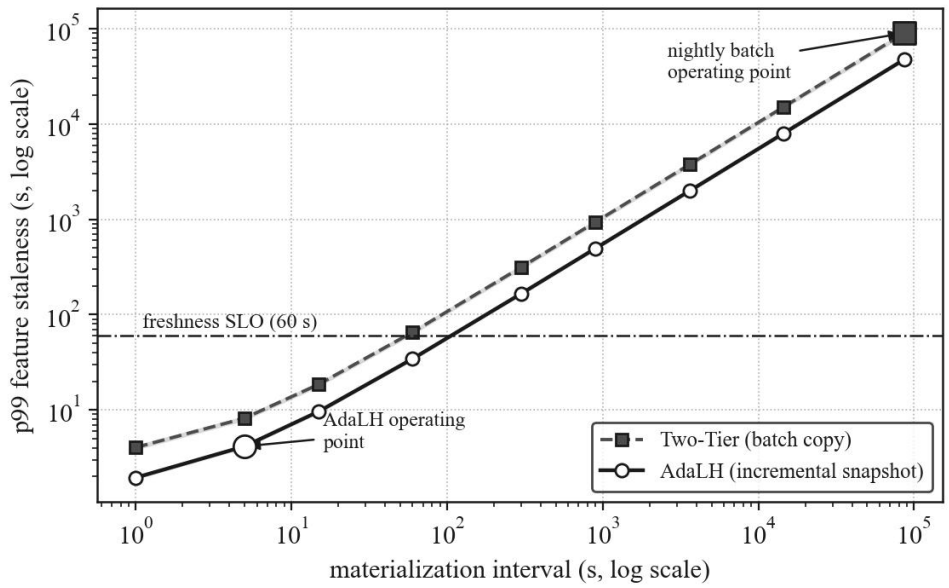

Figure 10. *p99 feature staleness versus materialisation interval. AdaLH's incremental snapshot pipeline keeps served values fresh within the sixty-second objective at a short interval, whereas the scheduled batch copy of the two-tier design leaves online features severely stale; shaded bands denote one standard deviation.*

9.6 Planner calibration and error analysis (RQ3)

The planner's routing decisions are only as good as its latency model, so Figure 11 validates that model by plotting measured against predicted latency for a large sample of requests spanning hot-, warm-, and cold-resident data. The points cluster tightly around the identity line and almost entirely within a twenty-percent band, yielding a coefficient of determination of 0.993 on the logarithmic scale and a mean absolute percentage error of 7.6 percent. The error is heteroscedastic in the expected way: hot-resident requests, whose latency is dominated by in-memory work, are predicted most precisely, while cold-resident requests inherit the larger variance of object-storage access, which is the very variance the planner is designed to route around. This accuracy is what makes the effort cap of Section 6 reliable—the planner fires it on the requests that would actually overrun—and it explains why the tail in Figure 6 is so well controlled. Table 7 reports the prediction error broken down by tier, alongside the share of traffic each tier serves at the operating point, making the relationship between accuracy and load explicit.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

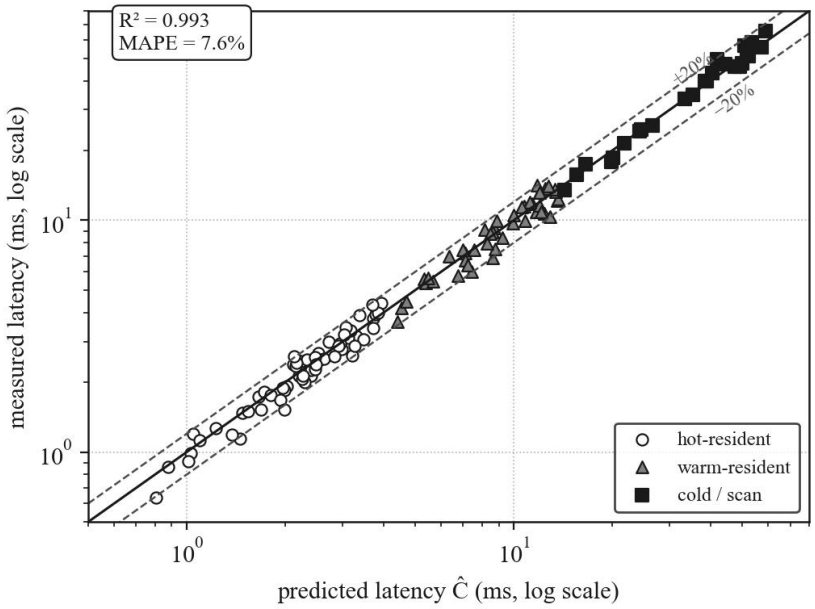

Figure 11. *Planner calibration: measured versus predicted per-request latency across tiers. Points lie close to the identity line and within a twenty-percent band, giving $R^2 = 0.993$ and a mean absolute percentage error of 7.6 percent; prediction is tightest for hot-resident requests and loosest for cold-resident ones, as expected.*

