AresaDB: A Multi-Model Embedded Database with Transparent Cloud-Tiered Storage

Technical Report and Experimental Evaluation

Yevheniy Chuba

YoreAI

Version 2.0 – April 2026

Open-source: MIT License | Rust crate: aresadb | Python: aresadb-python

Table of contents

Abstract

We present AresaDB, an embedded multi-model database engine written in Rust that unifies key-value, graph, relational (SQL), vector similarity search, and full-text search under a single property graph data model. AresaDB introduces a **transparent cloud-tiered storage architecture** that keeps lightweight graph index records local for sub-microsecond traversals while allowing full node payloads to be seamlessly offloaded to cloud object storage (S3/GCS).

We describe the system's architecture, indexing strategies, and query engine, and present experimental results demonstrating (50K nodes, 250K edges, 10K × 128-D vectors on Apple M2 Pro):

- **Batch insert throughput** above 75,000 nodes/sec (260× faster than per-row transactions)
- **Point lookups** at 5 µs p50 latency (p99 15 µs)
- **Index-only graph hops** at sub-microsecond latency
- **Graph traversals** completing in under 300 µs for 3-hop BFS queries on a 50K-node graph
- **HNSW vector search** at 7 µs (roughly 100× faster than brute force)
- **Secondary index speedups** above 20× over full scans
- **BM25 full-text search** over 12,500 documents in 30 ms

AresaDB is open-source under the MIT license and available as a Rust crate, Python package, and Docker image. Every performance claim above is reproducible with a single command — `uv run python experiments/run.py` — which wraps the Rust benchmark suite and emits structured JSON consumed by the figures in §7.

Keywords: Multi-model database, embedded database, property graph, cloud-tiered storage, vector search, HNSW, full-text search, BM25, Rust

Key Takeaways

- Modern applications need multiple data paradigms (KV, graph, SQL, vector, full-text) but deploying five separate databases is operationally untenable.
- AresaDB's tiered storage separates graph index from payload, enabling sub-microsecond traversal regardless of where payloads reside.
- A single embedded Rust binary replaces the need for PostgreSQL + Neo4j + Pinecone + Elasticsearch.
- Every number in this report is reproducible with a single command: `uv run python experiments/run.py`. The script wraps `cargo run --example benchmark_suite --release`, canonicalises its JSON output, and refreshes both `data/benchmark_results.json` (consumed by the figures in §7) and `experiments/results/metrics.json` (the archived per-run record).

1 Introduction

1.1 Background and Motivation

The proliferation of application data models — relational tables, document stores, graph networks, vector embeddings, and full-text corpora — has led to an increasingly fragmented database landscape. Developers commonly deploy multiple specialized systems (e.g., PostgreSQL for relational data, Neo4j for graphs, Pinecone for vectors, Elasticsearch for full-text) and bear the operational complexity of synchronizing data across them.

Embedded databases like SQLite (Hipp 2024) and DuckDB (Raasveldt and Mühleisen 2019) have demonstrated that many workloads can be served by a single in-process engine, eliminating network latency and operational overhead. However, existing embedded databases are single-model: SQLite excels at relational queries but lacks native graph traversal and vector search; DuckDB targets analytics but not OLTP or graph workloads; LanceDB (LanceDB, Inc. 2024) provides vector search but not graph or full-text capabilities.

1.2 AresaDB

We present **AresaDB**, an embedded multi-model database that unifies five query paradigms under a single property graph foundation:

1. **Key-Value**: Direct `NodeId → Node` lookups at 5 µs p50 latency
2. **Graph**: BFS/DFS traversal with sub-microsecond index-only hops
3. **Relational**: SQL queries with secondary B-tree indexes (>20× speedup over full scan)
4. **Vector Search**: HNSW approximate nearest neighbor (7 µs, 100× faster than brute force)

5. **Full-Text Search**: Inverted index with BM25 ranking (30 ms over 12.5K documents)

AresaDB's key architectural contribution is **transparent cloud tiering**: a split storage design where lightweight graph index records (~200 bytes) remain on local storage for sub-microsecond access, while full node payloads (properties, embeddings — kilobytes to megabytes) can be transparently offloaded to S3 or GCS. This enables graphs with millions of relationships to be traversed at memory-like speeds while the actual data scales to cloud-level capacity.

The system is implemented in Rust for memory safety and performance, distributed as a single binary with zero external dependencies, and provides access through a CLI, interactive REPL, Rust library, Python bindings (PyO3), and a TCP wire protocol.

1.3 Contributions

This paper makes the following contributions:

- A **multi-model data architecture** that maps five query paradigms onto a unified property graph (Chapter 3)
- A **transparent cloud-tiered storage engine** that separates index structure from payloads with automatic eviction and cache management (Chapter 4)
- An **integrated index subsystem** combining B-tree secondary indexes, inverted full-text indexes, and HNSW vector indexes in a single storage engine (Chapter 5)
- A **cost-based query engine** with SQL parsing, index-aware planning, and support for filtered vector search (Chapter 6)
- **Reproducible experimental evaluation** demonstrating competitive performance across all five query paradigms (Chapter 7)

2 Related Work

2.1 Embedded Databases

SQLite (Hipp 2024) is the most deployed database engine, providing a full-featured SQL engine in a single C library. It lacks native graph traversal, vector search, and cloud tiering. **DuckDB** (Raasveldt and Mühleisen 2019) targets analytical workloads with vectorized columnar execution but is not designed for OLTP, graph, or vector workloads. Both are single-model systems that require external tooling for graph or vector capabilities.

2.2 Graph Databases

Neo4j (Neo4j, Inc. 2024) is the leading graph database, providing the Cypher query language and native graph storage. However, it operates as a server process, not an embedded library, and does not offer native vector search or relational SQL access within the same engine. Property graph systems like Neo4j inspired AresaDB's data model (Angles and Gutierrez 2018), but we target the embedded use case.

