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 3 : Utkarsh Verma	University Roll No. : 2021510

Under the Guidance of

Mr. Navin Garg
Associate Professor
Project Team ID: MP2025CSE178

Department of Computer Science and Engineering
Graphic Era (Deemed to be University)
Dehradun, Uttarakhand

September-2025

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
Supervisor

Signature
Head of the Department

<u>Internal Evaluation (By DPRC Committee)</u>

Status of the Synopsis: Accepted / Rejected
Any Comments:

Name of the Committee Members:	**Signature with Date**
1. Dr. Santosh Kumar	
2. Dr. Ankit Tomar	
3. Mr. Sachin Bhatt	

Table of Contents

Chapter 1

Introduction and Problem Statement

1.1 Introduction

Relational databases serve as the foundational infrastructure of modern computing systems, powering everything from lightweight web applications to enterprise-scale data platforms. These systems have become indispensable in managing the vast amounts of structured data that drive today's digital economy.

This project presents the development of DaemonDB, a complete relational database engine built entirely from scratch using Go. Unlike implementations that rely on existing database libraries or frameworks, DaemonDB implements every core component independently—from low-level storage management to high-level query processing—providing comprehensive insight into database system internals.

The development of DaemonDB offers extensive hands-on experience with fundamental computer science concepts including B+ tree data structures, buffer pool management, query parsing and optimization, ACID transaction processing, concurrency control mechanisms, and crash recovery systems. This ground-up approach ensures a thorough understanding of how modern database engines operate under the hood.

DaemonDB represents both an educational exploration of database internals and a practical implementation of a functional database system. The project demonstrates the complexity and elegance of relational database design while producing a working system capable of handling real-world database operations with performance and reliability considerations built into its architecture.

1.2 Problem Statement

The objective of this project is to design and implement a lightweight, modular relational database management system (RDBMS) from scratch in Go. The goal is to build a foundational engine that addresses the key challenges of data management, including schema handling, data manipulation, indexing, and transaction processing. The system will be architected for reliability and future extension, such as adding a network layer or distributed features.

The core problem to be solved is not just the implementation of individual components, but the seamless integration of these components to create a system that is correct, reliable, and durable. This includes:

1.2.1 **SQL Query Processing**: The engine will provide robust support for essential SQL operations through a complete parsing and execution pipeline. This requires developing a sophisticated SQL tokenizer and parser capable of interpreting Data Definition Language (DDL) commands such as CREATE TABLE and DROP TABLE, alongside Data Manipulation Language (DML) operations including INSERT, SELECT, UPDATE, and DELETE. The SELECT implementation will support complex WHERE clauses with multiple predicate conditions and logical operators.

1.2.2 **Persistent Storage Architecture** : The system will guarantee absolute data durability and prevent data loss under any failure scenario. This will be achieved by implementing a disk-based storage layer using page files and a Write-Ahead Log (WAL). The WAL ensures that all changes are recorded to a persistent log before being applied to the main data files, guaranteeing durability in the event of a crash.

1.2.3 **High-Performance Indexing**: Data retrieval will be fast, even with large datasets. The project will implement B+-trees to create both a clustered primary index (where data records are stored directly in the leaf nodes) and secondary indexes, dramatically improving query performance.

1.2.4 **ACID Transaction Management**: To ensure data consistency and support concurrent database access, the system will implement a complete transaction processing subsystem that enforces full ACID properties (Atomicity, Consistency, Isolation, Durability). The initial implementation will utilize a strict Two-Phase Locking (2PL) concurrency control protocol to manage simultaneous transactions while preventing data corruption and maintaining isolation guarantees.

1.2.5 **Fault-Tolerant Recovery** : The system will be able to restore itself to a consistent state after an unexpected failure. A robust recovery mechanism will use the WAL to both redo committed transactions and undo incomplete ones upon restart, ensuring the database is always in a valid state.

This comprehensive approach addresses the inherent complexity of modern database systems while maintaining architectural clarity and extensibility for future enhancements such as distributed processing capabilities or advanced query optimization features.