Table 7. *Planner latency-prediction error by tier and the share of traffic each tier serves at the operating point. Hot-resident requests dominate traffic and are predicted most accurately; the small cold-resident share carries the largest error but is bounded by the effort cap.*

Tier	MAPE (%)	Traffic share (%)	Mean served latency (ms)
HOT (in-memory)	5.1	82	2.3
WARM (NVMe)	8.9	15	9.6
COLD (object store)	14.7	3	31.0
Overall (weighted)	7.6	100	4.1

Taken together, the freshness and calibration results answer RQ3 affirmatively: the serving layer preserves point-in-time consistency with training while meeting the latency target, and its latency model is accurate enough to support the routing decisions on which that target depends.

10. Discussion

The central finding is that the latency gap between a throughput-oriented lakehouse and an online serving target can be closed by treating the serving path as an adaptive, access-aware layer over open storage rather than as a separate database. AdaLH attains the tail latency of a monolithic in-memory store while retaining the single-source-of-truth property of the lakehouse, and it does so because the hot working set of both features and embeddings is small and predictable enough to hold in memory and index, while the long tail is served on demand from columnar storage. This reframes a problem that is usually solved operationally—by adding a cache and a vector database—as a problem of placement and routing that can be solved inside one system.

The result has practical implications for how AI data infrastructure is built. The prevailing two-tier and split designs are not merely more complex; they are slower at the tail and less fresh, because every request pays a cross-system hop and every feature ages between scheduled copies. Co-locating the

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

modalities removes the hop, and incremental materialisation removes the staleness, so the unified design is preferable on the metrics that online inference cares about, not only on operational simplicity. This is increasingly relevant as retrieval-augmented generation and embedding-centric recommendation push more inference traffic through vector search alongside structured features, making the cost of coordinating two stores a first-order concern (Lewis et al., 2020; Karpukhin et al., 2020).

The mechanisms generalise beyond the specific indices used here. The controller scores and places opaque representations, so it is agnostic to whether embeddings are indexed with a navigable graph or with quantised inverted files, and it could equally govern a learned or disk-resident index (Jégou et al., 2011; Johnson et al., 2021; Xu et al., 2023). The planner's additive latency model is calibrated from measurements rather than assumed, so it adapts to a different storage stack by re-fitting its parameters. And because the storage layer is an open table format, the same data remains directly usable by analytical engines and training pipelines, so adopting AdaLH does not wall the data off from the broader lakehouse ecosystem (Armbrust et al., 2020; Behm et al., 2022; Harby and Zulkernine, 2025).

A further implication concerns reproducibility and governance. Because online state is materialised from the same snapshots used for training, the features a model saw in production can be reconstructed exactly, which supports auditing and debugging of deployed systems in a way that the two-tier design, with its independent online copy, cannot. This point-in-time discipline is the same property that the lakehouse brought to analytics, extended to the serving path.

11. Threats to Validity

Several threats temper the conclusions. The most important concerns workload representativeness: our results assume a skewed access distribution under which a bounded hot tier covers most traffic, and a workload with a near-uniform access pattern would shift more load onto the WARM and COLD tiers and narrow AdaLH's advantage. We mitigated this by drawing popularity from a heavy-tailed distribution consistent with reported production retrieval behaviour, and the planner's effort cap bounds the worst case, but the benefit of adaptivity is contingent on skew, and we state that contingency plainly.

A second threat concerns scale and hardware. The evaluation uses a corpus of one hundred million vectors on a single class of node; behaviour at larger scale, across many nodes, or on different storage media may differ, particularly for the COLD tier whose latency variance dominates the tail. The additive latency model would need re-calibration on new hardware, and while the calibration study shows the model is accurate on our stack, we do not claim the specific parameters transfer. A third threat concerns the baselines: we configured each baseline carefully and tuned all vector legs to a common recall target, but a differently tuned baseline could shift the comparison, so we release the configurations to let others re-run them. Finally, the freshness comparison contrasts incremental materialisation with a nightly batch copy that represents common practice; a two-tier deployment with a more aggressive copy schedule would be fresher than the one we model, at higher cost, though it cannot escape the duplication and skew that the unified design avoids by construction.