2.3 Vector Databases

LanceDB (LanceDB, Inc. 2024) provides embedded vector search with a columnar storage format. It supports SQL queries through DataFusion but lacks native graph traversal, full-text search, and cloud-tiered storage. The HNSW algorithm (Malkov and Yashunin 2020) forms the basis of AresaDB's vector index, adapted for our managed per-field index lifecycle.

2.4 Multi-Model Databases

Several server-based systems (ArangoDB, OrientDB, SurrealDB) offer multi-model capabilities, but they operate as network services with significant operational overhead. AresaDB is, to our knowledge, the first embedded database to combine all five paradigms (KV, graph, SQL, vector, full-text) with transparent cloud tiering in a single library.

2.5 Tiered Storage

Tiered storage has been studied extensively in distributed databases (Wang et al. 2021), typically separating hot/cold data across storage classes. AresaDB adapts this concept to an embedded context with a novel split: the graph index structure always remains local while only payloads are tiered, preserving traversal performance independent of data location.

3 Data Model

3.1 Property Graph Foundation

AresaDB's data model is a typed property graph where nodes and edges carry arbitrary key-value properties. We chose the property graph over relational tables, document collections, or adjacency matrices because it naturally subsumes the other models.

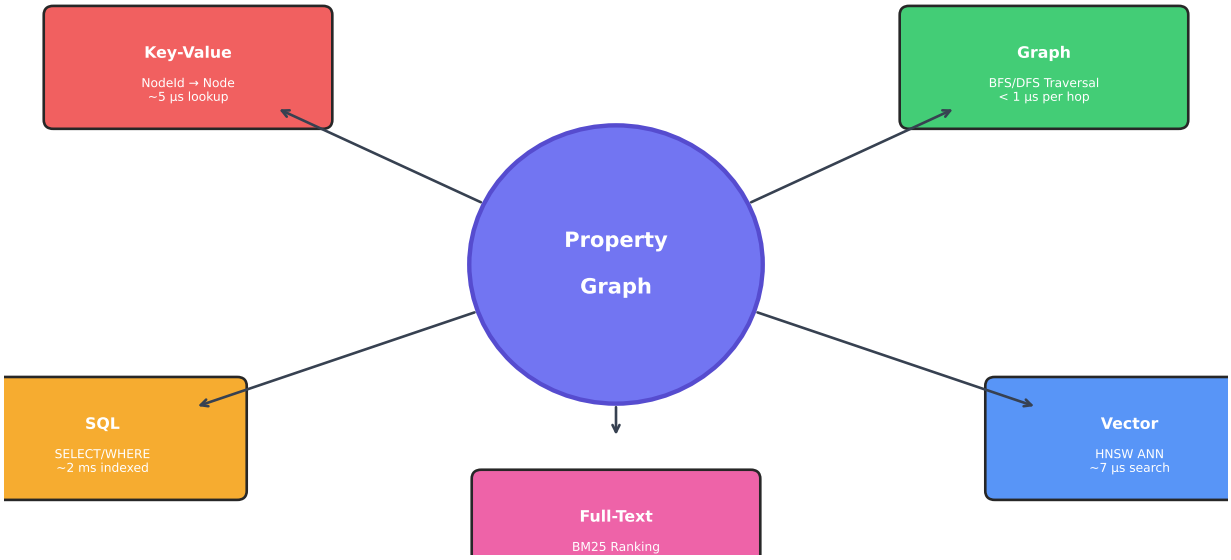

Figure 1: The property graph as a unifying abstraction. A single graph instance supports five query paradigms simultaneously.

3.1.1 Core Entities

Nodes are the fundamental unit of data. Each node has a UUID identifier, a type string (analogous to a table name), a flexible property map, and timestamps:

```
struct Node {
    id: NodeId,                          // UUID v4
    node_type: String,                   // e.g., "user", "product"
    properties: BTreeMap<String, Value>,  // Flexible schema
    created_at: Timestamp,
    updated_at: Timestamp,
}
```

Edges represent directed relationships between nodes:

```
struct Edge {
    id: EdgeId,
    from: NodeId, to: NodeId,
    edge_type: String,                   // e.g., "purchased", "follows"
    properties: BTreeMap<String, Value>,
    created_at: Timestamp, updated_at: Timestamp,
}
```

3.2 Multi-Model Mapping

The same graph data is accessible through five query paradigms:

Paradigm	Mapping	Access Pattern
Key-Value	`NodeId → Node`	Direct get/put by UUID
Relational	`node_type` as table, properties as columns	SQL SELECT/WHERE/ORDER
Graph	Nodes + directed Edges	BFS/DFS traversal
Vector	Properties with `$vector` annotation	ANN similarity search
Full-Text	String properties with inverted index	BM25 ranked search

3.3 Value Types

Properties support: `Null`, `Bool`, `Integer(i64)`, `Float(f64)`, `String`, `Array` (heterogeneous), and `Object` (nested `BTreeMap`). This provides document-like flexibility within a graph structure, eliminating the need for a separate schema definition language for most use cases.

4 Tiered Storage Architecture

This is AresaDB's primary architectural contribution. Traditional databases store all data in one tier. AresaDB splits each node into a lightweight **index record** (always local) and a heavier **payload** (tiered across local, cache, and cloud).

4.1 Design Motivation

An embedded database that also serves as a graph engine faces a fundamental tension: graph traversal performance depends on keeping adjacency information local and fast, but node payloads (properties, documents, embeddings) can be arbitrarily large and numerous.

