**A SYNOPSIS ON**



**DaemonDB - A Relational Database Engine Built In Go**



**Submitted in partial fulfilment of the requirement for the award of the degree of**

**BACHELOR OF TECHNOLOGY**

**IN**

**COMPUTER SCIENCE & ENGINEERING**

**Submitted by:**

**Student Name 1 : Shubham Negi** **University Roll No. : 2021460**

**Student Name 2 : Shiwang Bisht** **University Roll No. : 2021448**

**Student Name 2 : Utkarsh Verma** **University Roll No. : 2021510**

***Under the Guidance of***

**Mr. Navin Garg**

**Associate Professor**

**Project Team ID: MP2025CSE178**

A logo of a graphic era

AI-generated content may be incorrect.

**Department of Computer Science and Engineering**

**Graphic Era (Deemed to be University)**

**Dehradun, Uttarakhand**

**September-2025**

A white background with red and black text

AI-generated content may be incorrect.

**CANDIDATE’S DECLARATION**

I/We hereby certify that the work which is being presented in the Synopsis entitled **“DaemonDB – A Relational Database Engine Built In Go”** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science and Engineering in the Department of Computer Science and Engineering of the Graphic Era (Deemed to be University), Dehradun shall be carried out by the undersigned under the supervision of **Mr. Navin Garg, Associate Professor**, Department of Computer Science and Engineering, Graphic Era (Deemed to be University), Dehradun.

Shubham Negi   2021460 signature

Shiwang Bisht 2021448 signature

Utkarsh Verma 2021510 signature

The above mentioned students shall be working under the supervision of the undersigned on the **“DaemonDB – A Relational Database Engine Built In Go”**

         Signature Signature

**Supervisor** **Head of the Department**

**Internal Evaluation (By DPRC Committee)**

**Status of the Synopsis:**  Accepted / Rejected

**Any Comments:**

**Name of the Committee Members: Signature with Date**

1.

2.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Description** | **Page No.** |
| Chapter 1 | Introduction and Problem Statement |  |
| Chapter 2 | Background/ Literature Survey |  |
| Chapter 3 | Objectives |  |
| Chapter 4 | Hardware and Software Requirements |  |
| Chapter 5 | Possible Approach/ Algorithms |  |
|  | References |  |

**Chapter 1**

**Introduction and Problem Statement**

* 1. Motivation

Relational databases are ubiquitous. Many production RDBMSes implement an internal key‑value layer and build relational abstractions on top of it. Building a simplified relational engine from scratch is an excellent learning project and gives hands‑on exposure to storage formats, indexing, transactions, concurrency, and recovery.

* 1. Problem Statement

Create a functional, modular relational database engine implemented in Go that supports:

* Basic DDL/DML (CREATE TABLE, INSERT, SELECT, simple WHERE, DELETE, UPDATE),
* A SQL tokenizer & parser with a simple optimizer,
* Durable storage using WAL and page files,
* Efficient indexing (clustered and secondary indices) using B+‑trees,
* A transaction subsystem that preserves Atomicity and Durability (initially using strict 2PL and WAL; MVCC is a possible later enhancement),
* A recovery mechanism for crash consistency.

Constraints: implement primarily as a single‑node server (network layer optional). Keep the system lightweight but architected for extension (replication/distribution later).

* 1. Scope and Limitations

Initial scope focuses on core correctness and durability, not extreme performance. Advanced features (SQL standard completeness, full optimizer, distributed consensus, PBFT, complex query plans) are out of scope for the first milestone but will be planned as later modules.

**Chapter 2**

**Background/ Literature Survey**

* 1. Databases on top of Key‑Value Stores

Many modern RDBMS designs conceptually treat storage as key‑value: the row primary key maps to a value that contains the rest of the row. This simplifies the storage engine: keys become the primary way to find rows, while relational features are layered above. The project will adopt this approach: use a simple on‑disk key→value mapping as the base file format with page organization.

**Useful concepts:** logical record locators (page\_id, slot), page headers, fixed page sizes (e.g., 8KB), and offset arrays for variable‑length rows.

