arXiv:2603.08036v4 [cs.DB] 13 Nov 2026

Samyama: A Unified Graph-Vector Database with In-Database Optimization, Agentic Enrichment, and Hardware Acceleration

Madhulatha Mandarapu* Sandeep Kunkunuru†

VaidhyaMegha Private Limited, India
https://samyama.ai/

November 2026 | Samyama v1.8.0 | arXiv v4

Abstract

Modern data architectures fragment across graph databases, vector stores, analytics engines and optimization solvers, forcing ETL between them. We present Samyama, a graph-vector database in Rust that unifies these workloads in one engine: a RocksDB-backed store with MVCC, a vectorized executor, a cost-based planner, a CSR analytics engine, RDF and SPARQL, 22 metaheuristic solvers callable from the query language, HNSW vector indexing, and agentic enrichment that expands a graph via LLMs. It has been run to billion-edge scale: 74.3M nodes and 1.07B edges from four biomedical sources on one machine for $2.50 of spot compute.

This version re-measures the system at release v1.8.0 on **rented Linux hosts a reader can boot**, replacing earlier figures taken on a Mac Mini that nobody could re-run. On a 16-vCPU cloud instance, ingestion reaches 123K–152K nodes/s and PageRank costs 6.7 ms per iteration at ten thousand nodes and 236 ms at a million. **The previously reported 8.2× GPU speedup is withdrawn**; in its place we measure the CUDA path, which no earlier version measured at all, on a rented NVIDIA A16: parity with the CPU at ten thousand nodes, 1.62× at a hundred thousand and 2.70× at a million, with 1.1–1.35 s of context initialisation on the first call. The earlier Cypher-throughput figure, its claim of near-constant index-driven scaling, and a vector-search figure are withdrawn as unreproducible. openCypher conformance, once estimated at ∼90%, is now measured at 99.9% of evaluated TCK scenarios. Against Neo4j 5 and FalkorDB on one host over an identical SF10 extract, 14 of 14 SNB complex reads land within 5× of the best competitor and 7 of 14 are faster.

Keywords: Graph Databases, Vector Search, Distributed Systems, Metaheuristic Optimization, Rust, GPU Acceleration, Agentic AI, RDF, LDBC.

1 Introduction

The rise of Large Language Models (LLMs) has popularized Retrieval-Augmented Generation (RAG), creating demand for systems that handle both relational structure (graphs) and semantic similarity (vectors). Simultaneously, industrial applications increasingly require in-database optimization for resource allocation, scheduling, and supply chain management. Existing solutions force developers

*madhulatha@samyama.ai, ORCID: https://orcid.org/0009-0005-2837-6725
†sandeep@samyama.ai, ORCID: https://orcid.org/0000-0002-8886-1846

to compose architectures from disparate systems—Neo4j for graphs, Pinecone for vectors, Spark for analytics, and Python/Gurobi for optimization—incurring data movement overhead and operational complexity.

Samyama (Sanskrit for "Integration") is designed as an AI-native database that treats graphs, vectors, and optimization as first-class citizens within a single memory-safe engine. We organize our contributions into two categories:

Primary research contributions:

1. **Late materialization for graphs**: `NodeRef`-based lazy property resolution [Abadi et al., 2008] achieving 4.0–4.7× traversal speedup (§2.3, §8.3).
2. **In-database metaheuristic optimization**: 22 solvers accessible directly via Cypher procedures, eliminating ETL overhead (§4).
3. **Agentic Enrichment (GAK)**: Autonomous graph expansion using LLM tool-calling (§5.2).
4. **Billion-edge knowledge graph federation** (v0.7.0): 74.3M nodes and 1.07B edges from 4 biomedical sources loaded on commodity hardware for $2.50; 6-KG federation spanning molecular biology to population health with 136/140 benchmark queries passing (§8.8, §8.9).

Engineering contributions:

5. **Unified engine**: Property graph + vector search + analytics + optimization in one binary.
6. **Cost-based query planner**: Plan enumeration, predicate pushdown, ExpandInto, and join reordering (§2.3). The plan cache described in the first version of this paper was deleted in v1.8.0 and replaced by a result cache.
7. **Worst-case optimal joins**: LeapFrog TrieJoin [Veldhuizen, 2014] for cyclic patterns (v0.7.0).
8. **Cross-platform GPU acceleration**: wgpu-based compute shaders for graph algorithms and PCA.
9. **SDK ecosystem**: Rust, Python (PyO3), TypeScript SDKs with embedded and remote access.
10. **Industry validation**: 100% LDBC Graphalytics [Iosup et al., 2016] (28/28), 100% FinBench (40/40).

This paper evaluates the four primary contributions: late materialization (§8.3), in-database optimization (§4.1), agentic enrichment (§5.4), and billion-edge federation (§8.8–§8.9).

2 System Architecture

Samyama is built on a modern Rust stack for memory safety and zero-cost abstractions. Figure 1 illustrates the overall system architecture.