Chapter 2

Background/ Literature Survey

Relational databases have served as the fundamental infrastructure of modern computing systems for over four decades, providing structured mechanisms for storing, retrieving, and manipulating vast quantities of data with guaranteed consistency and reliability. Industry-leading systems including MySQL, PostgreSQL, Oracle Database, and SQL Server represent the culmination of extensive research and development efforts, incorporating sophisticated features such as advanced indexing algorithms, query optimization techniques, distributed transaction processing, and fault-tolerant recovery mechanisms. This literature survey examines the theoretical foundations and practical implementations that inform the design of relational database engines, with particular emphasis on storage architectures, indexing strategies, transaction management protocols, durability guarantees, and crash recovery algorithms.

2.1 Storage Structure and Data Organization

At the lowest level, databases organize information as key–value pairs, with the primary key serving as a unique identifier for rows and the remaining attributes stored as the value [1]. Higher-level relational abstractions such as schemas, constraints, and SQL query execution are built on top of this storage model. PostgreSQL, for example, uses a page-based storage architecture, where disk space is divided into fixed-size blocks for efficient I/O [2].

Durability in storage systems is achieved using the Write-Ahead Log (WAL). In WAL, before a data page is modified, the change is recorded in a persistent log file. This guarantees that in the event of a crash, all committed transactions can be recovered by replaying the log [3]. PostgreSQL implements WAL with checksums to ensure integrity, while MySQL's InnoDB engine also uses log-based fault tolerance [2].

2.2 Indexing Mechanism

Indexing is one of the most crucial components for query optimization. Without indexes, databases are forced to perform sequential scans, which become prohibitively slow on large datasets. Bayer and McCreight introduced the B-Tree and its optimized variant, the B+ Tree, which remain the de facto standard for indexing in disk-based systems [4].

B+ Trees minimize disk reads by organizing data into a hierarchical structure, supporting logarithmic-time search, insertion, and deletion. They are particularly effective for Online Transaction Processing (OLTP) workloads where low-latency queries are critical. PostgreSQL and MySQL both rely heavily on B+ Tree indexes for primary and secondary keys [2].

An alternative structure, the Log-Structured Merge (LSM) Tree, has been explored in write-heavy systems. However, due to its predictable performance and strong support for range queries, the B+ Tree remains the dominant choice for relational databases [5].

2.3 Transaction Management and ACID Properties

The foundation of reliable database systems lies in the ACID properties—Atomicity, Consistency, Isolation, and Durability [6]. Atomicity ensures that a transaction is either executed fully or not at all. Consistency guarantees that a transaction moves the database from one valid state to another. Isolation ensures that concurrent transactions do not interfere incorrectly, and Durability ensures that committed transactions persist despite failures.

Two-Phase Locking (2PL) is among the most widely studied and implemented concurrency control mechanisms to enforce isolation [7]. In 2PL, a transaction acquires all necessary locks during a growing phase and releases them only in the shrinking phase after completion. This guarantees serializability, although deadlocks may occur and will be resolved. PostgreSQL uses Multi-Version Concurrency Control (MVCC), allowing readers and writers to work without blocking each other, improving throughput under concurrent workloads [2].

2.4 Durability and Crash Recovery

Durability and crash recovery have been extensively studied. The ARIES recovery algorithm proposed by Mohan et al. [3] is a landmark contribution. ARIES uses WAL along with checkpoints and log sequence numbers to guarantee recovery. It operates in three phases—analysis, redo, and undo. This ensures that committed transactions are redone after a crash, while incomplete transactions are undone, restoring the system to a consistent state.

Periodic checkpoints further reduce recovery time by creating points in the WAL where all dirty pages have been flushed to disk. Both PostgreSQL and MySQL adopt ARIES-like strategies in their recovery subsystems, with slight variations in implementation [2].