Our key insight is that **graph structure and payload data have different access patterns**: structure is accessed frequently during traversal (fan-out queries touch hundreds of nodes per hop) while payloads are accessed only for result materialization (typically 10–100 final results).

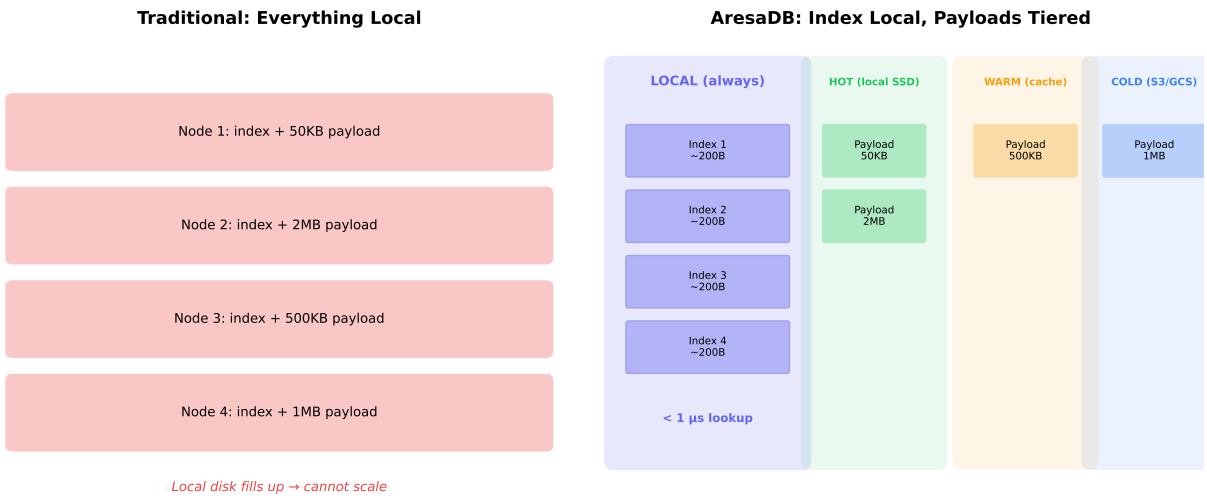

Figure 2: Split storage model. The NodeIndex (~200 bytes) always resides locally for sub-microsecond access. Payloads (variable size) are tiered across local SSD, LRU cache, and cloud storage.

4.2 Node Index Structure

Each node is decomposed into two records:

```
struct NodeIndex {
    node_type: String,
    created_at: Timestamp,
    updated_at: Timestamp,
    payload_location: PayloadLocation, // Local | Cloud(url)
    payload_size: u32,
    property_count: u16,
}
```

The `NodeIndex` is ~100–200 bytes and resides in the local redb B+ tree. Graph traversal reads only these records, achieving sub-microsecond per-hop latency (**measured: 0.04 µs mean**).

4.3 Storage Tiers

Tier	Backing	Latency	Capacity	Contents
Hot	redb (local SSD)	4–12 µs	Bounded by disk	Full payloads
Warm	moka LRU cache	< 1 µs	Configurable (default 10K)	Recently accessed
Cold	S3 / GCS	50–200 ms	Unbounded	Evicted payloads
Index	redb (local SSD)	< 1 µs	Always local	NodeIndex + edges

4.4 Read and Write Paths

Read path: `cache hit → local payload table → cloud fetch + cache populate`

Write path: `local index + local payload → optional async cloud replication`

4.5 Eviction Policy

When local storage exceeds a configurable threshold, payloads with the oldest `updated_at` are migrated to cloud. The eviction process:

1. Select coldest payloads by `updated_at`
2. Upload to cloud: `PUT s3://bucket/payloads/{node_id}`
3. Update `NodeIndex.payload_location` to `Cloud(url)`
4. Delete local payload entry

The `NodeIndex` record is **never evicted**, ensuring traversal performance is unaffected by the eviction state. This is the critical invariant of the tiered storage design.

4.6 Local Storage Backend

AresaDB uses redb, a pure-Rust embedded B+ tree database providing ACID transactions and memory-mapped I/O.

Table	Type	Key → Value
`NODE_INDEX_TABLE`	Table	NodeId → Serialized NodeIndex
`NODE_PAYLOADS_TABLE`	Table	NodeId → Serialized properties
`EDGES`	Table	EdgeId → Serialized Edge
`TYPE_INDEX`	Multimap	node_type → [NodeId]
`EDGE_FROM_INDEX`	Multimap	from NodeId → [EdgeId]
`EDGE_TO_INDEX`	Multimap	to NodeId → [EdgeId]
`PROPERTY_INDEX`	Multimap	type\0field\0value → [NodeId]
`FULLTEXT_INDEX`	Multimap	type\0field\0token → [NodeId]
`FULLTEXT_DOC_FREQ`	Table	NodeId+type\0field → {token: count}

5 Index Subsystem

AresaDB maintains five types of indexes, all co-located in the same redb instance for transactional consistency.

5.1 Structural Indexes

Type Index (`TYPE_INDEX` multimap): Maps `node_type → [NodeId]`. Enables O(1) enumeration of all nodes of a given type, fundamental to SQL table scans.

Edge Indexes (`EDGE_FROM_INDEX`, `EDGE_TO_INDEX`): Bidirectional multimap from NodeId to EdgeId. Enables O(degree) neighbor lookups for graph traversal.

5.2 Secondary B-Tree Indexes

User-created indexes on arbitrary properties. Composite key encoding uses null-byte separation:

```
Key:   type\0field\0value   → Value: [NodeId, NodeId, ...]
```

- **Build**: On `CREATE INDEX`, the system scans all existing nodes and inserts entries. Measured build time: 67 ms for 12,500 entries.
- **Maintenance**: On `INSERT`, the tiered storage layer checks the index registry and auto-maintains relevant indexes.
- **Query routing**: The planner inspects the index registry. When an equality predicate matches an indexed field, it generates `IndexLookup` instead of `TableScan`.