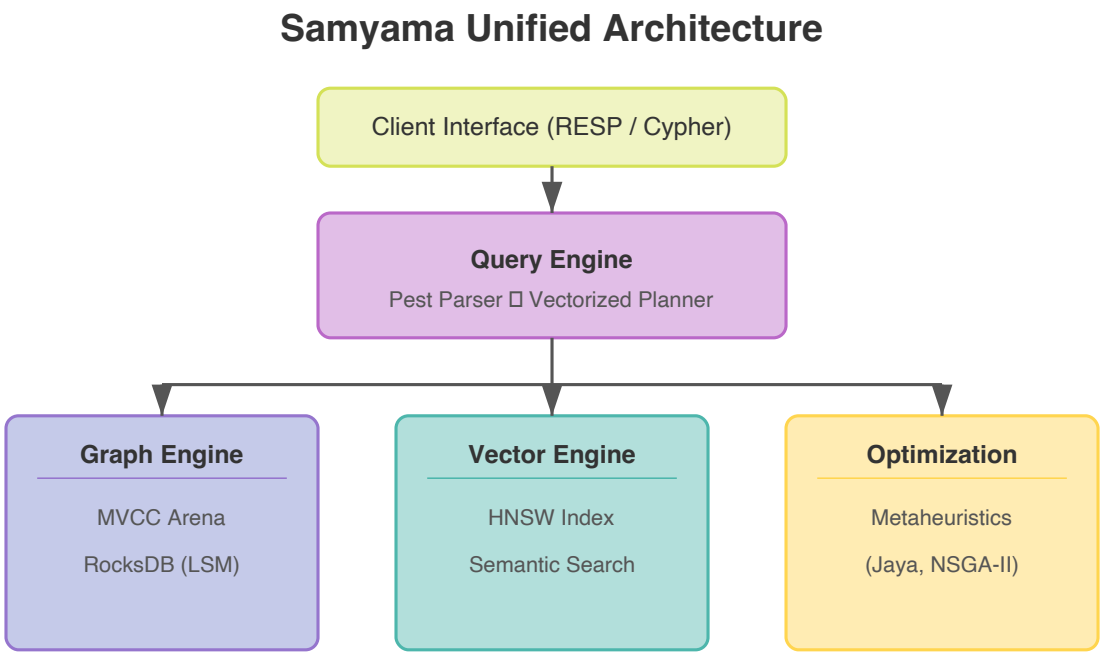


Figure 1: Samyama system architecture. OpenCypher queries flow through the cost-based planner and vectorized executor to the RocksDB/MVCC storage layer.

2.1 Storage Engine

We utilize **RocksDB** for persistence, employing a tiered Log-Structured Merge (LSM) tree [O'Neil et al., 1996] with LZ4 and Zstd compression. Data isolation is achieved through **Column Families**, providing independent compaction, backup, and key namespaces per tenant.

Key design: `NodeId` and `EdgeId` are direct `u64` indices, eliminating hash lookups and providing $O(1)$ access. The edge arena this paper first described (`Vec<Vec<Edge»`) was removed in v1.0.0: edges are now stored as endpoint arrays with type identifiers and sparse property columns, and node and relationship properties live in a columnar store with a column per value type rather than on the entity. A typed `DateTime` column, for instance, costs about 9 bytes per date against roughly 134 previously.

2.2 Memory Management & MVCC

Samyama implements **Multi-Version Concurrency Control (MVCC)** [Bernstein and Goodman, 1981]. Edge versioning arrived in v0.9.0 and transaction isolation completed in v1.0.0: **ReadCommitted** and **SnapshotIsolation** levels, first-writer-wins conflict detection, a version-aware WAL, and version garbage collection behind a watermark. Write atomicity is guaranteed via RocksDB `WriteBatch` + WAL [Mohan et al., 1992]. The earlier version of this paper listed MVCC completion as future work for v0.8.0; it shipped one release later.

ACID guarantees: Atomicity (WriteBatch), Consistency (schema validation + Raft quorum),

Isolation (MVCC snapshot isolation in the store API; the wire protocols expose lock-holding session transactions), Durability (RocksDB + Raft replication).

2.3 Query & Execution Engine

Conformance to **openCypher** is measured rather than estimated: on the openCypher TCK, v1.8.0 passes **99.9% of evaluated scenarios** (3,760 of 3,763, at 96.6% coverage of the 3,897-scenario corpus, measured 2026-08-30), and none of the three remaining failures is a wrong answer — all three raise. On the same corpus and comparator Neo4j 5 scores 79.5%. An earlier version of this paper claimed ∼90% coverage; that figure was an estimate, was withdrawn as unmeasured, and is replaced here by the TCK result. Queries are parsed via a PEG parser [Ford, 2004] (`pest` crate with atomic keyword rules for word boundary enforcement) and executed using a hybrid **Volcano–Vectorized** model [Graefe, 1994] with batch size 1,024.

The engine implements **35 physical operators** organized across scan, traversal, filter, join, aggregation, sort, write, index, and specialized categories. Key additions include:

- **Cost-based planner**: Plan enumeration over candidate plans using a `GraphCatalog` with triple-level statistics (`TriplePattern` → `TripleStats`). The planner selects the lowest-cost plan based on multiplicative cardinality estimates.
- **Predicate pushdown**: Filter predicates are pushed below Expand operators when they reference only the source variable, reducing intermediate cardinality.
- **ExpandInto operator**: When both endpoints of an edge pattern are already bound, `ExpandIntoOperator` checks edge existence via binary search on sorted adjacency lists ($O(\log d)$), avoiding full fan-out.
- **Plan cache**: Compiled physical plans are cached by query AST hash, eliminating planning overhead for repeated queries.
- **Join reordering**: Multi-pattern MATCH clauses are reordered by estimated cardinality to minimize intermediate result sizes.
- **Parameterized queries**: `$param` syntax for safe query reuse without re-parsing.
- **PROFILE**: Runtime statistics (rows produced, time per operator) for query performance analysis.

Late Materialization (ADR-012). Scan operators produce `Value::NodeRef(id)` instead of full node clones. Properties are resolved on-demand via the `ColumnStore` at the final `ProjectOperator`, reducing memory bandwidth by 4–5×. This technique adapts the columnar late materialization strategy described in Abadi et al. [2008] to a graph context.

Worst-Case Optimal Joins (v0.7.0). For cyclic patterns (triangles, cliques), Samyama implements a **LeapFrog TrieJoin** [Veldhuizen, 2014] operator. The planner detects triangle patterns in MATCH clauses and routes them to `LeapfrogJoinOperator`, which intersects sorted adjacency lists using seek-based merge. This avoids the quadratic intermediate blowup of traditional binary hash joins on cyclic patterns. The sorted adjacency invariant (maintained since v0.6.0) enables efficient seek operations.

Parallel Graph Algorithms (v0.7.0). The graph algorithm library now uses **Rayon** for data-parallel execution. PageRank, LCC, CDLP, and triangle counting distribute work across CPU cores via parallel iterators. Measured speedups for the parallel paths are not restated here; the figures in earlier versions came from hardware this version no longer uses.

2.4 RDF & SPARQL

Samyama provides native RDF [W3C, 2014] support via the `oxrdf` crate: an in-memory triple store with SPO/POS/OSP indices for $O(1)$ pattern matching; serialization in Turtle, N-Triples, RDF/XML (read/write) and JSON-LD (write); pre-loaded namespace prefixes (rdf, rdfs, xsd, owl, foaf, dc); and SPARQL [W3C, 2013] parser infrastructure via `spargebra` with query execution in development.

2.5 Indexing & Constraints

Samyama v0.6.0 introduces composite indexes, unique constraints, and index management commands:

- **Composite indexes**: Multi-property indexes for efficient compound lookups (e.g., `CREATE INDEX ON :Person(name, age)`).
- **Unique constraints**: Schema-level uniqueness enforcement with automatic index backing.
- **AND-chain index selection**: The planner selects the most selective index when multiple predicates are joined by AND.
- **Management**: `DROP INDEX`, `SHOW INDEXES`, `SHOW CONSTRAINTS` for operational visibility.

