EMBEDDED GRAPH DATABASES

BUILDING THE SQLITE OF GRAPH DATA

BUILD A PRODUCTION-QUALITY, EMBEDDED GRAPH DATABASE FROM FIRST PRINCIPLES WITH COMPLETE WORKING CODE

- EMBEDDABLE AND SERVERLESS
- TRANSACTIONAL AND DURABLE
- HIGH PERFORMANCE GRAPH ENGINE
- WRITE-AHEAD LOGGING AND CRASH RECOVERY
- COMPLETE SOURCE CODE, TESTS, AND BENCHMARKS

STORAGE ENGINES AND FILE FORMATS

QUERY PARSING, PLANNING, AND EXECUTION

GRAPH TRAVERSAL AND OPTIMIZATION

APIS, BINDINGS, TESTING, AND PRODUCTION HARDENING

ANDREW CARVER

Embedded Graph Databases

Building the SQLite of Graph Data

Steve Publications

This book is available at https://leanpub.com/embeddedgraphdatabases

This version was published on 2026-08-26

Contents

Building the SQLite of Graph Data

This book teaches you how to build a production-quality embedded graph database from first principles, with complete working code. You will design and implement a compact, embeddable, transactional, durable, high-performance graph database library that links directly into applications without requiring a separate server process , analogous in deployment philosophy to SQLite but engineered specifically for graph workloads. By the end, you will understand every major subsystem of a modern database engine as it applies to graphs: storage pages and file formats, indexing, transactions, write-ahead logging and crash recovery, graph traversal optimization, query parsing and planning, execution engines, APIs, bindings, testing, performance engineering, and production hardening.

Introduction: Why the SQLite of Graph Data Does Not Yet Exist

There is a gap in our data infrastructure that most engineers never notice because they have not needed to look for it.

SQLite has become so ubiquitous that its existence feels like background radiation. It runs inside your browser, your phone, your operating system, your embedded device, your desktop application, your containerized microservice. It requires no daemon, no configuration file, no port number, no authentication. You drop a single file on disk, open it through a C library call, and you have a fully ACID-compliant relational database with indexes, transactions, and crash recovery. The entire engine is fewer than a megabyte of compiled code.

Now imagine you need to store graph data , relationships between users in a social network, dependencies between modules in a build system, entities and facts in a knowledge graph, permissions and roles in an authorization system. You want the same operational simplicity: one file, no server, link a library, query your graph. What do you use?

You reach for Neo4j and start a JVM process. You reach for Amazon Neptune and provision a managed cluster. You reach for JanusGraph and configure Cassandra or HBase as a backing store. You reach for an in-memory structure like igraph or NetworkX and accept that your graph lives only as long as your process does. None of these options gives you what SQLite gave relational data: a zero-operation, embeddable, durable graph database that runs wherever your application runs.

This book exists to close that gap. We are going to build one , not by wrapping an existing engine, not by adding a graph layer on top of a key-value store, but by designing a storage format, indexing strategy, transaction system, and query engine from the ground up for graph workloads. The result will be a library we call Graphite: a compact, embeddable, transactional, durable graph database written in Rust that you can link into your application and use without ever starting a separate process.

What You Will Build

Graphite is designed to be the SQLite of graph data. Its defining characteristics are:

- **Embedded**: The database engine is compiled directly into your application as a library. There is no server process, no network port, no configuration daemon. Your application opens a file and queries it through function calls.
- **Single-file storage**: All graph data , nodes, edges, properties, indexes, transaction logs , resides in one or two files on disk that you can copy, back up, version-control, email to a colleague, or store on a USB drive.
- **ACID transactions**: Every modification is atomic, consistent, isolated, and durable. If your application crashes mid-transaction, committed changes survive and incomplete changes are rolled back automatically on recovery.
- **Property graph model**: Graphite stores labeled nodes with typed properties connected by typed edges with their own properties. This is the dominant data model in production graph systems, standardized as part of ISO/IEC 39075 GQL [4].
- **Pattern-matching query language**: You query Graphite using a Cypher-inspired declarative syntax that expresses graph patterns visually: find all users who know someone who works at a given company, find the shortest path between two nodes, count neighbors with specific properties.
- **High-performance traversal**: The storage layout is optimized for graph traversal , expanding neighbors of a node is fast because related data is physically clustered on disk and in memory, not scattered across random pages.
- **Crash recovery**: Graphite uses write-ahead logging to guarantee durability. When it restarts after a crash, it replays the log to recover committed transactions and rolls back incomplete ones. We will build deterministic crash-testing infrastructure to prove this works.
- **Small footprint**: The entire engine compiles to a few megabytes of code. No JVM, no container runtime, no external dependencies beyond the standard library.

This is not a toy project or an academic exercise. Every design decision is motivated by real workload requirements and backed by working code that you

can inspect, modify, benchmark, and deploy. The reference implementation is written in Rust because memory safety matters when you are building systems that manage persistent storage, zero-cost abstractions let us express complex data structures without runtime overhead, and the type system catches entire classes of bugs at compile time that would otherwise corrupt database files in production [2].

Why Graph Data Needs Its Own Embedded Engine

You might wonder why we cannot simply use SQLite to store graphs. After all, you can represent nodes as rows in a table and edges as rows in another table with foreign keys. Many applications do exactly this.

The problem is that relational storage layouts are optimized for set-oriented operations on tabular data, not for topological traversal of irregular structures. When you query a graph pattern like "find all friends-of-friends who have a given property," SQLite must perform multiple joins across edge tables, each join potentially scanning large portions of the table and following pointer chains through B-tree indexes. The physical layout of rows on disk does not respect graph topology: neighboring nodes are stored wherever the database allocator puts them, which means traversing from one node to its neighbors involves random I/O that thrashes CPU caches and disk heads.

