A black and white logo

AI-generated content may be incorrect.

**School of Engineering and Computing**

**MSc Information Technology**

**Interim Report**

***Adaptive Database Indexing: Switching Between B-Tree and LSM Tree based on workload***  
    
    
    
**by**   
 

***Prajwalaradhya Shivakumaraswamy Kesaramadu***

***B01759301***

**Date of submission: *11/07/2025***

**Supervisor: Wajahat Ali Khan**

**Moderator: *Saqib Hussain***

Table of Contents

[Literature Review 3](#_Toc203154576)

[Research Methodology 4](#_Toc203154577)

[Plan for Completion 6](#_Toc203154578)

[References 7](#_Toc203154579)

# Literature Review

The rapid evolution of data-intensive applications has driven significant research into database indexing techniques that balance performance, scalability, stability, and storage efficiency under various workloads. Traditional B-Tree indexes remain the backbone of many relational systems due to their predictable performance for transactional queries and rock-solid stability, while Log-Structured Merge (LSM) trees have emerged in late 90’s as a compelling alternative for write-heavy environments, offering superior write throughput and space efficiency. However, each approach has its own trade-offs, and there is growing interest in adaptive indexing strategies capable of dynamically leveraging both data structures based on workload characteristics. This literature summary reviews key research contributions, implementations, and performance analyses relevant to adaptive indexing and the comparative strengths of B-Trees and LSM trees, laying the groundwork for exploring a system that can switch between these indexing paradigms to optimize database performance.

The B-Tree has been a fundamental data structure in database systems since its introduction by Bayer and McCreight, revolutionized data indexing by offering a balanced, disk-friendly structure that minimizes I/O operations. Its design divides data into nodes stored in pages, maintaining a balanced height even as records are inserted or deleted. This ensures that searches, inserts, and deletes consistently execute in logarithmic time (O(log n), where n is the size of dataset), a property crucial for managing large-scale datasets stored on block-based storage systems. Early systems like IBM’s Virtual Storage Access Method (VSAM) integrated B-Trees to efficiently index sequential datasets, allowing rapid record retrieval and updates without scanning entire files. In modern relational databases, B-Trees remain the dominant index type: MySQL employs them as the core of InnoDB’s clustered and secondary indexes, where leaf nodes directly store row data or record pointers, optimizing point lookups and range queries. Similarly, PostgreSQL relies heavily on B-Trees for its default index implementation, where each index entry comprises a key and a pointer to a tuple in the table heap, supporting efficient multi-version concurrency control. B-Trees excel under transactional workloads with a balanced mix of reads and writes, offering stable performance with predictable latencies. However, B-Trees can struggle under write-intensive workloads with high rate due to the cost of maintaining node balance and frequent page splits or merges. Despite newer alternatives like LSM trees, B-Trees remain ubiquitous thanks to their well-understood behaviour, mature implementations, and suitability for OLTP (Online Transactional Processing) systems that demand reliable indexing and fast query performance.

The Log-Structured Merge-tree (LSM-tree), introduced by O’Neil et al. (1996), is a write-optimized data structure designed to improve write throughput by deferring and batching disk operations. Unlike B-Trees, which modify data in-place, LSM-trees accumulate inserts, updates, and deletes in memory (memtables) and flush them periodically to disk as sorted, immutable files called SSTables. These flushed files are organized across multiple levels and are merged in the background through compaction processes, significantly reducing random I/O and improving write performance. To mitigate the overhead introduced during reads—especially the need to search across multiple SSTables—LSM-trees often incorporate auxiliary data structures like Bloom filters and fence pointers to reduce unnecessary disk accesses. Kleppmann emphasizes that this architecture makes LSM-trees ideal for workloads characterized by high write volumes and sequential write patterns, such as log ingestion systems (metrics), time-series databases, event tracking applications and recently vector storage systems. However, this comes with trade-offs in read latency and write amplification during compactions, which must be carefully balanced depending on the application’s workload pattern.

Building upon the foundation of LSM-trees, modern key-value storage engines like RocksDB and Pebble have introduced significant optimizations for production use in large-scale systems. RocksDB, developed by Facebook, is a high-performance, embeddable engine that enhances the original LevelDB (by Google) with advanced features such as configurable compaction strategies, compression, write-ahead logs, and support for column families. It is extensively tuned for heavy write workloads and powers many Facebook services that require low latency writes with high durability. Pebble, developed by Cockroach Labs, is a RocksDB-inspired key-value store written in Go, specifically engineered to integrate with CockroachDB’s distributed SQL layer. While retaining core LSM principles, Pebble focuses on predictable performance, better write amplification control, and improved integration with Go-based systems (Cockroach Labs, 2020). Both systems reflect the evolution of LSM-tree design into robust, production-ready engines that trade rigid consistency for throughput and efficiency, and they serve as real-world examples of how LSM-based indexes can outperform B-Trees in write-dominant use cases.