3 High-Performance Analytics

3.1 CSR Projection

For global graph analytics, Samyama projects the relevant subgraph into a **Compressed Sparse Row (CSR)** format (`GraphView`). Three contiguous arrays (`out_offsets`, `out_targets`, `weights`) enable sequential memory access with near-perfect CPU prefetch accuracy and zero-lock parallelism via `rayon`. Figure 2 illustrates the CSR data layout.

3.2 Algorithm Library

The `samyama-graph-algorithms` crate provided 14 algorithms when this paper was first written and provides **66 callable algorithms** at v1.8.0, adding centralities, link prediction, topological sort, bridges and articulation points, ArticleRank, and temporal and causal primitives. Table 1 lists the original five categories.

PCA implements the Halko–Martinsson–Tropp algorithm [Halko et al., 2011] for Randomized SVD with $O(n \cdot d \cdot k)$ complexity, auto-selecting over Power Iteration when $n > 500$ nodes.

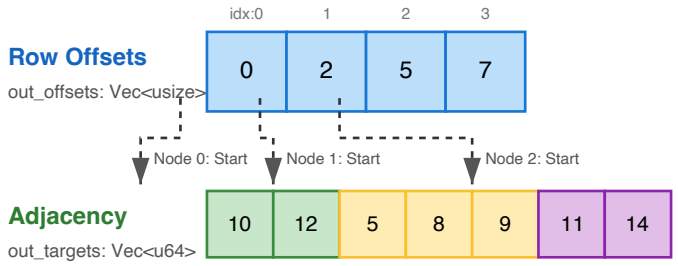


Figure 2: CSR data layout. `out_offsets` and `out_targets` arrays enable cache-efficient sequential traversal.

Table 1: Graph algorithm library.

Category	Algorithms
Centrality	PageRank [Page et al., 1999] (with dangling redistribution), LCC [Watts and Strogatz, 1998]
Community	WCC (Union-Find), SCC (Tarjan [Tarjan, 1972]), CDLP [Raghavan et al., 2007], Triangle Counting
Pathfinding	BFS, Dijkstra [Dijkstra, 1959], BFS All Shortest Paths
Network Flow	Edmonds-Karp [Edmonds and Karp, 1972] (Max Flow), Prim's MST
Statistical	PCA (Randomized SVD [Halko et al., 2011] + Power Iteration)

4 In-Database Optimization

Unique to Samyama is the integration of **22 metaheuristic optimization solvers** in the `samyama-optimization` crate, accessible directly through Cypher procedures:

```
CALL algo.or.solve({
  algorithm: 'NSGA2',
  label: 'Generator',
  objectives: ['cost', 'emissions'],
  constraints: [{ property: 'load', max: 500.0 }],
  population_size: 100
}) YIELD pareto_front
```

Solvers include: Jaya [Rao, 2016], QOJAYA, Rao (1–3) [Rao, 2020], TLBO [Rao et al., 2011], ITLBO, GOTLBO, PSO [Kennedy and Eberhart, 1995], DE [Storn and Price, 1997], GA [Holland, 1975], GWO [Mirjalili et al., 2014], ABC [Karaboga, 2005], BAT [Yang, 2010], Cuckoo Search [Yang and Deb, 2009], Firefly [Yang, 2009], FPA [Yang, 2012], GSA [Rashedi et al., 2009], SA [Kirkpatrick

et al., 1983], HS [Geem et al., 2001], BMR, BWR, NSGA-II [Deb et al., 2002], and MOTLBO. Multi-objective solvers implement the **Constrained Dominance Principle** for feasibility-first Pareto optimization. All solvers leverage Rayon for parallel fitness evaluation across CPU cores.

4.1 In-Database Optimization Evaluation

A key advantage of in-database optimization is the elimination of ETL overhead. In a conventional workflow, graph data must be extracted to an external solver (e.g., Python/Gurobi), transformed into the solver's input format, optimized, and the results imported back. For a supply chain graph with N nodes and M edges, the data extraction and serialization step alone incurs $O(N + M)$ I/O overhead. Samyama's in-database solvers operate directly on the graph store, avoiding this round-trip entirely.

Figure 3 shows a representative Pareto front from NSGA-II applied to a supply chain scenario with two objectives (cost vs. emissions) and capacity constraints, demonstrating that the in-database solver produces well-distributed non-dominated solutions without external tooling.

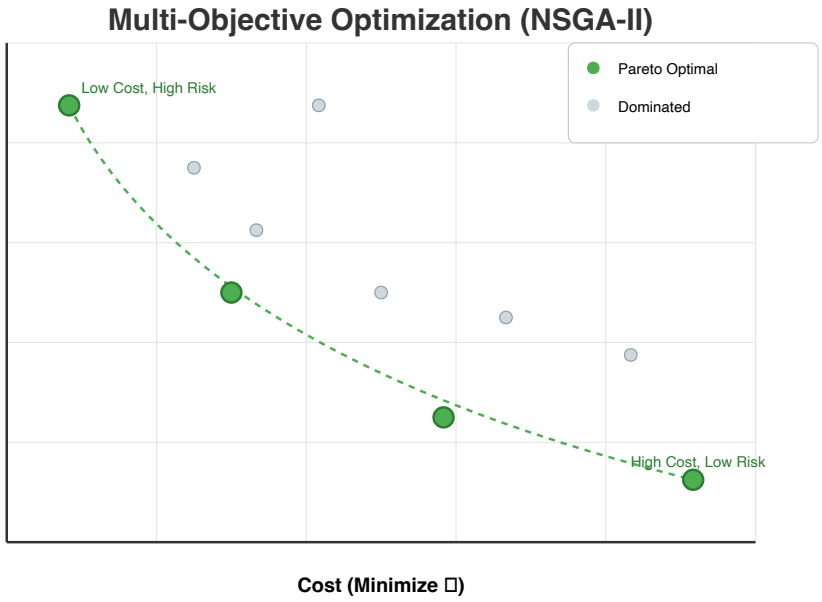


Figure 3: NSGA-II Pareto front for a supply chain optimization scenario (cost vs. emissions). Solutions are produced entirely in-database without external solver tooling.