Specialized graph storage layouts cluster adjacent data physically. In a node-centric design, all outgoing edges from a node are stored contiguously near that node's record. When you expand neighbors during traversal, the database reads a single page or a small sequence of pages sequentially, maximizing cache hit rates and minimizing I/O operations. Research on transactional graph storage systems like LiveGraph demonstrates that purely sequential adjacency list scans can be up to 7.45 times faster than B-tree-backed approaches on realistic workloads [6].

This is not just about raw speed. Graph traversal has fundamentally different access patterns from relational queries: irregular branching factors, hot nodes with millions of edges (supernodes), recursive exploration, visited-set tracking to avoid cycles, early termination when a result is found. An engine designed for these patterns can make optimizations that a general-purpose relational engine cannot , or would implement as awkward workarounds.

Conversely, you might consider using an existing embedded key-value store like RocksDB, LMDB, or redb and layer graph semantics on top. This approach

has merits: key-value stores provide durable storage primitives, efficient indexing, and proven crash recovery. But they still force you to encode graph topology as string keys ("node:1234", "edge:5678") and manage adjacency lists manually through prefix scans. The result is a functional system that lacks the tight integration between storage layout, traversal optimization, and query planning that a purpose-built graph engine provides.

An embedded graph database must be designed holistically: the on-disk format determines traversal performance, which influences index design, which affects transaction overhead, which constrains concurrency models, which shapes the query language semantics. You cannot assemble these pieces from unrelated components without paying a tax in complexity, performance, or correctness.

What This Book Is Not

This book is not a tutorial on using existing graph databases. If you want to learn Cypher syntax for Neo4j or Gremlin for JanusGraph, there are excellent resources for that. We will reference established systems and languages to ground our design choices, but the focus is always on building, not consuming.

This book is not about distributed graph systems. Systems like TigerGraph, Nebula Graph, or Amazon Neptune solve different problems: horizontally scaling graphs across clusters of machines, handling petabytes of data, providing multi-region replication. These are important challenges, but they are orthogonal to the embedded thesis. We deliberately exclude distributed consensus protocols, sharding strategies, and network communication layers because they would obscure the core material: how to build a correct, efficient graph database engine that runs in a single process on a single machine.

This book is not about graph algorithms as an end in themselves. We will implement traversal primitives , breadth-first search, depth-first search, neighbor expansion, path reconstruction , because they are fundamental operations that the storage engine must support efficiently. But we will not cover PageRank, community detection, centrality measures, or graph neural networks except where their memory access patterns illuminate storage design trade-offs.

This book is not a literature review or a feature catalog. Every section either explains something you need to understand, justifies a design decision with

evidence, or shows working code that implements the concept. There are no surveys of "the landscape," no opinionated rankings of graph databases, no buzzword-driven discussions of where the industry is heading.

How to Read This Book

The book is organized as a progressive construction project. Each chapter introduces new concepts and adds new capabilities to the Graphite implementation. The chapters build on each other: you cannot understand the query optimizer without understanding the storage layout, you cannot design crash recovery without understanding transactions, you cannot implement transactions without understanding the page cache.

If you are reading for learning, follow the chapters in order. Each one assumes familiarity with material from previous chapters and extends the implementation incrementally. The code evolves across chapters: early versions are simpler to illustrate core concepts, later versions add production hardening like error handling, concurrency safety, and corruption detection. When a chapter refactors earlier code, it explains what changed and why.

If you are reading for reference, use the chapter headings to find specific subsystems: storage format (Chapter 5), transactions (Chapter 10), crash recovery (Chapter 11), query language (Chapter 13), or performance engineering (Chapter 16). Each chapter is written as a self-contained deep dive into its topic, with enough context that you can understand it without reading the entire book first.

If you are implementing your own embedded graph database, treat Graphite as a reference architecture, not a template to copy. Different workloads justify different design choices. The book explains alternatives at every major decision point and documents why we chose one path over another for our target workload profile. Your requirements may differ, and that is fine , the goal is to give you enough understanding to make informed decisions about your own system.

Prerequisites

This book assumes you are comfortable with systems-level programming concepts: memory management (stack vs. heap, pointers/references, alignment), I/O operations (files, buffers, system calls), concurrency (threads, locks,

atomic operations), and basic algorithm analysis (time and space complexity). You should have experience with at least one compiled language , Rust is our implementation choice, but C++, Go, or Java will prepare you for the material.

You do not need prior database engineering experience. The book explains database internals from first principles: what a page is, how B-trees work, why write-ahead logging guarantees durability, how query optimizers choose execution plans. If you have used a database but never wondered what happens when you execute a query, this book will answer those questions in the context of building an actual system.

You do need curiosity and patience. Building a correct database engine is hard. Subtle bugs in page allocation, transaction ordering, or crash recovery can corrupt data silently and manifest only after hours or days of operation. The book emphasizes correctness over cleverness: simple designs that are easy to verify beat complex designs that are theoretically optimal but practically fragile.

The Journey Ahead

We begin by defining what an embedded graph database is and why it matters, then establish the mathematical foundations of graph structures as they relate to storage decisions. We will design the on-disk file format byte by byte, implement page management and indexing from scratch, build transactions and crash recovery with formal guarantees, create a query language and optimizer that turn declarative patterns into efficient execution plans, and expose everything through clean APIs and language bindings.

Along the way we will measure performance rather than assume it, test for crashes rather than hope for correctness, and explain every trade-off explicitly so you understand not just what we built but why we built it that way. By the final chapter, Graphite will be a working, documented, tested, benchmarked embedded graph database library , and you will have the knowledge to build your own.

Let us begin.

Chapter 1: The Embedded Graph Database Thesis

What "Embedded" Means: No Server, No Daemon, No Network

The word "embedded" in database terminology has a precise meaning that is often confused with related concepts like "lightweight," "local," or "serverless." An embedded database is one whose engine is compiled directly into the application process as a library. There is no separate server daemon to start, configure, or monitor. There is no network protocol between the application and the storage engine , all communication happens through function calls within the same address space. The database file is just a regular file on the filesystem that the application opens with standard I/O APIs.