* 1. B‑tree vs. B+‑tree vs. LSM‑tree

**Sorted arrays / Hashing:** simple but limited; hashing has collision issues and poor range query support.

**B‑tree / B+‑tree:** industry standard for OLTP; B+‑tree stores keys in internal nodes and full records at leaves (clustered) or references to records (non‑clustered). Good for point/range queries and balanced on updates.

**LSM‑tree:** write‑optimized, good for high write throughput; good for append‑heavy workloads and used by many KV stores.

Design decision: implement a persistent **B+‑tree** first (balanced, good for low latency OLTP). LSM concepts can be explored later if write amplification or high ingest performance becomes a requirement.

* 1. Storage Page Layout and Persistence

Data files are organized as fixed‑size pages (commonly 4KB or 8KB). Each page includes:

* Header (file id, page id, checksum, free space map)
* Row directory / slot array (for variable length rows)
* Payload (row bytes)

Logical record locators (page, slot) are used by non‑clustered indexes; clustered indexes store rows at leaf nodes directly.

* 1. Write‑Ahead Logging (WAL) and Durability

Durability is achieved via a WAL: before mutating on‑disk pages, write intent records to an append‑only log and flush. After durable log write, apply to data pages; checkpoints periodically flush in‑memory state and advance durable state. Additionally, using checksums in page headers helps detect corruption and aid recovery.

* 1. Transactions, Concurrency and Recovery

**Atomicity & Durability:** WAL + atomic commit (fsync semantics) give strong guarantees.

**Concurrency control:** initial implementation can use strict two‑phase locking (S2PL). MVCC is a later enhancement to improve read concurrency.

**Recovery manager:** on startup, use WAL to redo committed transactions and rollback incomplete ones.

* 1. Query Processing Pipeline

Typical pipeline: Network client → Tokenizer → Parser → Logical plan → Optimizer → Physical plan → Executor. We will implement a small but extensible SQL parser and a planner/executor that supports table scans, index lookups, and simple joins (nested loop or hash joins later).

* 1. Replication, Pub/Sub and Distribution (Future Work)

 **LISTEN/NOTIFY:** a pub/sub primitive to notify sessions on changes; lightweight implementation over a message queue.

 **Replication & Consensus:** Master‑Slave replication first; distributed consensus via Raft for leader election and log replication. Research PBFT as an optional adaptive mode for Byzantine environments.

* 1. Key References and Resources

 Build‑Your‑Own Database tutorials and book (link provided by project lead)

 Designing Data‑Intensive Applications — Martin Kleppmann (recommended)

 Standard DB textbooks (Silberschatz, Korth, Sudarshan etc.)

 PostgreSQL and InnoDB internals (for clustered vs non‑clustered index behavior)

**Chapter 3**

**Objectives**

* 1. Primary Objectives

Implement a persistent storage engine in Go with WAL and page files.

Implement a durable B+‑tree index (clustered primary index and secondary index support).

Build a SQL tokenizer and parser to execute basic SQL statements (DDL + DML + simple SELECT with WHERE).

Implement a transaction manager that provides Atomicity and Durability (initially using WAL + strict 2PL).

Implement recovery on restart using WAL checkpoints and checksums.

* 1. Secondary Objectives

 Implement simple query optimizer heuristics and an execution engine supporting index lookups and table scans.

 Provide hooks for pub/sub notifications and a basic LISTEN/NOTIFY mechanism.

 Design modular components to allow adding replication (Master‑Slave) and distributed consensus (Raft) later.

* 1. Evaluation Objectives

 Correctness (ACID properties for single‑node workloads)

 Functional completeness for a core subset of SQL

 Performance baselines vs naive file scan (index vs no index)

 Crash recovery correctness (simulated crashes and WAL redo)

**Chapter 4**

**Possible Approach/ Algorithms**

* 1. **High Level Architecture**

Components (modular):

* **Frontend:** simple CLI or TCP server (optional) that accepts SQL.
* **SQL Layer:** tokenizer → parser → logical plan → simple optimizer → physical plan.
* **Execution Engine:** operators for scans, index seeks, projection, filters, and simple joins.
* **Transaction Manager:** manages transaction lifecycle, locks (Lock Manager), and ensures WAL ordering.
* **Storage Engine:** Disk Manager (files & pages), Buffer Manager (in‑memory page cache), Index Manager (B+‑tree), and WAL manager.
* **Recovery Manager:** checkpointing and WAL replay on startup.
  1. Storage Format and Page Management

 Fixed page size (e.g., 8KB).