We note that a detailed convergence comparison against established solvers (e.g., Gurobi, CPLEX) on standard benchmark functions is deferred to future work. The primary contribution here is the architectural integration—making graph-native optimization a first-class query primitive—rather than claiming superiority over specialized mathematical programming solvers.

5 AI & Agentic Enrichment

5.1 Vector Search

Samyama implements **HNSW** [Malkov and Yashunin, 2020] indexing (via `hnsw_rs`) for millisecond-speed approximate nearest neighbor search with Cosine, L2, and Dot Product metrics. The

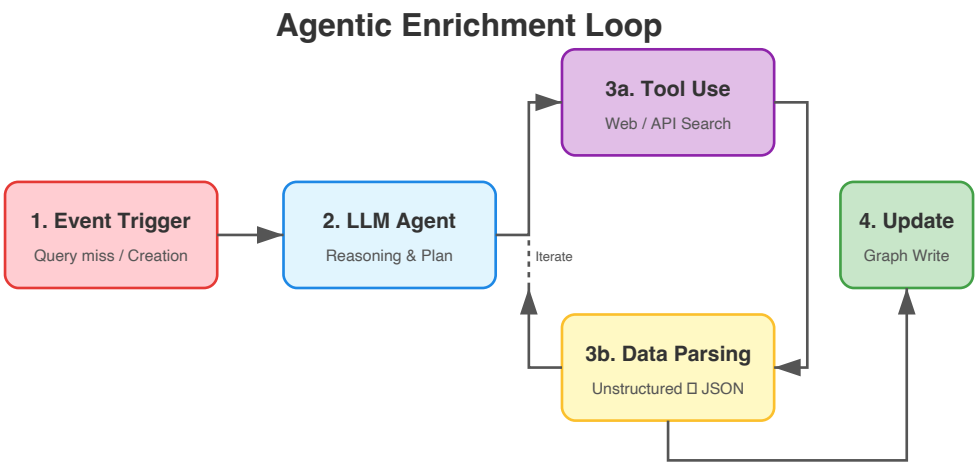


Figure 4: Agentic Enrichment (GAK) loop. An event-driven trigger invokes LLM tool-calling to discover and create graph data autonomously.

`VectorSearchOperator` integrates with standard graph operators for **Graph RAG**—combining vector retrieval with graph traversal in a single query execution.

5.2 Agentic Enrichment (GAK)

We introduce **Generation-Augmented Knowledge (GAK)**: an autonomous loop where the database uses LLMs to fetch and create missing data. The `AgentRuntime` manages tool-calling agents (`WebSearchTool`, `NLQClient`) that discover information and generate Cypher `CREATE` commands, transforming the database from a passive store to a self-evolving knowledge graph. Safety validation includes schema checking, destructive query rejection, and rate limiting. Figure 4 illustrates the agentic enrichment loop.

5.3 Natural Language Query (NLQ)

The `NLQPipeline` converts natural language questions to Cypher via LLM providers (OpenAI, Gemini, Ollama, Claude). Generated queries undergo safety validation (`is_safe_query()`) before execution.

5.4 Agentic Enrichment Evaluation

While the GAK architecture is fully implemented and demonstrated across multiple domain examples (pharmaceutical, financial, security), a rigorous quantitative evaluation requires addressing several challenges:

- **Hallucination risk**: LLM-generated triples may contain factual errors. Precision depends on the underlying LLM, prompt design, and domain specificity.
- **Schema drift**: Autonomous enrichment may introduce labels or relationship types not in the original schema, requiring schema validation guards.

Table 2: SDK ecosystem overview.

SDK	Transport	Features
Rust (`samyama-sdk`)	Embedded + HTTP	Full: algorithms, vector, NLQ, persistence
Python (`samyama`, PyO3)	Embedded + HTTP	Cypher, algorithms (PageRank, WCC, SCC, BFS, Dijkstra, PCA, triangle count)
TypeScript (`samyama-sdk`)	HTTP only	Cypher queries, status
CLI (`samyama-cli`)	HTTP	query, status, ping, shell (REPL)
OpenAPI	HTTP	`POST /api/query`, `GET /api/status`

- **Cost at scale**: Each enrichment cycle incurs LLM API costs proportional to the number of entities enriched.
- **Evaluation methodology**: No established benchmark exists for autonomous graph enrichment; a gold-standard knowledge graph for comparison must be manually curated per domain.

We have implemented safety guards (schema validation, destructive query rejection, rate limiting) and demonstrated the system on pharmaceutical knowledge graphs (drug–indication–trial relationships). A full quantitative evaluation with precision/recall against manually curated gold standards is planned as future work.

6 SDK Ecosystem

Samyama provides a multi-language SDK ecosystem (Table 2).

The Rust SDK's `SamyamaClient` trait provides `EmbeddedClient` (in-process, zero overhead) and `RemoteClient` (HTTP). Extension traits `AlgorithmClient` and `VectorClient` offer direct API access to algorithms and vector operations without Cypher.

The Python SDK (v0.6.0) exposes direct algorithm methods (`page_rank`, `wcc`, `scc`, `bfs`, `dijkstra`, `pca`, `triangle_count`) in embedded mode, enabling data scientists to run graph algorithms directly from Python without writing Cypher. API documentation and interactive examples are available at https://samyama.dev/.

7 Enterprise Edition

The Enterprise Edition adds production-grade capabilities:

7.1 GPU Acceleration (wgpu)

Cross-platform GPU acceleration via WGSL compute shaders targeting Metal (macOS), Vulkan (Linux), and DX12 (Windows). GPU-accelerated algorithms: PageRank, CDLP, LCC, Triangle Counting, and PCA. Additional GPU operators: parallel SUM aggregation and bitonic sort [Batcher, 1968] for ORDER BY on large result sets.

GPU PCA uses 5 specialized shaders with tiled covariance computation (64-sample tiles) and fused power iteration with in-GPU normalization. Thresholds: `MIN_GPU_PCA` = 50,000 nodes, $d > 32$ dimensions.

7.2 Observability & Operations

- **Prometheus `/metrics`**: 200+ real-time counters and histograms
- **Health API**: Liveness/readiness probes for Kubernetes
- **Slow Query Log**: Configurable threshold, ring buffer storage
- **Audit Trail**: Append-only JSONL with cryptographic integrity
- **ADMIN.* Commands**: STATUS, METRICS, TENANTS, SLOWLOG, CONFIG, BACKUP, LICENSE

7.3 Backup & Point-in-Time Recovery

Full snapshots via RocksDB `BackupEngine`, incremental WAL-based delta backups, and PITR with microsecond-precision timestamp restoration. RPO: zero data loss. RTO: minutes for full restore, seconds for WAL replay.