SQLite is the canonical example. When your application links against libsqlite3.so and calls sqlite3_open(), the database engine runs inside your process using your threads, your memory allocator, your file descriptors. There is no sqlite-server daemon listening on port 5432. There is no authentication handshake, no query serialization over TCP, no connection pooling infrastructure. You open a file, execute statements through API calls, close the file when done. The entire operational model reduces to: does this file exist, and do I have permission to read and write it?

This deployment philosophy has profound consequences for system design:

- **Zero operational overhead**: There is no separate process to manage, restart, or scale. Database reliability is application reliability , if the application crashes, the database connection ends; when the application restarts, it reopens the file and recovers automatically through crash recovery mechanisms built into the engine.
- **No network latency**: All data access happens through in-process function calls. There is no serialization overhead, no TCP round-trip, no network partition to handle. Query latency is determined purely by computation and I/O, not by communication.

- **Process-local concurrency**: Multiple threads within the same application can share a database connection (with appropriate locking), but there is no native support for concurrent access from separate processes or machines without additional coordination mechanisms like file locks or shared memory.
- **File-based portability**: The entire database state resides in one or more regular files that you can copy, move, compress, encrypt, version-control, or email. Database backup is a filesystem operation, not a proprietary dump utility.
- **Minimal dependencies**: An embedded database library typically depends only on the operating system's standard libraries for file I/O, memory management, and threading. There are no external services, no container runtimes, no language-specific virtual machines required.

Graphite embraces this philosophy completely. It is designed to be linked into applications as a Rust crate (or accessed through FFI from other languages), opened via a single function call that takes a file path, queried through an in-process API, and closed when the application exits. No daemon, no network, no configuration files beyond the database file itself.

The Graph Workload Gap in Existing Embedded Systems

The embedded database market is dominated by relational engines (SQLite and its variants) and key-value stores (RocksDB, LMDB, redb). These systems excel at their intended workloads but struggle with graph data because their storage layouts and query models are not designed for topological traversal.

Consider a dependency graph for a build system: thousands of modules, each depending on several others. You need to answer questions like "what breaks if I change this module?" or "what is the shortest path from module A to module B?" If you store this in SQLite with nodes as rows and edges as foreign-key references, every traversal hop requires a join operation that scans an index and fetches rows from potentially scattered pages. For deep traversals (five or more hops), the query becomes a cascade of joins that is slow to plan and slow to execute.

Consider a knowledge graph for a desktop application: entities like people, places, organizations connected by relationships like "works at," "located in," "related to." You need pattern matching: find all people who work at companies

located in a given country and have collaborated with a specific person. In a relational schema, this is a multi-way join across entity and relationship tables. The query works but feels wrong , you are forcing graph topology into a tabular mold.

Consider an authorization graph for a local-first application: users, roles, permissions, resources connected by "has role," "grants permission," "accesses resource" edges. You need reachability queries: can user X access resource Y through any chain of role assignments? This is fundamentally a graph traversal problem that becomes increasingly expensive in relational storage as the authorization hierarchy deepens.

These workloads share common characteristics that existing embedded systems handle poorly:

- **Irregular access patterns**: Graph traversal does not follow predictable scan patterns like sequential table scans or index range queries. It branches unpredictably based on topology, accessing data in an order determined by graph structure rather than storage layout.
- **Hot nodes and skew**: Real-world graphs exhibit heavy-tailed degree distributions where a small number of nodes (supernodes) have orders of magnitude more edges than the average node [15]. A relational schema treats all rows equally, but a graph engine must handle supernodes specially to avoid cache thrashing and memory exhaustion during traversal.
- **Recursive exploration**: Many graph queries require exploring paths of variable length: find all nodes reachable within N hops, find shortest path between two nodes, detect cycles. Recursive CTEs in SQLite can express these queries but are slow because they materialize intermediate results as temporary tables rather than streaming through adjacency structures.
- **Property-rich entities**: Graph nodes and edges carry many properties (key-value pairs) that are queried independently of topology: find all users named "Alice" who live in "Boston," then traverse their connections. This requires efficient property indexing alongside topological access, which relational schemas handle through separate indexes but graph engines can optimize more tightly.

The gap is not theoretical. Applications building dependency analyzers, offline knowledge graphs, local recommendation engines, authorization systems, and network topology tools are forced to choose between slow relational queries, complex key-value encodings, or in-memory structures that lose data

on process exit. None of these options provides the operational simplicity of SQLite combined with the traversal performance that graph workloads demand.

Why SQLite Is Our North Star (And Where We Must Depart)

SQLite is the reference architecture for embedded databases because it solves the hard problems elegantly: ACID transactions in a single file, crash recovery through write-ahead logging, efficient indexing with B-trees, a full SQL query language and optimizer, all in less than a megabyte of compiled code [1]. Its design philosophy , "do one thing well, make it correct, keep it simple" , is exactly what an embedded graph database needs.

Graphite takes several architectural inspirations directly from SQLite:

- **Single-file storage**: Like SQLite, Graphite stores all data in a single file (plus a WAL file during active transactions). The file format is documented and stable so that tools can inspect or repair databases independently of the engine.
- **Page-based architecture**: Data is organized into fixed-size pages (default 4096 bytes) that are the atomic unit of I/O. Pages contain headers, metadata, and variable-length records packed efficiently using a slotted format.
- **B-tree indexes**: Graphite uses B+ trees for primary indexes on node IDs, edge IDs, and property values. B-trees provide logarithmic-time point lookups, efficient range scans, and proven crash recovery semantics [8].
- **Write-ahead logging**: Durability is guaranteed through WAL: changes are appended to a log file and flushed to disk before being applied to the main database file. On crash recovery, committed transactions are replayed and incomplete ones are rolled back.
- **Single-writer concurrency**: Like SQLite's default mode, Graphite serializes writers while allowing concurrent readers. This simplifies transaction management significantly compared to multi-writer designs while meeting the needs of most embedded workloads where writes are less frequent than reads [3].