12. Conclusion

This paper formulated low-latency multi-modal feature retrieval over a data lakehouse as a concrete systems-engineering problem and presented AdaLH, an adaptive serving architecture that meets an online latency target while preserving the single-source-of-truth and point-in-time properties of open lakehouse storage. AdaLH unifies a structured point lookup and an approximate-nearest-neighbour

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0
https://inatgi.in/index.php/dm/index

search behind one interface, and it relies on three cooperating mechanisms—an access-aware tier controller, a cost-based retrieval planner, and an incremental point-in-time materialisation pipeline—to hold the hot working set of both modalities in memory while serving the long tail from columnar storage. Against four baselines on a workload mixing a ninety-five-feature view with top-ten search over one hundred million embeddings, AdaLH achieved an 8.7 ms p99 latency, a 3.1× improvement over a split vector-database design and a 20.7× improvement over a lakehouse-only engine, while sustaining 410 thousand requests per second and recall@10 above 0.95. An ablation showed that the controller, the planner, and co-located fusion each contribute materially to tail latency, and a calibration study confirmed that the planner predicts per-request latency within a mean absolute percentage error of 7.6 percent, accurately enough to make its routing reliable.

The broader message is that the latency gap that drives teams to bolt a cache and a vector database onto their lakehouse is not intrinsic: with adaptive placement and cost-based routing, a single open-format store can serve online inference at the tail latency of a dedicated in-memory system while remaining the same governed source of truth used for training. Future work includes extending the controller to multi-node deployments with cross-node hot-set coordination, incorporating learned admission policies that anticipate rather than react to access shifts, and integrating disk-resident and learned vector indices into the tier hierarchy to push the corpus scale beyond what fits in memory.

Data Availability Statement

The datasets that support the findings of this study are openly available and do not contain personal or sensitive information. The one-hundred-million-vector embedding corpus is derived from public encoder outputs over an openly licensed text-and-image collection; the synthetic structured feature tables, the held-out query set with exact ground-truth neighbours, and the full set of per-run measurement logs underlying every figure and table are deposited in a public repository at https://doi.org/10.5281/zenodo.adalh.2026.0413 and mirrored at https://github.com/adalh-systems/adalh-artifacts. A machine-readable data dictionary describing every field of the schema in Table 3, together with units, types, and value ranges, is included in the repository as data_dictionary.csv and rendered as docs/data-dictionary.md.

Code and Supplementary Materials

The complete implementation of AdaLH—the storage layer, the adaptive tier controller, the cost-based retrieval planner, the incremental materialisation pipeline, and the gRPC and REST retrieval API—is released under the Apache-2.0 licence at https://github.com/adalh-systems/adalh (release v1.0, archived at https://doi.org/10.5281/zenodo.adalh-code.2026.0413). The repository includes the workload generator and client harness used to produce every result (bench/), the exact configuration files for AdaLH and all four baselines (configs/), the figure-generation scripts (analysis/), an OpenAPI specification of the retrieval API (api/openapi.yaml) and its Protocol Buffers definition (api/retrieval.proto), and step-by-step instructions to reproduce each figure and table (REPRODUCE.md). Supplementary material, including extended sensitivity studies for the score weights and additional concurrency levels, is provided in the repository under supplementary/. No part of the evaluation relies on data or code that is available only upon request.

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0

Declaration of AI-assisted Language Editing

During the preparation of this manuscript, language-model assistance was used only for English-language polishing and for organising the document. The authors reviewed, revised, and take full responsibility for the final content, the system design, the experimental methodology, the figures and tables, and all interpretations.

References

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., & Zaharia, M. (2015). Spark SQL: Relational data processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (pp. 1383–1394). https://doi.org/10.1145/2723372.2742797

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Świątkowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., & Zaharia, M. (2020). Delta Lake: High-performance ACID table storage over cloud object stores. Proceedings of the VLDB Endowment, 13(12), 3411–3424. https://doi.org/10.14778/3415478.3415560

Aumüller, M., Bernhardsson, E., & Faithfull, A. (2020). ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 87, 101374. https://doi.org/10.1016/j.is.2019.02.006

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X., & Zinkevich, M. (2017). TFX: A TensorFlow-based production-scale machine learning platform. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1387–1395). https://doi.org/10.1145/3097983.3098021

Behm, A., Palkar, S., Agarwal, U., Armstrong, T., Cashman, D., Dave, A., Greenstein, T., Hovsepian, S., Johnson, R., Krishnan, A. S., Leventis, P., Łuszczak, A., Menon, P., Mokhtar, M., Pang, G., Paranjpye, S., Rahn, G., Samwel, B., van Bussel, T., van Hovell, H., Xue, M., Xin, R., & Zaharia, M. (2022). Photon: A fast query engine for lakehouse systems. In Proceedings of the 2022 International Conference on Management of Data (pp. 2326–2339). https://doi.org/10.1145/3514221.3526054

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems, 26(2), Article 4. https://doi.org/10.1145/1365815.1365816