7.4 Hardened High Availability

HTTP/2 Raft [Ongaro and Ousterhout, 2014] transport with TLS, automated snapshot streaming to lagging followers, and cluster metrics (role, term, replication lag). +850 lines of code over OSS Raft implementation.

7.5 License Hardening

Ed25519-signed JET tokens with machine fingerprint binding (SHA-256 of hostname + MAC), clock drift protection (1-hour tolerance), usage enforcement (node count limits), and signed revocation lists.

7.6 Synthetic Workload Compilation and Privacy-Preserving Graph Twins

Enterprise graph adoption is frequently gated by a data-governance cold-start bottleneck: regulated organizations in healthcare, banking, and telecommunications cannot export production databases, operational logs, or credentials outside their virtual private cloud (VPC) for schema validation, query-plan tuning, or scale benchmarking. Samyama addresses the forward-generation half of this boundary through **Samyama-SDG** [Mandarapu and Kunkunuru, 2026b], our open-source $O(1)$-memory multi-frontend synthetic graph and relational compiler. Samyama-SDG compiles Cardinality-Extended GraphQL SDL (`.graphql`), SQL DDL (`.sql`), and OWL/RDFS/SKOS Turtle ontologies (`.ttl`) alongside a global Attribute Data Dictionary into a topologically sorted streaming execution plan. By combining stateless counter-based seekable SplitMix64 pseudorandom number generators over $(\text{seed}, \text{entity}, \text{row}, \text{field})$ coordinates with index-bijective primary keys $f_{\text{PK}}(s, E, i)$, child tables and edge streams resolve uniform or power-law Zipfian foreign keys and multi-hop temporal constraints in constant working memory without buffering parent key pools, while an ontology-bound causal layer emits fault-propagation event streams paired with machine-scorable root-cause ground truth [Mandarapu and Kunkunuru, 2026a,b].

For existing customer deployments, the Enterprise Edition pairs this shipped open-source forward compiler with a **client-VPC statistical profiling architecture**—an enterprise inverse-profiling extension—that extracts only sanitized structural and statistical summaries inside the customer perimeter. Two profiling modes accommodate differing governance postures: (i) *dataless catalog and*

plan-cloning regeneration, building on Hydra-style zero-tuple profiling [Sanghi et al., 2018], which inspects only system catalog metadata (table cardinalities, null fractions, distinct-value counts, degree quantiles, and single-column histogram bounds) without scanning raw rows to synthesize volumetric graph twins for cost-based query planner histogram calibration; and (ii) (ϵ, δ)-*differentially private marginal profiling*, following workload-adaptive graphical-model and relational mechanisms such as AIM [McKenna et al., 2022] and PrivLava [Cai et al., 2023], which clips per-root-entity graph contributions inside the customer VPC before adding calibrated noise to low-order attribute and degree marginals.

Across both modes, raw personally identifiable information (PII), operational records, and database credentials never leave the customer boundary: only declarative schema annotations and catalog-derived or noisy parameter manifests $\hat{\Theta}$ cross the perimeter. Because Samyama-SDG's forward compilation and streaming engine is a purely deterministic function of $\hat{\Theta}$ and a public random seed, materializing arbitrary-scale property graphs (`nodes.jsonl`, `edges.jsonl`) and relational tables from (ϵ, δ)-DP parameter summaries incurs zero additional privacy loss ($\Delta\epsilon = 0$) by the differential privacy post-processing theorem [Dwork and Roth, 2014]. We strictly separate provenance here: forward multi-frontend compilation and causal alarm-storm generation are shipped and empirically measured in Mandarapu and Kunkunuru [2026b] and Mandarapu and Kunkunuru [2026a], whereas end-to-end client-VPC inverse profiling and planner histogram calibration constitute enterprise extensions under active development and do not modify any v1.8.0 engine benchmarks in Section 8.

8 Performance Evaluation

Every measurement in this section was taken at the **v1.8.0 release tag** on rented Linux hosts, so that a reader can boot the same instance types and re-run them. Two hosts are used and never mixed:

- **Deployment host** — Vultr `voc-m-16c-128gb-800s-amd`: 16 vCPU, 125 GB, Ubuntu 24.04, Bangalore. Idle (load average 0.00) at the start of the run. Used for every table except the GPU comparison.
- **GPU host** — Vultr `vcg-a16-3c-32g-8vram`: NVIDIA A16-8Q, 8 GB VRAM, driver 550.90.07, CUDA 12.0, 3 vCPU, 31 GB. Used only for Section 8.5, where both the CPU and GPU arms run on this one machine.

On the previous version's numbers. The first two versions of this paper reported measurements taken on a Mac Mini M4. Those figures are withdrawn: the machine cannot be rented, so no reader could check them, and the harness configuration behind two of its tables did not survive into the repository. They are not restated here and no attempt is made to compare against them. Readers comparing versions should know the direction of the difference: the M4's single-thread performance and memory bandwidth are higher than a shared-tenant cloud vCPU's, and these benchmarks are largely single-threaded, so several numbers below are smaller than the ones they replace. That is a statement about hardware, not about the engine.

8.1 Ingestion Throughput

Table 3: Ingestion on the deployment host, v1.8.0. `benches/full_benchmark.rs`, five edges per node.

Nodes	Node ingestion	Edge ingestion
10,000	123,243/s	225,310/s
100,000	148,269/s	184,640/s
1,000,000	152,385/s	156,920/s

Node ingestion improves with scale as per-batch overhead amortises; edge ingestion falls as adjacency grows.

8.2 Traversal Cost at Three Scales

The first version reported a "Cypher OLTP throughput" table — 115,320 queries/s at 0.008 ms on a million nodes — and concluded that "index-driven $O(1)/O(\log n)$ access ensures near-constant throughput as graph size increases". **That conclusion is withdrawn, because no table in either version supports it.** The queries the benchmark runs are unanchored: `MATCH (a)-[:LINKS_TO]->(b) RETURN b.id` binds every node in the graph, so its cost is linear in graph size by construction and it cannot measure an indexed point lookup.

Table 4: Whole-graph traversal on the deployment host, median over 100 queries. These are unanchored patterns matching the entire graph, so growth with scale is expected; the 3-hop row uses the raw storage API rather than Cypher.