However, graph workloads require departures from SQLite's relational architecture in several critical areas:

- **Storage layout**: SQLite stores table rows in B-trees keyed by rowid, which optimizes for point lookups and range scans but not for adjacency traversal. Graphite clusters edges physically near their source nodes so that expanding neighbors during traversal reads contiguous pages sequentially rather than following scattered pointers through index structures [6].
- **Query language**: SQL is designed for set operations on tables: select columns from rows satisfying predicates, joined across tables on key equality. Graph queries are fundamentally about pattern matching on topology: find subgraphs that match a given shape. Graphite uses a Cypher-inspired declarative syntax that expresses graph patterns visually rather than forcing them into tabular joins [5].
- **Traversal optimization**: SQLite's query optimizer plans join orders based on cardinality estimates and index availability but has no concept of graph traversal cost. Graphite's optimizer understands that expanding neighbors from a low-degree node is cheaper than from a high-degree supernode, that bidirectional search can reduce intermediate results, and that visited-set tracking prevents exponential blowup in cyclic graphs [12].
- **Index design**: SQLite indexes are uniform across all tables: B-trees on key columns. Graphite needs specialized adjacency indexes that accelerate neighbor expansion, property indexes optimized for multi-hop filtering, and potentially hybrid structures that balance read performance with update overhead for dynamic graphs [7].

The goal is not to copy SQLite but to apply its design principles , simplicity, correctness, embeddability, single-file portability , to the specific requirements of graph data. Where SQLite's architecture serves graph workloads well (page management, WAL, B-tree fundamentals), we adopt it. Where relational assumptions conflict with graph topology (storage layout, query model, traversal optimization), we depart deliberately and explain why.

Target Deployment Models and Realistic Use Cases

An embedded graph database is not a replacement for distributed graph systems in every scenario. Graphite targets specific deployment models where operational simplicity, low latency, and data locality matter more than horizontal scalability:

Desktop applications: Knowledge management tools, note-taking apps with linked concepts, offline-capable research assistants, desktop IDEs analyzing code dependencies. These applications run on a single machine, manage graphs of thousands to millions of nodes, and require instant startup with zero configuration. An embedded graph database eliminates the need to bundle and manage a separate server process.

Local-first applications: Applications that operate primarily offline with eventual sync to remote services (version control clients, collaborative editors, field data collection tools). Local-first architectures require all data operations to work without network connectivity, which means the graph engine must run locally without depending on a remote server. SQLite is common in local-first apps for relational data; Graphite fills the same role for graph data.

Command-line tools and build systems: Dependency analyzers, package managers, build system schedulers, static analysis tools that represent relationships as graphs and need to traverse them efficiently. These tools run transiently, open a database file, perform queries, and exit. They benefit from zero startup overhead and file-based portability.

Edge services and IoT devices: Network topology management on routers, device relationship graphs in industrial IoT controllers, local recommendation engines on edge servers with limited resources. Embedded databases are ideal here because they run within the application process without consuming additional system resources for a separate database daemon.

Testing and development: Local graph databases for integration testing, prototyping, and development environments where spinning up a full graph server is overkill. An embedded database starts instantly, requires no configuration, and can be discarded or version-controlled like any other file.

Graphite is not designed for:

- **Multi-machine clusters**: If your graph spans petabytes of data across hundreds of nodes, you need a distributed system with sharding, replication, and consensus protocols. Graphite operates within a single process on a single machine.
- **High-concurrency web backends**: If you need thousands of concurrent writers from multiple application servers, you need a client-server architecture with connection pooling and transaction coordination. Graphite's single-writer model is optimized for embedded workloads where writes are less frequent than reads.

- **Streaming analytics**: If you need to process continuous streams of graph updates in real time with sub-millisecond latency, specialized streaming graph processors may be more appropriate. Graphite focuses on persistent storage with transactional guarantees, not stream processing.

Understanding these boundaries is essential for making the right architectural choices. An embedded graph database is not universally better than a distributed one; it is better for specific scenarios where its strengths , simplicity, locality, zero operations , align with the application's requirements.

Design Goals: Correctness, Durability, Locality, Small Footprint

Every design decision in Graphite is evaluated against five core goals, listed in order of priority:

1. Correctness: The database must never corrupt data under normal operation or after crashes. Transactions that commit are durable; transactions that abort leave no trace. Indexes remain consistent with stored data. Queries return results that match the current transactional state. This goal is non-negotiable and takes precedence over all performance optimizations. We verify correctness through formal invariants, extensive testing, property-based tests, fuzzing, and deterministic crash recovery tests.

2. Durability: Committed data survives process crashes, operating system reboots, and power failures (within the limits of underlying hardware). Graphite achieves this through write-ahead logging with explicit fsync ordering: log records are flushed to stable storage before modified database pages are written. We document exactly what durability guarantees hold under which failure scenarios and do not overclaim , consumer SSDs without power-loss protection cannot guarantee durability against sudden power loss regardless of software design [3].

3. Traversal locality: Graph traversal performance is the primary differentiator between a graph engine and a general-purpose database with graph-like schemas. Graphite's storage layout clusters adjacent data physically so that expanding neighbors during traversal maximizes sequential I/O, minimizes random page accesses, and exploits CPU cache behavior. This goal drives the choice of node-centric adjacency clustering over alternative layouts [6].

4. Small operational footprint: The engine must be simple to deploy, configure, and maintain. Single-file storage, zero configuration, no external dependencies, instant startup. The compiled library should be small enough to bundle with applications without significant size penalty. Operational simplicity is a feature, not an afterthought , it is what distinguishes embedded from client-server databases.