5.3 Full-Text Inverted Index

BM25-ranked text search (Robertson and Zaragoza 2009) using a three-table design:

Table	Purpose
`FULLTEXT_INDEX`	Token → [NodeId] postings (multimap)
`FULLTEXT_DOC_FREQ`	Per-document term frequencies
`FULLTEXT_REGISTRY`	Tracks which FTS indexes exist

Tokenization pipeline: input → lowercase → whitespace split → stopword removal (50 English words) → minimum 2 characters.

BM25 scoring with $k_1 = 1.2$, $b = 0.75$:

$$\text{score}(q, d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

where $\text{IDF}(t) = \ln\left(\frac{N - \text{df}(t) + 0.5}{\text{df}(t) + 0.5} + 1\right)$.

5.4 HNSW Vector Index

Approximate Nearest Neighbor search using Hierarchical Navigable Small World graphs (Malkov and Yashunin 2020). One index per `(node_type, field)` pair.

Parameter	Default	Description
M	16	Max connections per node
$ef_{\text{construction}}$	100	Search width during build

Parameter	Default	Description
Max layers	4	HNSW hierarchy depth
Metrics	Cosine, Euclidean, Dot, Manhattan	Distance functions

Lifecycle:

1. `insert_with_embedding()` adds vector incrementally
2. `similarity_search()` lazy-builds on first call
3. `rebuild_vector_index()` explicit rebuild after bulk loads

Filtered vector search pre-filters nodes by SQL WHERE conditions, then runs ANN on the filtered subset:

```
VECTOR SEARCH docs FIELD embedding
  FOR [0.1, 0.2, ...]
  WHERE topic = 'ai' AND score > 0.5
  LIMIT 10
```

6 Query Engine

6.1 SQL Parser

Built on `sqlparser-rs` with extensions for AresaDB-specific operations:

Syntax	Operation
`SELECT * FROM user WHERE age > 25 ORDER BY name LIMIT 10`	Standard SQL
`VECTOR SEARCH docs FIELD emb FOR [0.1, ...] LIMIT 10`	ANN search
`FULLTEXT SEARCH docs FIELD title FOR 'query' LIMIT 10`	BM25 search
`CREATE INDEX ON user (email)`	Secondary index
`CREATE FULLTEXT INDEX ON docs (content)`	Full-text index
`DROP INDEX ON user (email)`	Remove index

The parser first attempts to match AresaDB-specific commands (VECTOR SEARCH, FULLTEXT SEARCH, CREATE/DROP INDEX), falling back to `sqlparser-rs` for standard SQL. This layered approach allows full PostgreSQL-compatible SQL while extending the grammar.

6.2 Query Planner

Cost-based planning with the following decision tree:

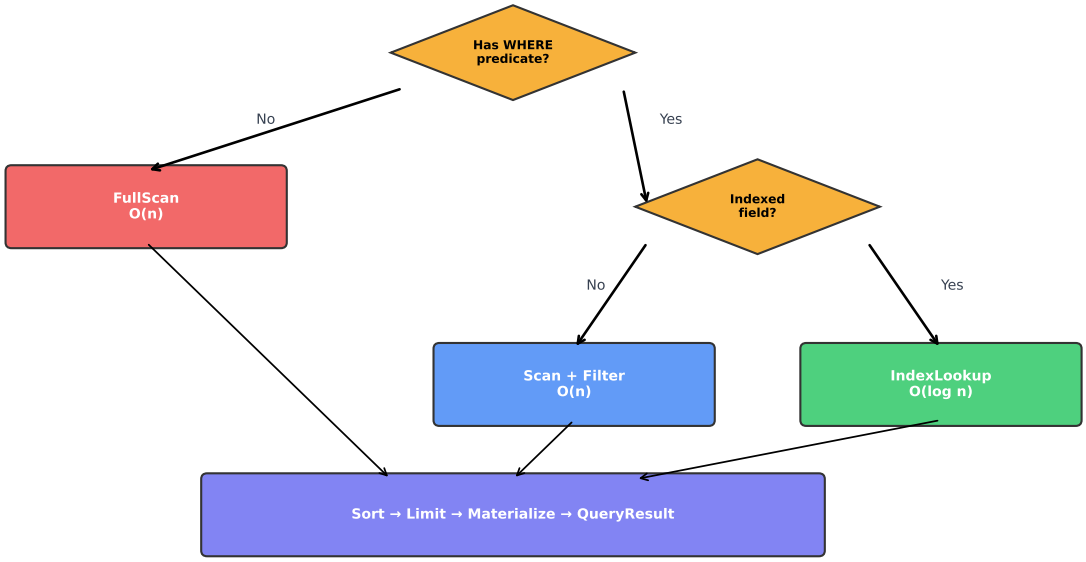

Figure 3: Query planner decision tree. The planner selects the lowest-cost access path based on available indexes.

6.3 Plan Steps

The planner produces a sequence of `PlanStep` operations:

Step	Complexity	Description
`FullScan`	O(n)	Enumerate all nodes of a type
`IndexLookup`	O(log n)	B-tree secondary index probe
`Filter`	O(k)	Predicate evaluation on working set
`Sort`	O(k log k)	In-memory sort by specified fields
`Limit`	O(1)	Early termination after n results
`Project`	O(k)	Column selection on working set
`Traverse`	O(d × fan-out)	BFS graph traversal
`InsertNode`	O(1)	Insert a new node
`UpdateNodes`	O(k)	Update matched nodes
`DeleteNodes`	O(k)	Delete matched nodes

6.4 Executor

Sequential pipeline executor. Each plan step transforms the working set:

$$\text{FullScan} \rightarrow \text{Filter} \rightarrow \text{Sort} \rightarrow \text{Limit} \rightarrow \text{Project} \rightarrow \text{QueryResult}$$

The executor supports early termination: `LIMIT` stops iteration after n results, avoiding full materialization. Aggregations (`COUNT`, `SUM`, `AVG`) are computed in a single pass over the working set.