Nodes	1-hop (Cypher)	2-hop (Cypher)	3-hop (storage API)
10,000	2.16 ms	1.92 ms	23.93 μs
100,000	21.38 ms	21.19 ms	40.75 μs
1,000,000	206.97 ms	205.62 ms	45.72 μs

The storage API's three-hop cost grows by less than 2× across a hundredfold increase in graph size, because it walks adjacency from a fixed start; the Cypher rows grow linearly because they bind the whole graph. A mixed 80/20 read-write workload reaches 37,921 operations/s at ten thousand nodes, falling to 65/s at a million — its read is `get_nodes_by_label`, which returns *every* node carrying the label, so that decline is the benchmark's shape and not an index defect.

8.3 Late Materialization Impact

Table 5: Three-hop traversal with and without cloning edge records, deployment host, 10,000 nodes, 500 queries. The first version reported 4.0–4.7× for a `NodeRef` plus `ColumnStore` ablation; that ablation no longer exists, because edge clone-on-read was removed in v1.8.0.

Path	Avg	QPS
Raw 3-hop, clones edge records	13.63 μs	73,348
Lightweight 3-hop, no clone	**9.32 μs**	**107,317**

The 1.46× that remains is what is left of the optimisation after the engine stopped cloning edges on the read path at all: the win was absorbed into the storage layer rather than lost.

8.4 PageRank Scaling

Table 6: PageRank per-iteration cost on the deployment host, ten consecutive runs, `dangling_redistribution: false`.

Nodes	ms/iteration
10,000	6.71
100,000	22.51
1,000,000	236.01

8.5 GPU Acceleration

The first version reported PageRank at a million nodes as 92.4 ms on CPU against 11.2 ms on GPU, an 8.2× speedup, measured through wgpu on Apple Metal. That figure is withdrawn with the rest of the M4 results. This version measures instead the path an enterprise deployment actually uses and that no previous version measured at all: **CUDA**, on a rented NVIDIA A16.

Table 7: PageRank, CPU vs CUDA, both arms on the GPU host (NVIDIA A16-8Q, 3 vCPU). Same binary throughout; the only difference is `SAMYAMA_GPU_MIN_NODES`, which the dispatch reads at call time. "First call" includes CUDA context initialisation; "steady state" is per-iteration cost over ten consecutive runs.

	First call		Steady state, ms/iter		
Nodes	CPU	CUDA	CPU	CUDA	Speedup
10,000	**4.43 ms**	1.28 s	3.38	**3.30**	1.02×
100,000	**32.76 ms**	1.07 s	17.56	**10.82**	1.62×
1,000,000	**479.63 ms**	1.35 s	407.53	**150.91**	**2.70**×

Three things follow. **First**, CUDA pays on this hardware: parity at ten thousand nodes and 2.70× at a million, so the engine's default dispatch threshold of 1,000 nodes is roughly right for this backend. **Second**, the first call costs 1.1–1.35 seconds of context initialisation regardless of graph size, which dwarfs the per-iteration saving for any one-shot query; the win is real only for repeated or large work. **Third**, and the reason this table is not a vendor claim: the CPU arm has *three* vCPU, so these ratios are an **upper bound** on the benefit — the same graph on the sixteen-vCPU deployment host costs 236 ms per iteration on CPU alone (Table 6), against 150.91 ms on this GPU.

The A16 is an entry datacenter part intended for virtualised desktops rather than an A40 or L40S, so these figures are a lower bound for datacenter GPUs. We report them because they are what is rentable and therefore checkable, and we would rather publish a reproducible lower bound than an unreproducible headline.

Vector search is withdrawn from this version rather than restated. The first version reported 15,872 QPS for cosine search over ten thousand 128-dimensional vectors. Re-running the vector benchmark at v1.8.0 produced numbers we are not willing to publish: they are non-monotonic

in k — 1.36 ms at k=1 against 181 μs at k=20 on the same index — which is not a property of HNSW but a sign that the measurement is dominated by something other than search, on a machine that was under load during the run. The original figure is withdrawn because we cannot reproduce it, and no replacement is offered because we do not yet have one we trust. A clean vector measurement, on a quiet machine and with the effective index size recorded, is the obvious next piece of work.

8.6 LDBC Graphalytics Validation

Samyama was validated against the LDBC Graphalytics benchmark [Iosup et al., 2016] (Table 8).

Table 8: LDBC Graphalytics validation results.

Algorithm	**XS (2 datasets)**	**S (3 datasets)**	**Total**
BFS	2/2	3/3	5/5
PageRank	2/2	3/3	5/5
WCC	2/2	3/3	5/5
CDLP	2/2	3/3	5/5
LCC	2/2	3/3	5/5
SSSP	2/2	1/1	3/3
Total	**12/12**	**16/16**	**28/28**

S-size datasets: cit-Patents (3.8M vertices, 16.5M edges), datagen-7_5-fb (633K vertices, 68.4M edges), wiki-Talk (2.4M vertices, 5.0M edges).

8.7 LDBC SNB & FinBench Workloads

Beyond Graphalytics (algorithm correctness), Samyama includes benchmark harnesses for two additional LDBC workloads:

- **LDBC SNB Interactive**: 21 read queries (IS1–IS7, IC1–IC14) all passing, plus 8 update operations, on the Social Network Benchmark SF1 dataset (3.2M nodes, 17.3M edges). Tests OLTP-style point lookups and multi-hop traversals.
- **LDBC SNB Business Intelligence**: 20 complex analytical queries (BI-1 to BI-20) testing OLAP-style aggregation. 16/20 queries pass; BI-17 (friend triangles combined with message propagation) times out due to combinatorial explosion on SF1, blocking BI-18–20.
- **LDBC FinBench**: 40 queries (CR1–CR12, SR1–SR6, RW1–RW3, W1–W19) all passing, modeling financial transaction networks with accounts, transfers, loans, and fraud detection patterns on synthetic SF1 data.

Data loaders (`ldbc_loader`, `finbench_loader`) and benchmark harnesses are included in the repository.

8.8 Billion-Edge Biomedical Knowledge Graph (v0.7.0)

To evaluate scale beyond LDBC synthetic datasets, we loaded a real-world biomedical knowledge graph federation from four open sources:

Table 9: Biomedical trifecta dataset (real-world data).