5. Predictable performance: Query latency and throughput should be consistent and understandable, not subject to unpredictable spikes from garbage collection pauses, background compaction storms, or cache invalidation cascades. Graphite avoids runtime systems with non-deterministic behavior (garbage collectors, just-in-time compilers) and documents performance characteristics for common operations so engineers can reason about their application's behavior.

These goals create tensions that require explicit trade-offs:

- Correctness vs. performance: Additional validation checks, checksums, and invariants slow down operations slightly but prevent subtle corruption bugs. We prioritize correctness.
- Durability vs. throughput: Frequent fsync calls guarantee durability but reduce write throughput. Graphite provides configurable durability modes (full fsync on every commit for maximum safety, batched fsync for higher throughput with acceptable risk) and documents the trade-offs explicitly.
- Locality vs. update cost: Clustering edges near source nodes optimizes reads but makes edge insertions and deletions more expensive due to page splits and record relocation. For embedded workloads where reads typically exceed writes by orders of magnitude, we optimize for traversal locality.
- Simplicity vs. feature completeness: A minimal query language is easier to implement correctly and optimize than a full SQL dialect. Graphite implements a focused subset of graph query patterns that covers the vast majority of practical use cases without the complexity of edge cases.

Non-Goals: Distributed Consensus, ML Integration, Cloud-Native Abstractions

Explicitly stating what Graphite does not do is as important as defining what it does. These non-goals prevent scope creep and keep the design focused:

No distributed consensus: Graphite does not implement Raft, Paxos, or any consensus protocol for multi-node replication. It operates within a single process on a single machine. If you need replicated graphs across data centers, use a distributed system designed for that purpose. Attempting to add consensus to an embedded engine fundamentally changes its architecture and operational model.

No machine learning integration: Graphite does not include built-in graph neural network training, embedding computation, or ML model serving. These are application-layer concerns that can use Graphite as a data source. Embedding ML infrastructure into the database engine would add massive complexity for a narrow set of use cases.

No cloud-native abstractions: Graphite does not provide Kubernetes operators, Helm charts, managed service APIs, or auto-scaling infrastructure. It is an embedded library, not a cloud product. Applications deploying Graphite in containerized environments manage it as part of their application state, not as a separate cloud-managed resource.

No multi-model storage: Graphite stores property graphs and nothing else. It does not support document storage, time-series data, wide-column families, or full-text search as first-class features. Multi-model databases often sacrifice depth in each model for breadth across models. Graphite focuses on doing graph storage exceptionally well within the embedded paradigm.

No interactive analytics at petabyte scale: While Graphite supports analytical queries like shortest path and connected components, it is not designed for batch processing of graphs with billions of nodes and edges that exceed available RAM. For large-scale analytics, systems like Apache Giraph, GraphX, or specialized graph processors are more appropriate. Graphite targets graphs that fit comfortably on a single machine's storage, with working sets that can be cached in RAM.

These non-goals are not permanent limitations but deliberate boundaries for the reference implementation. Future extensions could add capabilities in

any of these areas, but the core design is optimized for the embedded graph database thesis: a simple, correct, fast, embeddable engine for property graphs that runs wherever your application runs.

Chapter 2: Graph Foundations for Storage Engineers

Vertices, Edges, Directions, and the Property Graph Model

A graph, in its mathematical essence, is a pair G = (V, E) where V is a set of vertices and E is a set of edges connecting pairs of vertices. This definition is so minimal that it tells us almost nothing about how to store graphs in practice. Real-world graph databases operate on richer models that add structure: directions, labels, types, properties, multigraph semantics, and constraints. Each addition has concrete implications for storage layout, indexing strategy, and query processing.

Graphite uses the property graph model, which is the dominant data model in production graph systems and is standardized as part of ISO/IEC 39075 GQL [4]. The property graph model extends the mathematical definition with four key concepts:

- **Vertices (nodes)**: Discrete entities in the graph. Each vertex has a unique identifier, zero or more labels (categorical tags), and zero or more properties (key-value pairs).
- **Edges (relationships)**: Directed connections between vertices. Each edge has a unique identifier, a type (relationship kind), a source vertex, a target vertex, and zero or more properties.
- **Labels**: Categorical identifiers attached to vertices that classify them into types: Person, Organization, City, Module, User. Labels enable schema-like organization without rigid enforcement.
- **Properties**: Key-value pairs attached to vertices or edges that store attributes: name = “Alice”, age = 30, created_at = 2024-01-15T10:30:00Z. Property keys are strings; property values have typed representations (integer, float, string, boolean, null).

This model is intentionally flexible compared to the relational model. In a relational database, you define tables with fixed schemas: every row in the Person table has the same columns, and each column has a declared type. In a property graph, two vertices with the label Person can have completely different properties: one might have name, email, and age while another has only name and department. This flexibility is both a strength and a challenge for storage design.

From a storage perspective, the property graph model requires us to solve several problems that relational models handle more mechanically:

- **Variable schemas**: Since vertices with the same label can have different properties, we cannot use fixed-width row layouts. Each vertex record must encode which properties it has and their values, requiring variable-length encoding schemes.
- **Property indexing**: Queries filter on property values ("find all users where age > 25"), so we need indexes on property keys and values. But unlike relational databases where every row has every column, property indexes must handle sparse data where many vertices lack a given property.
- **Label-based queries**: Queries often start from or filter by labels ("MATCH (p:Person) WHERE ..."), requiring efficient label-to-vertex mapping that supports both point lookups and enumeration.
- **Edge directionality**: Edges are directed, meaning traversal can follow outgoing edges from a source, incoming edges to a target, or both. Storage must support efficient lookup in both directions without duplicating all edge data.