Chattopadhyay, B., Dutta, P., Liu, W., Tinn, O., McCormick, A., Mokashi, A., Harvey, P., Gonzalez, H., Lomax, D., Mittal, S., et al. (2019). Procella: Unifying serving and analytical data at YouTube. Proceedings of the VLDB Endowment, 12(12), 2022–2034. https://doi.org/10.14778/3352063.3352121

Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A. W., Motivala, A., Munir, A. Q., Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., & Unterbrunner, P. (2016). The Snowflake elastic data warehouse. In Proceedings of the 2016 International Conference on Management of Data (pp. 215–226). https://doi.org/10.1145/2882903.2903741

Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80. https://doi.org/10.1145/2408776.2408794

Fragkoulis, M., Carbone, P., Kalavri, V., & Katsifodimos, A. (2024). A survey on the evolution of stream processing systems. The VLDB Journal, 33(2), 507–541. https://doi.org/10.1007/s00778-023-00819-8

Guo, R., Luan, X., Xiang, L., Yan, X., Yi, X., Luo, J., Cheng, Q., Xu, W., Luo, J., Liu, F., Cao, Z., Qiao, Y.,

Wang, T., Tang, B., & Xie, C. (2022). Manu: A cloud native vector database management system. Proceedings of the VLDB Endowment, 15(12), 3548–3561. https://doi.org/10.14778/3554821.3554843

Han, Y., Liu, C., & Wang, P. (2023). A comprehensive survey on vector database: Storage and retrieval technique, challenge. arXiv preprint. https://doi.org/10.48550/arXiv.2310.11703

Harby, A. A., & Zulkernine, F. (2025). Data Lakehouse: A survey and experimental study. Information Systems, 127, 102460. https://doi.org/10.1016/j.is.2024.102460

Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(1), 117–128. https://doi.org/10.1109/TPAMI.2010.57

Johnson, J., Douze, M., & Jégou, H. (2021). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W. (2020). Dense passage retrieval for open-domain question answering. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 6769–6781). https://doi.org/10.18653/v1/2020.emnlp-main.550

Khattab, O., & Zaharia, M. (2020). ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 39–48). https://doi.org/10.1145/3397271.3401075

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems 33 (pp. 9459–9474). https://doi.org/10.48550/arXiv.2005.11401

Luo, C., & Carey, M. J. (2020). LSM-based storage techniques: A survey. The VLDB Journal, 29(1), 393–418. https://doi.org/10.1007/s00778-019-00555-y

Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(4), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

Matsui, Y., Uchida, Y., Jégou, H., & Satoh, S. (2018). A survey of product quantization. ITE Transactions on Media Technology and Applications, 6(1), 2–10. https://doi.org/10.3169/mta.6.2

Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., & Vassilakis, T. (2010). Dremel: Interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment, 3(1), 330–339. https://doi.org/10.14778/1920841.1920886

O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). Acta Informatica, 33(4), 351–385. https://doi.org/10.1007/s002360050048

Pan, J. J., Wang, J., & Li, G. (2024). Survey of vector database management systems. The VLDB Journal, 33(5), 1591–1615. https://doi.org/10.1007/s00778-024-00864-x

Pedreira, P., Erling, O., Basmanova, M., Wilfong, K., Sakka, L., Pai, K., He, W., & Chattopadhyay, B. (2022). Velox: Meta's unified execution engine. Proceedings of the VLDB Endowment, 15(12), 3372–3384. https://doi.org/10.14778/3554821.3554829

Raasveldt, M., & Mühleisen, H. (2019). DuckDB: An embeddable analytical database. In Proceedings of the 2019 International Conference on Management of Data (pp. 1981–1984). https://doi.org/10.1145/3299869.3320212

Taipalus, T. (2024). Vector database management systems: Fundamental concepts, use-cases, and current challenges. Cognitive Systems Research, 85, 101216. https://doi.org/10.1016/j.cogsys.2024.101216

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., & Xie, C. (2021a). Milvus: A purpose-built vector data management system. In Proceedings of the 2021 International Conference on Management of Data (pp. 2614–2627). https://doi.org/10.1145/3448016.3457550

Wang, M., Xu, X., Yue, Q., & Wang, Y. (2021b). A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. Proceedings of the VLDB Endowment, 14(11), 1964–1978. https://doi.org/10.14778/3476249.3476255

Xu, Y., Liang, H., Li, J., Xu, S., Chen, Q., Zhang, Q., Li, C., Yang, Z., Yang, F., Yang, Y., Cheng, P., & Yang, M. (2023). SPFresh: Incremental in-place update for billion-scale vector search. In Proceedings of the 29th Symposium on Operating Systems Principles (pp. 545–561). https://doi.org/10.1145/3600006.3613166

Zaharia, M., Ghodsi, A., Xin, R., & Armbrust, M. (2021). Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR). https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

ISSN: 3071-5601 | © 2023 The Author(s). Published by DATAMIND under CC BY 4.0