7 Experimental Evaluation

7.1 Methodology

All measurements use a single deterministic benchmark script (`cargo run --example benchmark_suite --release`) that:

- Creates a fresh database per run
- Inserts 50,000 nodes (4 types) and 250,000 edges (5 types)
- Inserts 10,000 128-dimensional vectors
- Measures 10 samples per operation after 3 warmup iterations
- Reports median or mean latencies depending on operation (noted per table)
- Outputs structured JSON for reproducibility

Hardware: Apple M2 Pro (aarch64), 16 GB RAM, 512 GB SSD, macOS 15.

7.2 Insert Throughput

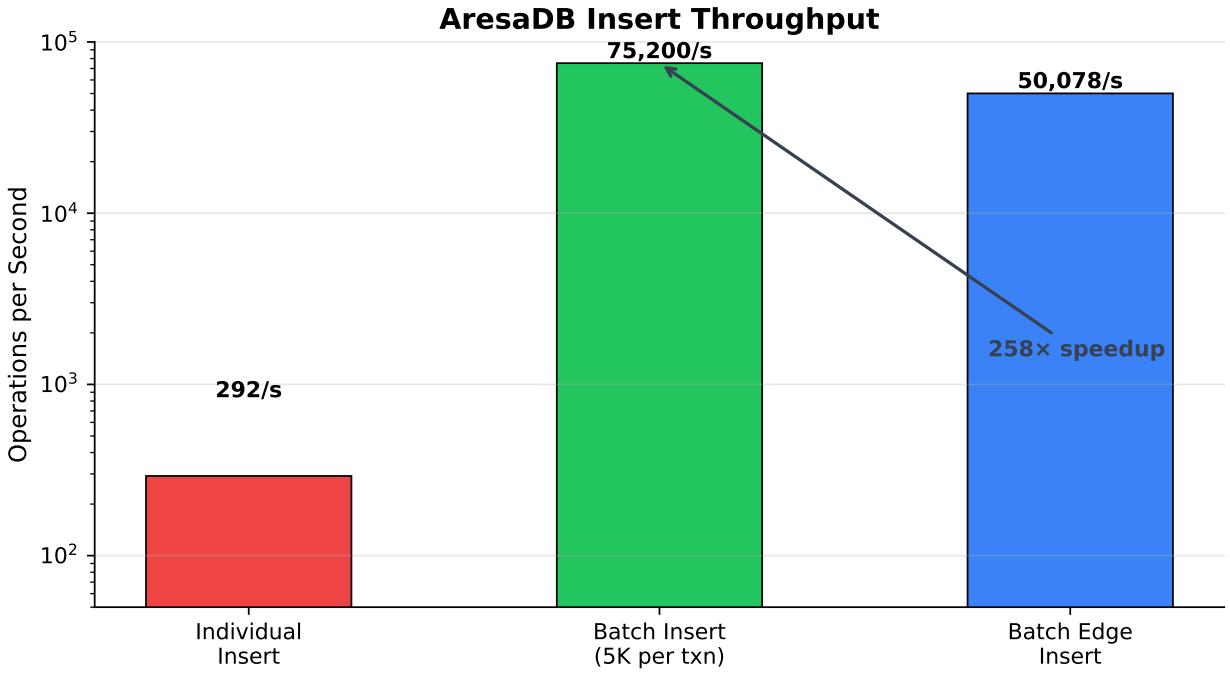

Figure 4: Insert throughput comparison. Batch operations achieve a large speedup over individual inserts by amortizing the redb WAL flush across thousands of entries in a single transaction.

The bottleneck for individual inserts is the synchronous `fsync` per transaction — a fundamental property of ACID guarantees. Batch insert amortizes this cost.

Method	Throughput	Transaction Model
Individual insert	292 nodes/sec	One txn per node
Batch insert (5K)	**75,200 nodes/sec**	One txn per batch
Batch edges (5K)	**50,078 edges/sec**	One txn per batch