The property graph model is not the only option. The RDF (Resource Description Framework) model, used by triple stores like Apache Jena and Blazegraph, represents data as subject-predicate-object triples where all three components are URIs or literals. RDF is more uniform and mathematically elegant but less natural for applications that think in terms of entities with attributes rather than atomic facts. Graphite chooses the property graph model because it aligns better with how most application developers reason about connected data and matches the mental model of existing graph query languages like Cypher [5].

Labels, Types, Properties: And Why They Matter for Indexing

Labels and types are not merely cosmetic metadata; they fundamentally shape how we organize storage and indexes. Consider a graph with millions of vertices representing users, products, and orders. A query that says "find all users who ordered a product" needs to quickly identify which vertices are users versus products versus orders without scanning the entire vertex set.

Graphite implements labels as an integer-mapped string system. When a label like "Person" is first used, it is assigned a small integer ID (e.g., 0). The mapping from string names to integer IDs is stored in a system catalog. Vertex records store their labels as compact integers rather than full strings, reducing storage overhead and enabling fast comparison operations.

This design has important implications:

- **Storage efficiency**: A vertex with three labels stores three bytes (using varint encoding) instead of potentially dozens of bytes for null-terminated strings. For graphs with millions of vertices, this saves megabytes of storage.
- **Index structure**: Label-based indexes can use integer keys directly without string comparison overhead. An index mapping label ID to vertex IDs is a simple B+ tree on integers, which packs more entries per page than string-keyed indexes.
- **Query optimization**: The query planner can use label statistics (how many vertices have each label) for cardinality estimation. "MATCH (p:Person)" has a very different cost estimate than "MATCH (x)" because the former is constrained to a known subset of vertices.

Edge types work similarly. Each relationship type ("KNOWS", "WORKS_AT", "ORDERS") is assigned an integer ID and stored compactly in edge records. This enables efficient filtering during traversal: when expanding neighbors with a specific relationship type, the engine can compare integers rather than strings.

Properties are more complex because they combine keys (strings) with values (typed data). Graphite stores properties as variable-length key-value pairs packed into vertex or edge records. For small property sets that fit within a page, properties are stored inline with the vertex/edge record. For vertices

or edges with many or large properties, overflow pages hold the excess data, referenced by a pointer in the main record.

Property storage design involves several trade-offs:

- **Inline vs. separate**: Storing properties inline with vertex records minimizes I/O for property access (the vertex and its properties are on the same page) but makes updates more expensive (changing a property may require rewriting the entire record). We choose inline storage as the default because read performance matters more than write flexibility for most embedded workloads.
- **Typed vs. untyped**: Storing property values with explicit type tags (like SQLite's serial types [1]) enables correct comparison semantics and efficient serialization but adds a byte or two per value. Graphite uses typed storage because correctness in comparisons and indexing is worth the small overhead.
- **Normalized vs. denormalized keys**: Property keys could be stored as full strings in each record or as references to a centralized key catalog. Full strings waste space when the same property name appears on many vertices; key IDs save space but add an indirection layer. Graphite uses a hybrid approach: commonly used property keys are assigned IDs, while rare keys are stored inline to avoid catalog bloat.

Multigraphs, Self-Loops, and Duplicate Semantics

A simple graph allows at most one edge between any pair of vertices in a given direction. A multigraph allows multiple edges between the same pair, potentially with different types or properties. Real-world applications frequently require multigraph semantics: two people might have multiple "COLLABORATED_ON" relationships (one per project), a user might have multiple "PURCHASED" edges to the same product (one per order).

Graphite supports multigraphs by design. Each edge has a unique identifier independent of its source and target vertices, so multiple edges between the same pair are naturally supported. This decision affects storage and indexing in several ways:

- **Edge identification**: Edges cannot be identified solely by (source_id, target_id) pairs because that would not distinguish parallel edges. Graphite

assigns each edge a unique 64-bit ID at creation time, stored explicitly in the edge record.

- **Adjacency representation**: When storing outgoing edges for a vertex, we cannot use a simple set of target IDs. Instead, adjacency lists store full edge references (edge_id or target_id plus type), enabling multiple entries for the same target.
- **Duplicate detection**: Queries that ask "is there an edge from A to B of type T?" must handle the case where multiple such edges exist. Graphite's query semantics return all matching edges by default; applications can use LIMIT 1 or uniqueness predicates if they need exactly one.

Self-loops are edges where source and target are the same vertex. They model reflexive relationships: a user follows themselves, a module depends on itself (directly or through circular dependencies), a location contains itself in hierarchical spatial graphs. Self-loops are valid in Graphite and stored identically to other edges , the storage format does not need special handling because source_id equals target_id is just another value combination.

The semantics of duplicate edges require careful definition. Consider these scenarios:

- **Creating an edge when one already exists**: In a simple graph, this would be a no-op or error. In Graphite's multigraph model, creating a second edge between the same vertices with the same type is allowed and creates a distinct edge with its own ID and properties. Applications that need uniqueness must enforce it explicitly through constraints (discussed in Chapter 9).
- **Deleting an edge**: Deleting an edge by its ID removes exactly one edge. Deleting "all edges from A to B of type T" removes all matching parallel edges. The semantics are explicit rather than implicit.
- **Querying edges**: Pattern matches like "(a)-[:KNOWS]->(b)" match all KNOWS edges between a and b, not just one. This is important for aggregation queries: "count the number of collaborations between two researchers" requires counting parallel edges.

These semantics align with how most graph applications reason about relationships but differ from relational foreign keys where duplicate rows are typically prevented by primary key constraints. Engineers porting relational mental models to graphs must adjust their expectations about edge uniqueness.

Paths, Walks, Cycles, Neighborhoods, and Reachability

Graph traversal terminology matters because different concepts have different algorithmic and storage requirements. Confusing "path" with "walk" or "reachability" with "shortest path" leads to incorrect implementations and inefficient queries.