Relational databases such as MySQL and PostgreSQL rely exclusively on B‑Tree indexes for both primary and secondary indexing, offering robust transactional guarantees and consistent performance in OLTP workloads. NewSQL systems like CockroachDB and TiDB combine SQL compatibility with distributed storage engines: CockroachDB uses the Pebble LSM engine, while TiDB builds on TiKV, another LSM-powered layer. Both systems leverage LSM structures for write efficiency and horizontal scaling across nodes. Meanwhile, MyRocks, implemented in Percona Server 5.7, enables applications to mix InnoDB (B‑Tree) and RocksDB (LSM) engines within the same instance. Facebook’s MyRocks was originally developed to serve the massive social graph of UDB, significantly reducing storage space and write amplification while leveraging relational database features. These implementations illustrate a clear industry shift: while B‑Tree-based systems remain central to transactional workloads, modern distributed and write-optimized systems increasingly adopt LSM architectures—and hybrid engines like MyRocks aim to offer the best of both worlds.

# Research Methodology

This project uses an experimental research approach to study hybrid indexing, where both B+ Tree and LSM Tree structures are combined to improve database performance under different types of workloads. Modern applications often have mixed workloads that include high volumes of inserts, updates, and deletes, alongside frequent reads and range queries. Traditional databases typically rely on B+ Trees for indexing, which work well for balanced read-write workloads but can struggle with heavy write operations. In contrast, LSM Trees are designed for fast writes but can introduce complexity in read performance due to multiple data levels and compaction processes. This project investigates how allowing users to choose between these indexing methods—or combine them—could improve efficiency in relational databases. The research is based on findings from academic papers and involves practical experiments to benchmark real-world databases using both B+ Tree and LSM Tree storage engines.

In this project, synthetic data generation forms the foundation for conducting fair and consistent performance comparisons across different database engines. The data will be created using the Python library Faker, which can produce large volumes of realistic yet entirely random data, such as names, addresses, email addresses, numeric values, and timestamps. This allows the simulation of diverse, real-world workloads without relying on sensitive or proprietary datasets. Millions of records will be generated and stored in CSV files to facilitate easy and consistent bulk imports into multiple database systems. Separate datasets will also be generated specifically for testing data insertion performance, enabling measurement of how quickly each engine can handle large-scale data writes under different conditions. For this study, the chosen databases include PostgreSQL as a representative of systems using B+ Tree indexing, CockroachDB for the LSM Tree-based engine, and MySQL with the MyRocks storage engine from Percona Server 5.7, which supports a hybrid approach combining B+ Tree (via InnoDB) and LSM Tree structures (via RocksDB). Although schemas across these systems will be designed to be as similar as possible, certain differences in data types or table definitions may be necessary to accommodate engine-specific constraints or to optimize performance based on each system’s architecture. Indexing strategies will be explicitly defined for each table, informed by analysis of which columns are most likely to be queried frequently or involved in range scans, thereby reflecting realistic application usage. By default, primary key indexes will be implemented to ensure uniqueness and basic query performance. Additional indexes for high-priority columns will be created after table definitions are finalized but before loading the bulk data, so their influence on query and write performance can be properly evaluated. Foreign key constraints will be added only after initial data loading, as enforcing these constraints during bulk inserts can significantly complicate and slow down the import process. This user-defined indexing approach is intended to give developers the flexibility to choose indexing methods best suited for their application workloads, allowing them to optimise performance based on specific query patterns, data update frequencies, and storage considerations unique to their use cases.

The experimental setup for this project will include both local and cloud-based environments to simulate a range of real-world deployment scenarios. Locally, databases will be run in Docker containers, while cloud benchmarking will be carried out using AWS EC2 virtual machines. Due to cost constraints and to reflect the realities faced by small- to medium-sized businesses, all test instances—both in the cloud and locally—will be deliberately limited in resources, with configurations typically constrained to 1–2 GB of RAM and 1–2 CPU cores per instance. This ensures that the evaluation considers how these databases engines perform under resource-limited conditions, which is a common challenge in production environments outside large enterprises. Benchmarking tools such as HammerDB, along with custom Python and SQL scripts, will be employed to run a diverse set of SQL queries that mirror practical workloads. These include querying time-series data like transaction logs, retrieving specific records by unique identifiers, performing range queries such as fetching order details for a specific user, searching for products within a price range, and executing analytical operations like calculating averages or summing columns. This combination of varied query types will help assess the performance characteristics of B+ Tree, LSM Tree, and hybrid indexing engines under different query patterns and workloads.