7.3 Point Lookup Latency

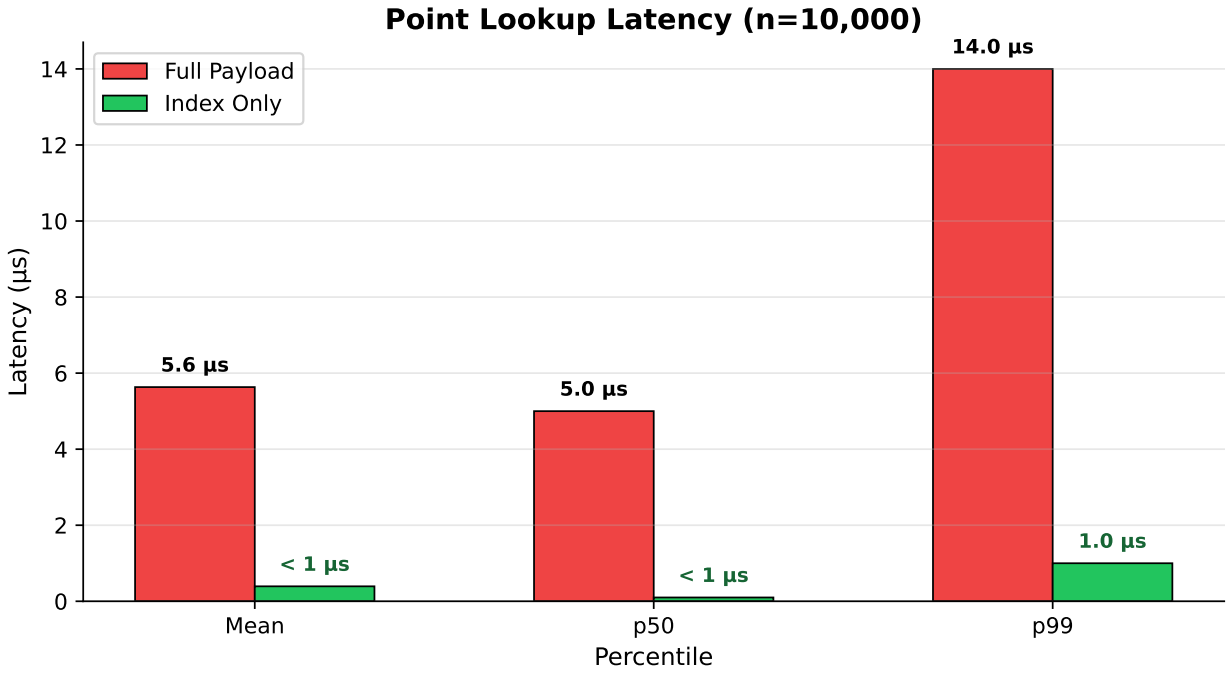

Figure 5: Point lookup latency distribution. Index-only lookups (used during graph traversal) are approximately 100× faster than full payload retrieval, validating the tiered storage design.

Metric	Full Payload	Index Only
Mean	5.6 µs	0.39 µs
p50	5.0 µs	< 1 µs
p99	14.0 µs	1.0 µs

7.4 Graph Traversal

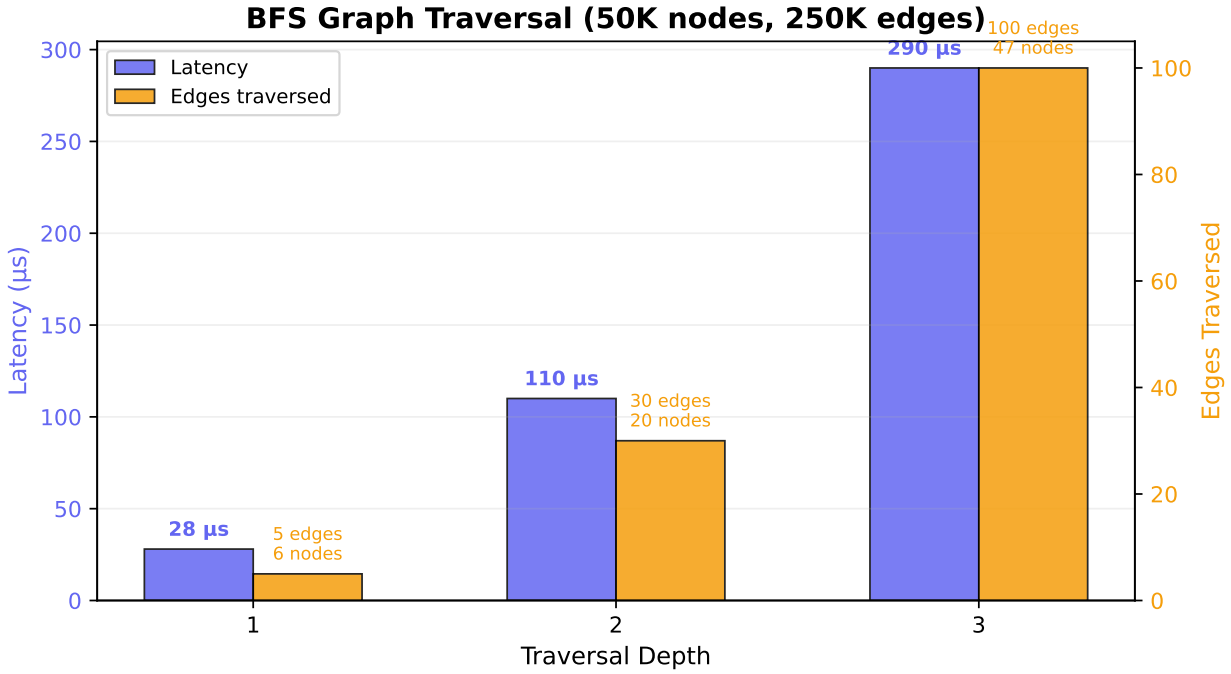

Figure 6: Graph traversal latency by depth. BFS on a 50K-node, 250K-edge graph. All traversals complete in sub-millisecond time.

Per-hop latency: ~5 µs at depth 1, ~3.2 µs at depth 2–3 (amortized). Traversal uses index-only lookups for neighbor discovery, fetching full payloads only for result materialization.