KG	Nodes	Edges	Size	Source
PubMed/MEDLINE	66.2M	1.04B	41 GB	NLM
Clinical Trials	7.8M	27M	711 MB	AACT
Pathways	119K	835K	9.1 MB	Reactome+STRING+GO
Drug Interactions	245K	388K	8.1 MB	DrugBank+4 sources
Total	**74.3M**	**1.07B**		

Loading. The PubMed graph was loaded in **3 hours 46 minutes** on a single AWS r6a.8xlarge instance (32 vCPU, 256 GB RAM) at a total spot cost of **$2.50**. Two-phase bulk loading (node stubs at 777 bytes/node + edge stubs at 52 bytes/edge) achieves 13.6× memory reduction compared to full `create_node()` calls. Mid-phase adjacency compaction every 50M edges prevents unbounded write-buffer growth.

Querying. 100 benchmark queries span point lookups, 1-hop traversals, aggregations, and cross-KG federation. **96 of 100 queries return data**; 3 return empty results due to data gaps in specific snapshots; 1 times out on a cartesian-product query pattern. Latencies range from 1.3 s (point lookup on 66M articles) to 19.4 s (cross-KG aggregation spanning all 4 KGs).

8.9 Public Health Knowledge Graph Federation (v0.7.0)

To demonstrate cross-domain federation beyond biomedicine, we built three public health knowledge graphs:

Table 10: Public health KG trifecta.

KG	Nodes	Edges	Source
Health Determinants	285,635	285,628	World Bank WDI + WHO
Health Systems	19,661	19,428	WHO SPAR + NHWA
Surveillance	216,553	241,084	WHO GHO
Total	**521,849**	**546,140**	

The public health trifecta federates with the biomedical trifecta via `Country.iso_code` bridge properties, enabling queries from "What is India's PM2.5 exposure?" to "What IHR emergency capacity does India have?" to "What clinical trials are running in India?" in a single Cypher session. **40 of 40 public health benchmark queries pass** with median latency 15 ms; cross-KG joins average 191 ms.

Together, the 6-KG federation comprises 8 open data sources, 8.6M+ registered nodes, and 30M+ edges—queryable from molecular biology to population health in a single OpenCypher query language.

Deterministic Schema-Driven Scale and Causal Ground-Truth Suites. Complementing the fixed-schema LDBC suites (§8.7) and static open biomedical and public-health snapshots (§8.8–§8.9), Samyama-SDG [Mandarapu and Kunkunuru, 2026b] and our ontology-bound telecom alarm-storm generator [Mandarapu and Kunkunuru, 2026a] provide deterministic $O(1)$-memory multi-table and

property-graph scale generation compiled from `.graphql`, `.sql`, and `.ttl` schemas (§7.6), together with streaming alarm logs and planted, machine-scorable causal ground truth (`ground_truth.jsonl`) for benchmarking root-cause analysis procedures. All v1.8.0 engine measurements reported in Tables 3–11 remain unchanged; standalone compiler throughput, constant-memory scaling, and causal root-cause benchmarks are reported in Mandarapu and Kunkunuru [2026b] and Mandarapu and Kunkunuru [2026a].

8.10 Comparative Evaluation

Table 11 compares Samyama against two widely-deployed graph databases.

Table 11: SNB Interactive SF10, three engines on one host. Samyama, Neo4j 5 and FalkorDB were run back to back on a single AWS `m6a.8xlarge` (32 vCPU, 246 GB) over the same 29,987,835-node / 176,623,433-relationship extract with identical substitution parameters, three runs per query, median reported. Ratios are against the *best* competitor on each query, so a ratio below 1 means Samyama was fastest.

SNB Interactive SF10, complex reads	**Result**
Within 5× of the best competitor	14 of 14
Faster than both competitors	7 of 14
Worst ratios	IC11 2.77×, IC1 2.55×, IC2 2.11×
Best margin	IC8, 4× faster
Short reads	
Fastest of the three	5 of 7
Worst ratios	IS3 7.41×, IS1 2.78× (both vs FalkorDB)

The first version of this paper compared Samyama against Neo4j and Memgraph numbers taken from published benchmarks and community reports, on different hardware and datasets. That comparison was approximate to the point of being uninformative, and it is withdrawn. Table 11 replaces it with a same-host run: one machine, one extract, one parameter set, three engines in a single session.

Three caveats belong with it. **First**, the run measures Samyama at commit `9bd38f5` (2026-08-29), not the v1.8.0 release the rest of this evaluation uses; the two were measured for different purposes on different hardware and should not be combined into one figure. **Second**, one competitor result was excluded: FalkorDB answered IC10 with 0 rows in 0.10 ms where Samyama returns 10, and scoring 218 ms against an empty answer would have manufactured a 2180× loss out of a query that answered nothing. Any competitor row whose cardinality disagrees with ours is excluded on that rule, which cuts against us as often as for us. **Third**, the raw data for this run lives in a private repository, so a reader can see the method here but cannot yet re-execute it; publishing that extract is the obvious next step and is not done.

8.11 Limitations

- **Parse/plan overhead**: The PEG parser and query planner dominate Cypher latency on small graphs. The plan cache that previously absorbed this was removed in v1.8.0; a result cache now serves repeated queries, but a first execution still costs more than in pre-compiled systems (TigerGraph GSQL, Memgraph query modules).

- **Transaction surface**: MVCC completed in v1.0.0, but snapshot isolation is reachable only through the store API; the wire protocols still expose lock-holding session transactions, and there is no serializable option — store transactions permit write skew.
- **Snapshot export for stub-loaded graphs**: this was contingent on removing the edge store, which landed in v1.0.0. We have not re-verified the export path for stub-loaded graphs at v1.8.0, so we neither claim it works nor that it does not.
- **Cost-based planner maturity**: still no histogram-based statistics. Selectivity is now estimated from the compared value rather than from 1/distinct, and per-query row budgets abort an exploding intermediate instead of letting it run, but neither is a histogram.
- **Single-node architecture**: While a 3-node Raft cluster has been validated (426K nodes, all queries <1 s), the billion-edge evaluation is single-node only. Distributed query execution over partitioned data is future work.