Benchmarking results in this project will be evaluated using several key performance metrics, including CPU utilisation, memory consumption, disk I/O activity, and query latency. By measuring these factors across different database engines, the research aims to identify performance trade-offs and quantify the practical benefits of adopting a hybrid indexing strategy in SQL-based relational databases. To facilitate meaningful comparisons, various visualisations such as charts and graphs will be used to illustrate differences in resource usage and response times between B+ Tree, LSM Tree, and hybrid configurations. The analysis of these visual results will provide insights into how each indexing approach handles diverse workloads, ultimately informing the conclusions and recommendations presented at the end of the study.

In summary, the methodology for this project combines synthetic data generation, controlled experimental setups, and detailed benchmarking to investigate the performance implications of hybrid indexing strategies in relational databases. By systematically comparing B+ Tree, LSM Tree, and hybrid storage engines under various workloads and resource constraints, the study aims to generate practical insights into the strengths and limitations of each approach. The findings are intended to guide developers and database architects in selecting indexing strategies that best suit their specific application requirements and operational environments.

# Plan for Completion

At present, significant progress has been made in both the technical and theoretical foundations of the project. On the data side, Python scripts have been developed to generate a synthetic e-commerce dataset, simulating a typical business domain with entities such as users, products, product categories, orders, payment transactions, and reviews. Datasets for users, products, and categories have been completed, as these are relatively independent of other data relationships. Work continues on creating interrelated datasets, such as orders and transactions, which require careful handling of foreign key references and realistic value distributions.

In parallel, substantial effort has been devoted to preparing the experimental environment. This includes writing the literature review, drafting the methodology section, and exploring the practical steps required for deployment and testing. Activities so far include setting up Docker environments, installing and configuring the target database systems—PostgreSQL, CockroachDB, and Percona Server with MyRocks—and learning the nuances of SQL operations specific to each engine. Work is also ongoing in understanding the setup and use of benchmarking tools like HammerDB for workload simulation and performance testing.

Looking ahead, the next phase of the project will involve several key activities. Technically, remaining datasets will be completed and integrated into the test databases. Benchmarking will be performed both locally (via Docker) and in the cloud using AWS EC2 instances. A significant part of this phase involves learning how to collect system and database performance metrics from both local and cloud environments. For cloud experiments, Grafana Cloud has been identified as a promising solution for monitoring and visualisation, while for local testing, tools like Google’s cAdvisor are being explored for container-level metrics collection. The plan includes scraping and storing this performance data, potentially in a time-series database, to support detailed analysis.

Once benchmarks are executed, results will be analysed using statistical methods and visualisation tools, which might include Python’s Matplotlib, Grafana dashboards, or other suitable data visualisation platforms yet to be finalised. The objective is to identify and interpret differences in resource consumption, throughput, and latency across the tested engines. The findings will contribute to forming recommendations for the use of hybrid indexing strategies in relational database systems.

In the event that some parts of the testing or data collection process do not proceed as planned—for instance, difficulties in collecting metrics from AWS due to network restrictions, unexpected software compatibility issues, or insufficient benchmark data—the plan is to rely on the local Docker-based tests as a fallback to ensure the study can still be completed. Additionally, if the anticipated performance benefits of hybrid indexing are not observed, the project will pivot to a discussion of why these results emerged and will reflect critically on the circumstances under which hybrid approaches may or may not be suitable.

The remaining weeks will therefore focus on completing data generation, configuring benchmarks, executing tests, collecting and analysing results, and finalising the dissertation report. The final analysis will not only evaluate technical outcomes but will also reflect on how the skills and knowledge gained relate to the MSc Information Technology programme and future professional development.

# References

Comer, D. (1979). Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2). doi:https://doi.org/10.1145/356770.356776.

Dong, S., Kryczka, A., Jin, Y. and Stumm, M. (2021). RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage*, 17(4), pp.1–32. doi:https://doi.org/10.1145/3483840.

Evans, G. (2025). *Understanding the mechanics of PostgreSQL B-Tree indexes*. [online] Fastware.com. Available at: https://www.postgresql.fastware.com/pzone/2025-01-understanding-the-mechanics-of-postgresql-b-tree-indexes [Accessed 5 Jun. 2025].

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. ‘O’Reilly Media, Inc.’

Lu, L., Pillai, T.S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. (2016). WiscKey: separating keys from values in SSD-conscious storage. *File and Storage Technologies*. [online] Available at: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lu.pdf [Accessed 5 Jun. 2025].

Matsunobu, Y., Dong, S. and Lee, H. (2020). MyRocks. *Proceedings of the VLDB Endowment*, 13(12), pp.3217–3230. doi:https://doi.org/10.14778/3415478.3415546.

Mattis, P. (2020). *Introducing Pebble: A RocksDB-inspired key-value store written in Go*. [online] Cockroachlabs.com. Available at: https://www.cockroachlabs.com/blog/pebble-rocksdb-kv-store/ [Accessed 5 Jun. 2025].

O’Neil, P., Cheng, E., Gawlick, D. and O’Neil, E. (1996). The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), pp.351–385. doi:https://doi.org/10.1007/s002360050048.