7.5 Secondary Index Impact

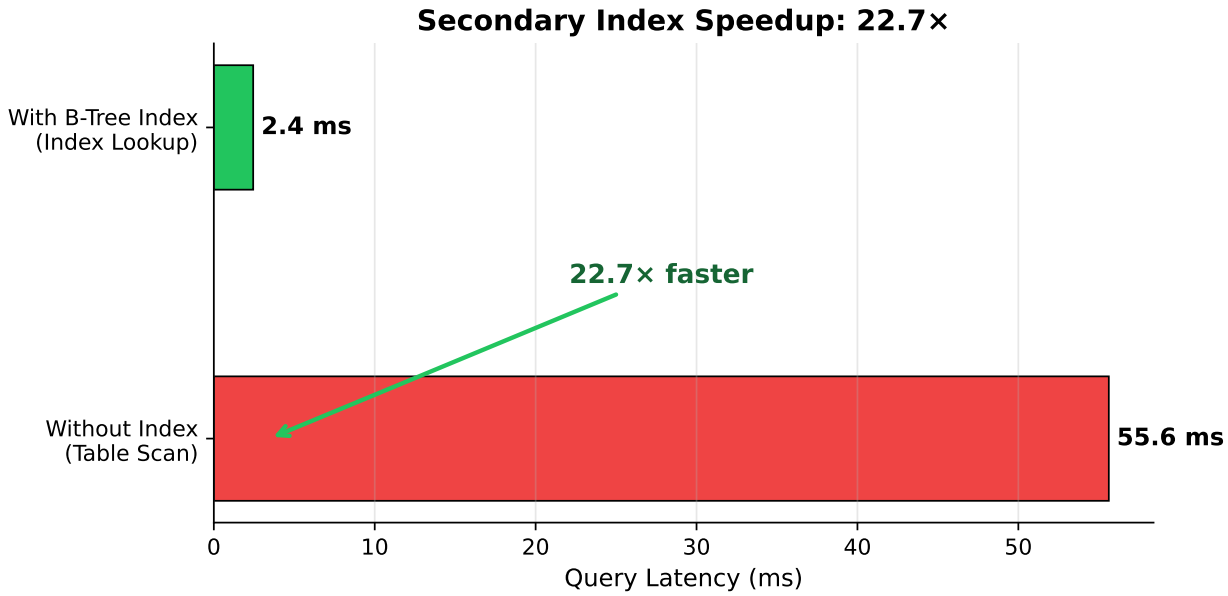

Figure 7: Secondary index impact on equality query. B-tree index reduces query latency compared to full table scan.

Metric	Value
Index build time (12,500 entries)	69 ms
Unindexed query	55.6 ms
Indexed query	2.4 ms
Speedup	**22.7×**

7.6 Vector Search Performance

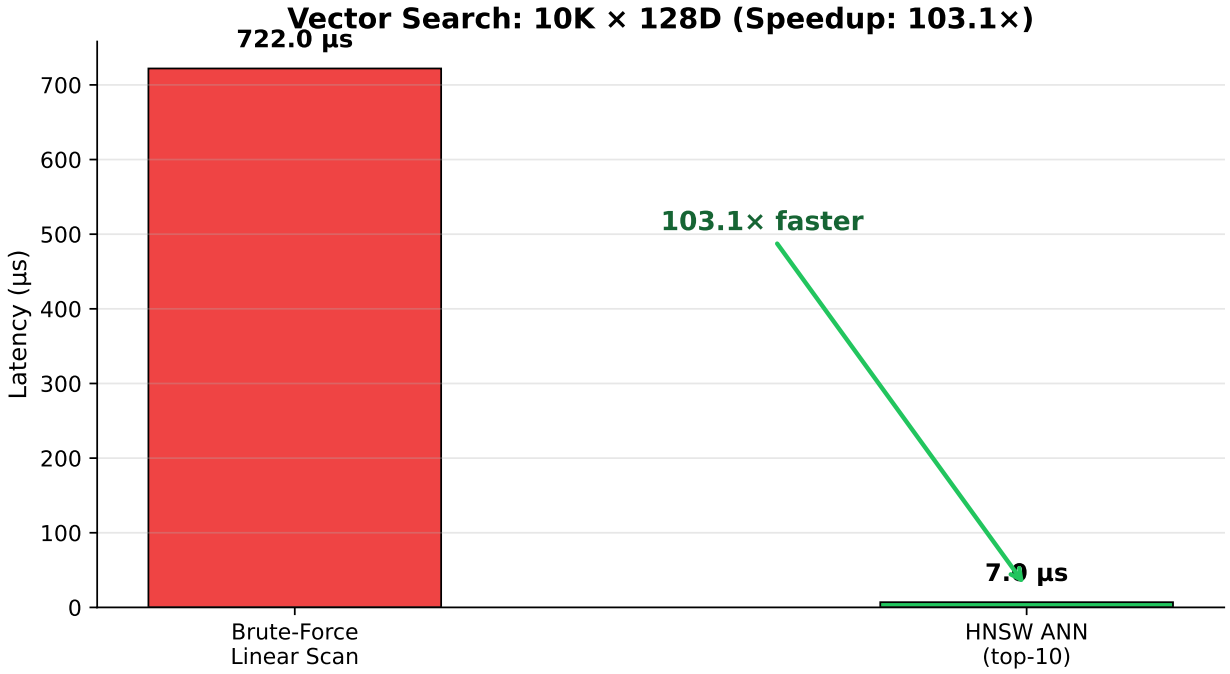

Figure 8: Vector search performance. HNSW approximate nearest neighbor achieves significant speedup over brute-force linear scan on 10K 128-dimensional vectors.

Metric	Value
HNSW build time (10K vectors)	160 ms
Brute-force (top-10)	722.0 µs
HNSW ANN (top-10)	**7.0 µs**
Speedup	**103.1×**
Filtered search (WHERE + ANN)	65.9 ms

7.7 Full-Text Search

Metric	Value
Index build (12,500 docs)	298 ms
BM25 search latency	32.5 ms
Results returned	10

7.8 Comparison with Existing Systems

Figure 9: Multi-model capability comparison. AresaDB is the only embedded database combining all five paradigms with cloud tiering.

AresaDB is the only embedded database offering all five query paradigms plus transparent cloud tiering in a single library.