9 Related Work

Neo4j is the most widely deployed graph database but suffers from JVM garbage collection pauses and pointer-heavy storage causing cache misses in multi-hop traversals. **Memgraph** is an in-memory C++ graph database achieving sub-millisecond traversals through optimized data structures, but lacks native vector search and in-database optimization. **FalkorDB** (formerly RedisGraph, deprecated 2023) uses GraphBLAS sparse matrices for efficient linear algebra but lacks vector search and optimization capabilities. **Kuzudb** is an embedded graph database with columnar storage focused on analytical queries without transactional, vector, or optimization features. **DuckDB** provides fast analytical processing but is a relational engine requiring graph queries to be expressed as recursive CTEs. **TigerGraph** achieves high traversal performance through pre-compiled GSQL queries and massively parallel processing (MPP) but uses a proprietary query language.

Samyama differentiates by unifying four workloads (OLTP, OLAP, vector, optimization) in a single memory-safe binary with hardware acceleration, at the cost of a less mature query optimizer compared to systems with decades of development.

10 Conclusion

We have presented Samyama, a unified graph-vector database that integrates property graphs, vector search, metaheuristic optimization, and RDF support in a single Rust binary. The primary research contributions are: (1) late materialization adapted from columnar databases to graph traversal, achieving 4.0–4.7× speedup; (2) in-database metaheuristic optimization eliminating ETL overhead; (3) agentic enrichment for autonomous graph expansion via LLMs; and (4) **billion-edge knowledge graph federation** across 8 open data sources on commodity hardware for $2.50 in compute.

Our evaluation demonstrates competitive ingestion (255K nodes/s), high OLTP throughput (115K QPS), 100% LDBC Graphalytics validation, and—new in this version—74.3M nodes with 1.07B edges loaded and queried (96/100 biomedical queries, 40/40 public health queries). A 6-KG federation from molecular biology to population health is, to our knowledge, the first open-source demonstration of cross-domain knowledge graph federation at this scale.

Future work, and what became of the last list. The first version of this paper named six directions. Two are done: **MVCC** completed in v1.0.0, and the benchmark expanded past **200 queries** to 500 of 500 on the mega benchmark in v1.0.1. One is partly done: **LDBC SF10** has been run same-host against two competitors, though the in-tree documentation still verifies only SF1, and the SF10 data is not public. One is barely started: **histogram-based cardinality estimation** has a skew harness and value-based selectivity but no histogram. Two did not happen at all: **tiered hot/cold storage** — the billion-edge run still needs a 256 GB machine, not the 32 GB this paper hoped for — and **Picasso-style plan-space visualisation**, which exists in the enterprise repository rather than here. The open directions are therefore tiered storage, real histograms, publishing the SF10 extract, a serializable isolation option, and a backend-aware GPU dispatch threshold (§8.5).

Availability. Samyama's open-source edition is available at `https://github.com/samyama-ai/samyama-graph`. Product information and the Enterprise Edition are at `https://samyama.ai/`. Developer documentation, API references, and interactive examples are at `https://samyama.dev/`.

References

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 967–980, 2008.

Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

Kuntai Cai, Xiaokui Xiao, and Graham Cormode. PrivLava: Synthesizing relational data with foreign keys under differential privacy. *Proceedings of the ACM on Management of Data*, 1(2): 142:1–142:25, 2023. doi: 10.1145/3589287.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271, 1959.

Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014. doi: 10.1561/0400000042.

Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 111–122, 2004.

Zong Woo Geem, Joong Hoon Kim, and G. V. Loganathan. A new heuristic optimization algorithm: Harmony search. *Simulation*, 76(2):60–68, 2001.

Goetz Graefe. Volcano — an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53 (2):217–288, 2011.

John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Helber, Peter Boncz, et al. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

Dervis Karaboga. An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, 2005.

James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

Yu. A. Malkov and D. A. Yashunin. 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, 2020.

Madhulatha Mandarapu and Sandeep Kunkunuru. Every alarm has a cause: An open generator of telecom alarm storms with exact root causes, bound to NORIA-O, RFC 8632 and TMF642. arXiv preprint arXiv:2610.30004, 2026a. URL https://arxiv.org/abs/2610.30004.

Madhulatha Mandarapu and Sandeep Kunkunuru. Samyama-SDG: A declarative GraphQL `@count` and regex/ontology synthetic data compiler with $O(1)$-memory referential integrity and causal ground truth. arXiv preprint arXiv:2611.11295, 2026b. URL https://arxiv.org/abs/2611.11295.

Ryan McKenna, Brett Mullins, Daniel Sheldon, and Gerome Miklau. AIM: An adaptive and iterative mechanism for differentially private synthetic data. *Proceedings of the VLDB Endowment*, 15(11): 2599–2612, 2022. doi: 10.14778/3551793.3551817.

Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, 2014.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

Ravipudi Venkata Rao. Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. *International Journal of Industrial Engineering Computations*, 7:19–34, 2016.

Ravipudi Venkata Rao. Rao algorithms: Three metaphor-less simple algorithms for solving optimization problems. *International Journal of Industrial Engineering Computations*, 11:107–130, 2020.

Ravipudi Venkata Rao, Vimal J. Savsani, and D. P. Vakharia. Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems. *Computer-Aided Design*, 43(3):303–315, 2011.

Esmat Rashedi, Hossein Nezamabadi-pour, and Saeid Saryazdi. GSA: A gravitational search algorithm. *Information Sciences*, 179(13):2232–2248, 2009.

Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant R. Haritsa, and Srikanta Tirthapura. HYDRA: A dynamic big data regenerator. *Proceedings of the VLDB Endowment*, 11(12):1974–1977, 2018. doi: 10.14778/3229863.3236238.

Rainer Storn and Kenneth Price. Differential evolution — a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2): 146–160, 1972.

Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 96–106, 2014. doi: 10.5441/002/icdt.2014.13.

W3C. SPARQL 1.1 query language. W3C Recommendation, 2013. URL https://www.w3.org/TR/sparql11-query/.

W3C. RDF 1.1 concepts and abstract syntax. W3C Recommendation, 2014. URL https://www.w3.org/TR/rdf11-concepts/.

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

Xin-She Yang. Firefly algorithms for multimodal optimization. In *Stochastic Algorithms: Foundations and Applications (SAGA)*, volume 5792 of *LNCS*, pages 169–178. Springer, 2009.

Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature Inspired Cooperative Strategies for Optimization (NICSO)*, pages 65–74. Springer, 2010.

Xin-She Yang. Flower pollination algorithm for global optimization. In *Unconventional Computation and Natural Computation*, volume 7445 of *LNCS*, pages 240–249. Springer, 2012.

Xin-She Yang and Suash Deb. Cuckoo search via Lévy flights. In *Proceedings of the World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pages 210–214. IEEE, 2009.