2.5 Summary of Literature

From the literature, it is evident that efficient indexing, transactional integrity, durability, and recovery mechanisms form the backbone of relational database engines. The B+ Tree remains the most practical indexing structure for disk-based OLTP workloads, while WAL-based approaches like ARIES provide robust crash recovery. Systems like PostgreSQL demonstrate how these principles can be applied in practice with page-based storage, MVCC, and WAL logging.

The proposed project, DaemonDB, builds upon these established concepts by implementing:

2.5.1 Page-based storage with Write-Ahead Logging for durability.

2.5.2 B+ Tree indexing for efficient query performance.

2.5.3 SQL parsing and execution for core DDL and DML commands.

2.5.4 A transaction subsystem based on strict Two-Phase Locking for atomicity and durability.

2.5.5 WAL-based recovery to restore consistency after crashes.

This modular approach ensures that the project is grounded in proven database principles while allowing for incremental extensions in the future.

Chapter 3

Objectives

3.1 Primary Objectives

3.1.1 **Persistent Storage Architecture:** Design and implement a robust disk-based storage subsystem utilizing page-oriented file structures combined with a comprehensive Write-Ahead Logging (WAL) mechanism. This storage layer will ensure absolute data durability through strict logging protocols that record all database modifications in persistent log files before applying changes to primary data pages. The implementation will support efficient random and sequential access patterns while maintaining data consistency across all failure scenarios, including unexpected system shutdowns and hardware malfunctions.

3.1.2 **Indexing:** Develop B+-tree indexes to support clustered (primary) and secondary indexes for fast and efficient data retrieval. Enable support for range queries and quick lookups to improve query performance significantly.

3.1.3 **SQL Processor :** Implement a SQL tokenizer and parser to handle core DDL and DML commands, including CREATE, INSERT, SELECT, UPDATE, and DELETE.

3.1.4 **ACID Transaction Management:** Implement a transaction subsystem using strict Two-Phase Locking (2PL) to guarantee atomicity and durability during concurrent operations.The transaction manager will handle lock acquisition and release protocols, deadlock detection and resolution, transaction rollback mechanisms, and commit processing while maintaining optimal system throughput under high-concurrency workloads.

3.1.5 Fault-Tolerant **Crash Recovery :** Design a recovery mechanism using WAL to restore database consistency and integrity after system crashes or failures.Support undo and redo operations to recover both committed and uncommitted transactions correctly.

3.2 Secondary Objective

3.2.1 **Distributed Database Architecture:** Extend the foundational single-node implementation to support distributed database capabilities, enabling horizontal scaling across multiple server nodes. This enhancement will incorporate distributed transaction processing, data partitioning strategies, consensus algorithms for maintaining consistency across nodes, and fault tolerance mechanisms for handling partial system failures. The distributed architecture will maintain compatibility with the core SQL interface while providing transparent scalability for large-scale applications.

Chapter 4

Possible Approach/ Algorithms

The database engine will be developed in the following layers

4.1 Storage Engine

The storage engine is the lowest layer of the database, responsible for managing data on disk.

4.1.1 Page-Based Storage: The core approach is to treat the disk as a collection of fixed-size pages. This is a standard method that optimizes disk I/O, as most I/O operations are most efficient when they are block-oriented.

4.1.2 Write-Ahead Logging (WAL): To ensure the Durability of committed transactions, a WAL will be used. All changes to the database will first be recorded as an append-only log entry on disk before the corresponding data pages are updated.

4.1.2.1 Log Records: Every database modification will generate a log record. This record contains enough information to redo or undo the change.

4.1.2.2 Page Flushing: The modified data pages can be written to disk asynchronously, as long as the corresponding WAL records have been safely written.

4.2 Indexing and Data Storage

The indexing layer is critical for efficient data retrieval without having to scan the entire table.

4.2.1 B+-tree Implementation: B+ tree is highly effective for disk-based systems because its structure minimizes the number of disk reads required to locate a record.