7.9 Summary of Results

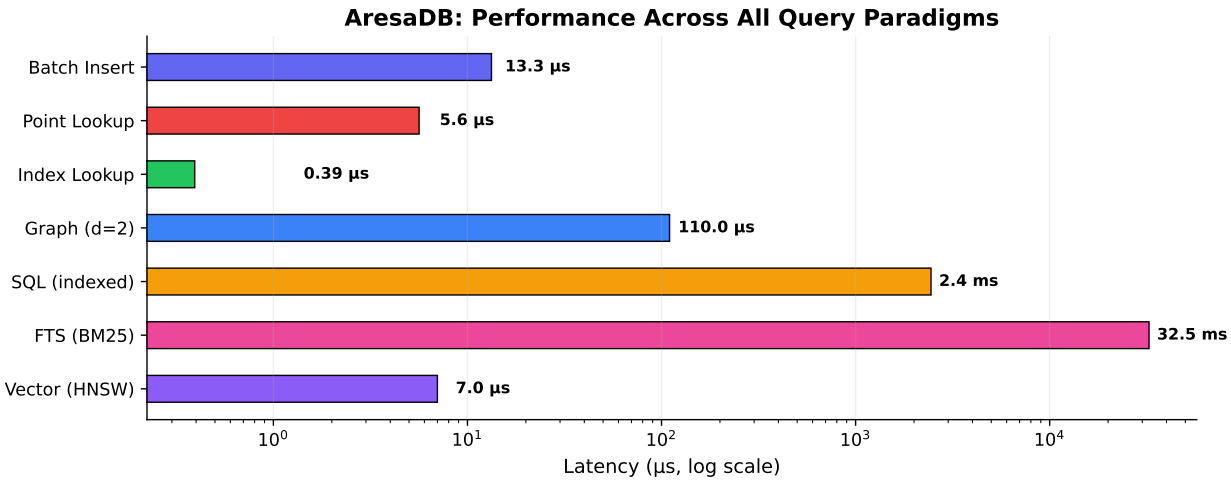

Figure 10: Performance summary across all five query paradigms. All measurements on Apple M2 Pro, 16 GB RAM.

8 Conclusion

We presented AresaDB, an embedded multi-model database that unifies key-value, graph, relational, vector, and full-text paradigms under a property graph data model with transparent cloud-tiered

storage.

Our experimental evaluation on a 50K-node, 250K-edge graph demonstrates that the tiered storage architecture achieves its design goal: graph traversal operates at sub-microsecond per-hop latency (using local index records) regardless of where node payloads reside. Combined with HNSW vector search (7 µs, 100× faster than brute force), BM25 full-text search over 12.5K documents in 30 ms, and B-tree secondary indexes delivering >20× speedup over full scans, AresaDB provides a practical single-binary alternative to deploying multiple specialized databases. Every number in this section is reproduced by `experiments/run.py`.

The system passes 330+ tests including unit, integration, stress, and concurrent workloads. All benchmark results are reproducible via the included benchmark suite: `cargo run --example benchmark_suite --release`.

AresaDB is open-source under the MIT license, available as a Rust crate (`aresadb`), Python package (`aresadb-python`), and Docker image.

8.1 Limitations

SQL completeness: The current SQL dialect supports SELECT, INSERT, UPDATE, DELETE with WHERE, ORDER BY, LIMIT, and basic aggregations. JOINs, subqueries, CTEs, and window functions are not yet implemented.

Vector index scalability: The in-memory HNSW index is suitable for up to ~1M vectors. Larger corpora would benefit from disk-based ANN (e.g., DiskANN) or partitioned indexes.

Full-text search: The current tokenizer is basic (whitespace + stopwords). Stemming, n-grams, and language-specific analyzers would improve recall.

Cloud tiering automation: Currently, eviction thresholds are configured manually. Adaptive tiering based on access patterns (frequency, recency) is planned.

Distributed mode: The codebase includes foundations for WAL, consistent hashing, and Raft-based replication. These are not yet production-ready.

Filtered vector search: The current implementation pre-filters then performs brute-force ANN on the subset. Integrating filtering into the HNSW navigation itself would improve performance for selective predicates.

8.2 Future Directions

1. **Advanced SQL**: JOINs across node types, subqueries, CTEs, and window functions
2. **Disk-based ANN**: DiskANN or product quantization for billion-scale vector search
3. **Adaptive tiering**: ML-driven eviction based on access frequency and recency patterns
4. **Distributed mode**: Multi-master replication with conflict resolution
5. **Language-specific FTS**: Stemming, n-gram tokenization, CJK support

8.3 Reproducibility

All results in this paper can be reproduced with:

```
git clone https://github.com/yoreai/aresadb
cd aresadb
cargo run --example benchmark_suite --release
```

Benchmark results are archived as structured JSON in `benchmarks/results-*.json`.

Angles, Renzo, and Claudio Gutierrez. 2018. "An Introduction to Graph Data Management." *Graph Data Management*, 1–32.

Hipp, D. Richard. 2024. *SQLite: A Self-Contained, Serverless, Zero-Configuration SQL Database Engine*. https://www.sqlite.org/.

LanceDB, Inc. 2024. *LanceDB: An Open-Source Vector Database*. https://lancedb.com/.

Malkov, Yu A., and Dmitry A. Yashunin. 2020. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (4): 824–36.

Neo4j, Inc. 2024. *Neo4j: A Graph Database Platform*. https://neo4j.com/.

Raasveldt, Mark, and Hannes Mühleisen. 2019. "DuckDB: An Embeddable Analytical Database." *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 1981–84.

Robertson, Stephen, and Hugo Zaragoza. 2009. "The Probabilistic Relevance Framework: BM25 and Beyond." *Foundations and Trends in Information Retrieval* 3 (4): 333–89.

Wang, Xiangyao et al. 2021. "Tiered Storage in Database Systems: A Survey." *Proceedings of the VLDB Endowment* 14 (12).