

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

Prajwalaradhya Shivakumaraswamy Kesaramadu

B01759301

Thesis for the partial fulfilment

of the requirements for the Masters Degree

in Information Technology

University of the West of Scotland

School of Computing, Engineering and Physical Sciences

<Date of submission>

A close-up of a logo

AI-generated content may be incorrect.

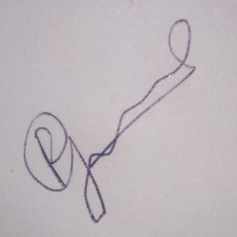
**DECLARATION OF ORIGINALITY**

I declare that this is an original study based on my own

work and that I have not submitted it for any other

course or degree.

Signature



****

**Library Form to Accompany MSc Project**

|  |
| --- |
| **Surname: Shivakumaraswamy Kesaramadu** |
| **First Name: Prajwalaradhya Initials:** |
| **Banner No: B01759301** |
| **Course Code: COMP11117** |
| **Course Title: MSc Masters Project** |
| **Project Supervisor: Wajahat Ali Khan** |
| **Project Title: Adaptive Database Indexing: Switching Between B-Tree and LSM Tree based on workload** |
| **Session: Date of Submission:** |
| **Signature:** |

Abstract

The primary objective of this study is to investigate how different database indexing techniques impact performance under varying workloads, with a focus on adapting storage engines dynamically to meet specific system demands. The research is set against the backdrop of modern applications—such as e-commerce platforms—that experience fluctuating read and write patterns, where indexing strategy directly influences performance and resource usage.

The central research question examined is whether combining B-Tree and LSM Tree storage engines within a single system can improve database efficiency by aligning engine strengths with the workload type—fast reads with B-Trees and fast writes with LSM Trees.

To explore this, a benchmark environment was created using PostgreSQL, Percona MySQL with MyRocks, and CockroachDB. An SME-scale e-commerce dataset was generated, and tests were conducted in both local and cloud (AWS ECS) environments. Apache JMeter was used to simulate workloads, while Prometheus, Grafana, and cAdvisor monitored performance metrics.

The results showed that MyRocks significantly outperformed others in write-intensive tasks, while PostgreSQL offered superior read performance. CockroachDB showed higher latency overall. These findings highlight the value of using hybrid storage engines to tune database performance according to real-time workload patterns.

This research contributes to performance-aware database architecture and offers practical insights for system scalability.

Table of Contents

[Introduction 8](#_Toc205455527)

[Literature Review 11](#_Toc205455528)

[Methodology 14](#_Toc205455529)

[Comparison of PostgreSQL, MySQL (InnoDB & MyRocks), and CockroachDB (PebbleDB) 19](#_Toc205455530)

[Data Generation and Preprocessing 20](#_Toc205455531)

Table of Figures

[Figure 1 Choosing InnoDB as Storage Engine 14](#_Toc205455399)

[Figure 2 Choosing MyISAM as Storage Engine 14](#_Toc205455400)

[Figure 3 Choosing CSV as Storage Engine 14](#_Toc205455401)

[Figure 4 Choosing RocksDB (LSM Tree) as Storage Engine 15](#_Toc205455402)

[Figure 5 RocksDB Architecture Diagram 16](#_Toc205455403)

[Figure 6 Entity Relationship Diagram of all Entities with Relationship 21](#_Toc205455404)

Table of Tables

[Table 1 Comparison b/w PostgreSQL, InnoDB, MyRocks and Pebble Storage Engine 20](#_Toc205455463)

[Table 2 Users Entity 21](#_Toc205455464)

[Table 3 Products Entity 21](#_Toc205455465)

[Table 4 Category Entity 21](#_Toc205455466)

[Table 5 Orders Entity 22](#_Toc205455467)

[Table 6 Order Entry Entity 22](#_Toc205455468)

[Table 7 Transactions Entity 22](#_Toc205455469)

# Introduction

A database is a system with an organised collection of data that enables efficient storage, retrieval, and manipulation of information. It forms the foundation of modern applications for data storage, from e-commerce platforms and financial systems to content management and mobile apps. One of the key strengths of databases is their ability to provide structured access to information. This is achieved through the use of schemas, which define the structure, types, and relationships of data. Structured access allows users and applications to interact with data in a consistent and logical manner, regardless of the data’s size or complexity. They are also designed to support concurrent access by multiple users or systems. This is made possible through advanced transaction management and concurrency control mechanisms, which ensure that simultaneous operations do not lead to data inconsistencies or corruption aka ACID.