 Page header: file\_id, page\_no, lsn (log seq no), checksum, free\_space.

 Row directory with slot entries pointing to variable length tuples.

 Pages mapped to persistent files in a deterministic way (so logical page numbers are stable across restarts).

* 1. Write‑Ahead Log (WAL)

 Append‑only WAL file(s) with WAL records describing intent (INSERT/UPDATE/DELETE and before/after images or undo/redo info).

 WAL records have a LSN, transaction id, and checksum.

 Commit requires WAL flush to durable storage. After commit, changes may be applied to pages and checkpointed later.

* 1. B+ Tree Design

 Persistent B+‑tree with fixed‑size nodes stored on pages.

 Node layout: header + array of keys + array of child pointers for internal nodes / array of (key, record\_locator) for leaf nodes (or full row for clustered index).

 Splits, merges and parent updates will be logged for durability or replayable via WAL.

 Consider adopting copy‑on‑write for tree modifications to simplify crash consistency (optional).

* 1. Index Types

 **Clustered primary index:** leaf nodes contain the full row or a direct mapping to the row in the clustered file.

 **Secondary (non‑clustered) index:** leaf nodes store (key, record\_locator) → (page, slot) for lookup back to the clustered table.

* 1. Transaction & Concurrency

 Start with strict two‑phase locking (S2PL). Lock manager provides shared/exclusive locks on row or page granularity.

 Later: evaluate MVCC (versioned records and snapshot isolation) for better concurrency.

* 1. Query Processor and Optimizer

Minimal AST from parser.

Logical plan generation: table scan, index scan, filter pushdown.

Simple rule‑based optimizer: prefer index scan when predicate matches an indexed column; otherwise do table scan.

Execution operators: TableScan, IndexScan, Filter, Projection, NestedLoopJoin.

* 1. Recovery Strategy

On startup: read last checkpoint, apply WAL forward to bring pages to most recent committed state (redo). Optionally perform undo for in‑flight transactions.

Checkpointing: periodically write dirty pages and record checkpoint LSN.

Use checksums to validate pages and WAL records.

* 1. Replication & Pub/Sub (Planned)

Replication:simple asynchronous master→slave replication via WAL shipping. For stronger guarantees, integrate Raft to replicate WAL and elect leader.

LISTEN/NOTIFY:maintain in‑server channels where transactions can publish notifications after commit; clients subscribed to channel receive notices.

* 1. Testing and Benchmarks

Unit tests for parser, B+‑tree operations (search, insert, delete, split/merge), WAL write/read, and recovery scenarios.

Integration tests: crash recovery (kill process during heavy writes and restart), correctness tests.

Microbenchmarks: point lookup latency (with/without index), insert throughput, range scan performance.

Optional: implement a TPC‑like simple workload generator.

* 1. Deliverables and Milestones

M1 (Core storage) — Page files, WAL, simple table IO, basic CLI.

M2 (Indexing) — Persistent B+‑tree implementation, primary index.

M3 (SQL) — Tokenizer, parser, basic DDL/DML, execution engine.

M4 (Transactions & Recovery) — WAL commit semantics, strict 2PL, recovery tests.

M5 (Extras) — Secondary indexes, LISTEN/NOTIFY, replication prototype.

* 1. Risk Analysis and Mitigation

Data corruption due to crashes: mitigate with WAL + checksums + thorough recovery testing.

Concurrency bugs and deadlocks: design deadlock detection and provide test harnesses to reproduce situations.

Performance pitfalls: start with correct simple implementation; then profile and optimize hotspots (buffer manager, serialization, tree splits).

**References**

[1] N. K. Kanhere and S. T. Birchfied, “Real-time incremental segmentation and tracking of vehicles at low camera angles using stable features,” *IEEE Trans. Intell. Transp. Syst*., vol. 9, no. 1, pp.148-160, March 2008 **(Example : Journal papers)**