A **walk** is a sequence of vertices and edges where each edge connects consecutive vertices, with no restriction on repetition. Walks can revisit vertices and edges arbitrarily. A **path** is a walk with no repeated vertices , it visits each vertex at most once. A **trail** is a walk with no repeated edges (vertices may repeat). These distinctions matter for query semantics: "find all paths from A to B within 5 hops" must avoid cycles, while "enumerate all walks of length 3 from A" allows revisiting nodes.

A **cycle** is a path that starts and ends at the same vertex (with at least one edge). Cycles are common in real-world graphs: dependency graphs with circular dependencies, social networks where friendship is mutual, authorization graphs with role hierarchies that loop. Cycle detection is an important operation for validating graph integrity (e.g., ensuring a build dependency graph is acyclic) and for preventing infinite traversal in queries.

A vertex's **neighborhood** is the set of vertices directly connected to it by edges. The **out-neighborhood** includes only targets of outgoing edges; the **in-neighborhood** includes only sources of incoming edges. Neighborhood expansion , retrieving all neighbors of a given vertex , is the atomic operation of graph traversal and the most performance-critical operation in a graph database. Storage layout decisions are primarily driven by making neighborhood expansion fast [6].

Reachability asks whether there exists any path from vertex A to vertex B, regardless of length. Reachability queries are fundamental for authorization ("can user X access resource Y?"), dependency analysis ("does module A transitively depend on module B?"), and connectivity analysis ("are these two components in the same connected subsystem?"). Reachability is computationally expensive for large graphs because it may require exploring many vertices before finding a path or proving none exists. Graphite supports bounded-depth reachability (limiting maximum hops) to prevent unbounded exploration on large graphs.

From a storage perspective, these concepts drive specific design require-

ments:

- **Efficient neighbor expansion**: The adjacency structure must allow retrieving all neighbors of a vertex with minimal I/O. This is the primary motivation for node-centric storage layouts where edges are clustered near their source vertices [6].
- **Visited set support**: Traversal algorithms that explore paths or detect cycles need to track which vertices have been visited. For traversals over graphs larger than RAM, the visited set itself can become a bottleneck. Graphite provides traversal primitives that integrate visited-set tracking with storage access patterns.
- **Bidirectional traversal**: Some queries are more efficient when traversing from both endpoints simultaneously (bidirectional BFS for shortest path). Storage must support efficient lookup of both outgoing and incoming edges without prohibitive overhead.
- **Early termination**: Traversal queries often can stop as soon as a result is found (e.g., "is there any path from A to B?"). The execution engine must support streaming results and early termination rather than materializing all possible paths before returning.

Degree Distributions and the Supernode Problem

Real-world graphs do not have uniform degree distributions. In most networks , social graphs, web link graphs, citation networks, dependency graphs , the distribution of vertex degrees follows a heavy-tailed pattern where most vertices have few connections but a small number of vertices (called hubs or supernodes) have orders of magnitude more connections than average [15].

This skew has profound implications for storage and query performance:

- **Cache behavior**: Expanding neighbors from a typical low-degree vertex fits in a single cache line or page. Expanding neighbors from a supernode with millions of edges requires reading megabytes of data, thrashing CPU caches and potentially spilling to disk.
- **Memory usage**: Traversal algorithms that maintain frontier sets or visited sets can consume excessive memory when exploring from supernodes. A BFS starting from a supernode might enqueue millions of neighbors in the first iteration.

- **Query planning**: A join order that expands from a supernode first can generate enormous intermediate results, while expanding from low-degree vertices first keeps intermediate sets small. The query optimizer must understand degree distributions to choose efficient traversal orders [12].
- **Index design**: Indexes on edges incident to supernodes become hot spots for both reads and writes, potentially becoming bottlenecks under concurrent access.

Graphite addresses the supernode problem through several mechanisms:

- **Degree statistics**: The catalog tracks approximate degree counts per vertex (or at minimum per label), enabling the query planner to estimate traversal costs and avoid expanding from high-degree vertices when alternatives exist.
- **Bounded expansion**: Traversal operations support degree-based limits to prevent a single supernode from consuming all available resources. Applications can configure maximum neighbors expanded per step.
- **Streaming results**: Rather than materializing all neighbors of a supernode in memory, traversal operators stream neighbors one at a time, applying filters and predicates during iteration to avoid building large intermediate collections.
- **Adjacency pagination**: For vertices with extremely high degrees, adjacency lists are split across multiple pages that can be read incrementally rather than loading the entire adjacency list at once.

The supernode problem is not unique to graph databases , it appears in any system dealing with skewed data distributions , but it is particularly acute for graphs because traversal performance depends directly on degree. A storage engine that treats all vertices uniformly will perform poorly on realistic graphs where a few vertices dominate the edge count. Understanding and planning for skew is essential for building a production-quality graph database.

Graph Schemas: Enforced vs. Optional vs. Hybrid

The property graph model is traditionally schema-optional: you can create vertices and edges with arbitrary labels and properties without declaring a

schema upfront. This flexibility accelerates development and adaptation but creates challenges for correctness, performance, and tooling.

A fully enforced schema would require declaring all allowed labels, edge types, and property definitions before inserting data. Every vertex and edge must conform to its declared type: required properties must be present, property types must match, constraints must be satisfied. This provides strong guarantees but reduces flexibility and adds overhead for validation on every write.

Graphite adopts a hybrid approach that balances flexibility with structure:

- **Labels and types are open**: New labels and edge types can be created dynamically without schema migration. When a vertex is created with a label that has not been seen before, Graphite registers it in the system catalog automatically.
- **Property constraints are optional but supported**: Applications can define constraints on specific labels: "all vertices with label Person must have a non-null name property of type string," "the email property on Person vertices must be unique across all Person vertices." These constraints are enforced at write time and indexed for efficient checking.
- **Schema introspection is always available**: The catalog provides metadata about all labels, types, properties, and constraints in the graph, enabling tooling for schema documentation, migration planning, and query optimization statistics.