The B+-tree implementation will support standard operations:

4.2.1.1 Search: A logarithmic traversal from the root to the leaf node to find a key.

4.2.1.2 Insert: Find the appropriate leaf node and insert the new record.

4.2.1.3 Delete: Find and remove a key from a leaf node.

4.2.2 Data-in-Leaf-Nodes: Instead of a separate heap file for records, the project will use a clustered B+-tree where the actual data records are stored directly within the leaf nodes.

4.2.3 Primary and Secondary Indexes:

4.2.3.1 Clustered Primary Index: This index stores the full data record in its leaf nodes, ordered by the primary key. This arrangement makes primary key lookups extremely fast.

4.2.3.2 Secondary Indexes: These indexes will be separate B+-trees that store the indexed column's value and a reference to the corresponding data record. Since the data is in the primary index, the reference will be the primary key of the record. This means a secondary index lookup requires an additional lookup in the primary index to retrieve the full record.

4.3 SQL Processing

The SQL processor translates a human-readable query into an executable plan for the database engine.

4.3.1 Modular Pipeline: The process will be broken down into a series of steps:

4.3.1.1 Tokenization: The raw SQL string is broken into tokens.

4.3.1.2 Parsing: The sequence of tokens is validated against a grammar and converted into an Abstract Syntax Tree (AST).

4.3.1.3 Execution Engine: The AST is used to create and execute a query plan.

4.4 Transaction Management

Transaction management ensures that the database remains in a consistent state, even with multiple operations happening at once.

4.4.1 Strict Two-Phase Locking (2PL): This concurrency control protocol will be the initial implementation to guarantee strong isolation.

4.4.1.1 Growing Phase: A transaction acquires locks (shared for reads, exclusive for writes) but does not release any.

4.4.1.2 Shrinking Phase: After the transaction commits or aborts, all its locks are released at once.

4.4.2 Deadlock Detection: A simple algorithm (e.g., a wait-for graph check) can be implemented periodically to detect cycles and abort.

4.5 Recovery

The recovery mechanism ensures the database can restore a consistent state after a crash.

4.5.1 Approaches:

WAL-based Redo/Undo: The recovery system will use the WAL to both redo committed transactions that didn't make it to the disk and undo uncommitted transactions that were in progress.

4.5.2 Algorithms:

4.5.2.1 ARIES (Algorithm for Recovery and Isolation Exploiting Semantics): An ARIES-style recovery process is a robust and widely used approach for systems with a WAL. It has three main phases:

Analysis: On restart, the system reads the WAL from the last checkpoint to identify which transactions were active and which data pages were dirty at the time of the crash.

Redo: It replays all committed and uncommitted actions in the WAL to bring the database to the state it was in at the time of the crash.

Undo: It rolls back the effects of any transactions that were active (not committed) at the time of the crash, ensuring atomicity.

4.5.2.2 Checkpoints: Periodically, the database will take a checkpoint to reduce the amount of work required during recovery. A checkpoint writes a special record to the WAL, indicating that all dirty pages have been flushed to disk, allowing the recovery process to start from that point rather than the beginning of the log.

References

[1]Stonebraker, M., & Hellerstein, J. M. *What Goes Around Comes Around*. Readings in Database Systems, 4th Edition, MIT Press, 2005.

[2]PostgreSQL Global Development Group. *PostgreSQL Documentation: Storage and WAL Internals*, 2024.

[3]Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." *ACM TODS*, 1992.

[4]Bayer, R., & McCreight, E. "Organization and Maintenance of Large Ordered Indexes." *Acta Informatica*, 1972.

[5]O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. "The Log-Structured Merge-Tree (LSM-Tree)." *Acta Informatica*, 1996.

[6]Haerder, T., & Reuter, A. "Principles of Transaction-Oriented Database Recovery." *ACM Computing Surveys*, 1983.

[7]Gray, J., & Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8]Build Your Own Database From Scratch in Go. *https://build-your-own.org/database*, accessed September 2025.

[9]Go database/sql tutorial. *https://go-database-sql.org*, accessed September 2025.