This hybrid model aligns with how most graph applications evolve: they start with flexible schemas during development and gradually add constraints as requirements stabilize. It also matches the GQL standard's approach to property graph schema, which supports both unconstrained graphs and administrator-defined graph types with enforced structure [4].

From a storage perspective, optional schemas mean that vertex and edge records must be self-describing: each record encodes which properties it has so that readers can interpret the data without external schema information. This requires variable-length record formats with property headers, as discussed in Chapter 5. Optional schemas also mean that indexes are created dynamically (when constraints or explicit index definitions are added) rather than being fixed at database creation time, requiring online index building and maintenance capabilities.

Chapter 3: Database Internals from First Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Pages, Blocks, Records, and Slots: The Vocabulary of Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Fixed vs. Variable-Length Records and Slotted Page Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

File Formats: Headers, Metadata, and Self-Description

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

The Memory Stack: Application Heap, OS Cache, Database Buffer Pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Disk, SSD, NVMe Characteristics and I/O Patterns That Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Checksums, Corruption Detection, and the Cost of Trust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 4: Architecting Graph Storage on Disk

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Adjacency Lists vs. Matrices vs. Edge Tables: The Fundamental Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Node-Centric Layouts: Clustering Relationships Near Their Source

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

CSR/CSC-Inspired Structures for Traversal Locality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Pointer Chasing, Offsets, and the Cost of Indirection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Update Patterns: Appends, Deletes, Fragmentation, and Compaction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

The Reference Architecture: Why We Choose What We Choose

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 5: The Binary Database File Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Magic Values, Version Numbers, and Byte Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Page Size Selection and Global Header Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Node Records: IDs, Labels, Property Pointers, Degree Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Edge Records: Source/Target References, Types, Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Variable-Length Properties and Overflow Page Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Free-Page Tracking, Checksums, and Corruption Invariants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 6: Identifiers, References, and Relocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Physical Record IDs vs. Logical Stable Identifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Page/Slot Addressing and Generation Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Deleted Object Reuse Without Tombstone Explosion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Stale Reference Detection and Safe Indirection Layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Relocation During Compaction and Vacuum Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Trade-off Analysis: Compactness vs. Durability vs. Simplicity

Chapter 7: Memory Management and Page Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Page Allocation, Free Lists, and Persistent Freelist Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Buffer Pool Architecture: Pinning, Dirty Pages, Eviction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

LRU vs. Clock Algorithms: Measured, Not Assumed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Memory Mapping: When mmap Helps and When It Hurts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Prefetching Strategies for Traversal Locality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Bounded Memory Usage and Graceful Degradation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 8: Indexing for Graph Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

B+ Trees from Scratch: Pages, Splits, Merges, Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Hash Indexes for Equality Lookups and Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Property Indexes: Single-Key, Composite, and Inverted Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Adjacency Indexes: Accelerating Neighbor Expansion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Index Maintenance Under Transactional Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Statistics, Selectivity, and Query Planner Integration

Chapter 9: Core Graph Operations and Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Node Creation, Reading, Update, and Deletion Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Edge Creation with Duplicate and Self-Loop Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Property Operations: Sets, Nulls, Missing vs. Explicit Absence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Cascading Deletes, Dangling Edges, and Referential Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Schema Constraints: Uniqueness, Type Enforcement, Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Transactional Visibility of Graph Mutations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 10: Transactions from First Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

ACID in the Embedded Context: What It Actually Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Transaction States: Active, Prepared, Committed, Aborted

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Lock-Based Concurrency: Granularity, Deadlocks, Starvation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Optimistic Concurrency and Timestamp-Based MVCC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Read Sets, Write Sets, Conflict Detection at Commit Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

The Reference Choice: Why We Implement What We Implement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 11: Durability, Logging, and Crash Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Write-Ahead Logging vs. Rollback Journals vs. Copy-on-Write

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

WAL Design: Log Records, Transaction Boundaries, Group Commit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

fsync Semantics, Durability Primitives, and OS Buffering Hazards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Checkpoints: Materializing WAL State to the Main Database File

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Recovery Procedure: Redo of Committed, Undo of Uncommitted

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Crash Testing Infrastructure: Fault Injection and Invariant Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 12: Graph Traversal Primitives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Neighbor Expansion: The Atomic Unit of Graph Traversal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Breadth-First Search with Frontier Management and Visited Sets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Depth-First Search, Cycle Handling, and Path Reconstruction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Directional Filtering, Type Filtering, Property Predicates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Memory-Efficient Traversal Over Graphs Larger Than RAM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Distinguishing Traversal Primitives from Analytical Algorithms

Chapter 13: Query Language and Frontend

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Language Design Goals: Expressiveness vs. Simplicity vs. Familiarity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Lexical Structure: Tokens, Keywords, Literals, Identifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Grammar: Patterns, Predicates, Projections, Traversals, Mutations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Parser Implementation: Recursive Descent with Error Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Semantic Analysis: Name Resolution, Type Checking, Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Logical Query Representation and Plan Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 14: Query Planning, Optimization, and Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Logical vs. Physical Plans: Operators and Their Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Cardinality and Selectivity Estimation from Statistics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Index Selection: Scan vs. Seek Decisions for Graph Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Traversal Order Planning: Avoiding Intermediate Result Explosion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Rule-Based and Cost-Based Optimization Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Iterator Execution Engine: Volcano Model, Memory, Spilling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Chapter 15: APIs, Bindings, Tools, and Production Readiness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Embedded API Design: Lifecycle, Transactions, Prepared Statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Rust Implementation and Python Bindings via FFI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Database Shell and CLI: Querying, Inspection, Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Security: Untrusted Files, Parser Attacks, Resource Exhaustion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Error Model, Observability, and Tuning Guidance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Backup, Restore, Integrity Checks, and Production Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

Conclusion: The Cohesive Engine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/embeddedgraphdatabases.