In addition, databases enforce data integrity by applying rules such as constraints, keys, and validation checks, ensuring that only valid and meaningful data is stored. For example, a database can prevent the insertion of duplicate entries, enforce referential integrity between tables, and maintain consistent formatting for fields. To protect sensitive information, most databases provide robust security mechanisms. These include user authentication, role-based access control, and encryption of data both at rest and in transit with Transport Layer Security(TLS). Furthermore, databases support recovery and backup features to ensure that data can be restored in case of accidental loss, system failure, or cyber-attacks.

As organisations increasingly dependent on data-driven decision-making, the volume and complexity of data have grown substantially. With the rise of real-time systems, streaming data, and distributed architectures, databases must now handle diverse and unpredictable workloads. These can include read-heavy operations for reporting, write-heavy operations for transaction processing, or mixed workloads that require both to be handled efficiently. As a result, designing databases that can maintain high performance under varying conditions has become a significant area of research and engineering focus.

One of the critical features that influence database performance is indexing. Indexes are auxiliary data structures that speed up the retrieval of records by allowing the database engine to avoid scanning the entire table for each query. Instead, indexes act like a roadmap that narrows down the search space, especially in queries involving filters, sorts, joins, and aggregations. Effective indexing can significantly reduce query response times, particularly in large datasets with millions of rows. While indexing offers substantial performance benefits, especially in read-heavy workloads, it also introduces certain trade-offs that must be carefully managed. One of the primary costs associated with indexing is increased storage usage. Indexes themselves occupy space on disk, and in systems with many indexed columns or composite keys, the overhead can become significant. Furthermore, maintaining indexes comes with computational costs, particularly during write operations. When data is inserted, updated, or deleted, the relevant indexes must also be updated to reflect the changes. This process may involve recalculating key positions, restructure index nodes, or even rewriting entire portions of the index. As a result, excessive or poorly designed indexing strategies can degrade performance during heavy write operations. Another important consideration is the balance between read and write performance. While indexes greatly accelerate read operations, they tend to slow down write-intensive workloads due to the additional overhead of keeping the index structures up to date. This trade-off becomes especially important in systems where workloads are not static but fluctuate over time.

To address these challenges, database systems adopt different indexing mechanisms, each optimised for specific workload characteristics. Among the most widely used indexing structures are the B-Tree and the Log-Structured Merge Tree (LSM Tree). B-Trees are typically used in traditional relational database systems due to their efficient support for point lookups and range-based queries. They maintain a balanced tree structure that allows data to be accessed in sorted order, making them well-suited for analytical and transactional read queries. In contrast, LSM Trees are designed to optimise write performance. They achieve this by writing data in an append-only fashion to in-memory structures called MemTables and periodically flushing it to disk in sorted batches called SSTables (Sorted String Tables) which are immutable. This model avoids in-place updates and reduces random disk I/O, which makes LSM Trees particularly suitable for write-heavy workloads. However, this design can lead to higher read latencies, especially for random access queries, as data may be spread across multiple SSTables and require background compaction for consistency and efficiency.

Although B-Trees are more common in SQL-based systems and LSM Trees are widely used in NoSQL databases, this distinction is not inherent to the data model. These indexing mechanisms are storage engine-level implementations, meaning their use is independent of whether the database supports SQL or not. For example, MySQL supports both B-Tree (via InnoDB) and LSM Tree (via MyRocks) engines within the same database system, offering the flexibility to assign different engines to different tables based on the expected workload. This flexibility opens opportunities for adaptive indexing strategies, where systems can be designed to take advantage of both indexing models by aligning them with workload patterns—using B-Trees for tables with frequent read access and LSM Trees for write-intensive operations. This forms the central focus of the present study.

While choosing a single storage engine simplifies system design and operations, it can introduce limitations in scenarios where diverse workloads must be optimally handled. This does not imply that using a single engine is inherently a disadvantage; however, certain applications may require more specialized performance characteristics—for instance, a system that handles both read-heavy and write-intensive tasks. In such cases, using different databases for different tasks becomes a common approach, but it adds complexity in terms of learning curve, operational overhead, infrastructure maintenance, and integration between systems. Maintaining separate clusters for different databases demands significant engineering effort and domain expertise. In contrast, having a database system that supports multiple storage engines natively provides greater flexibility in choosing performance trade-offs based on specific needs. It eliminates the need for application-layer sharding or synchronization and minimizes disruptions caused by switching database platforms. Additionally, it streamlines development by keeping the codebase unified and consistent.

The primary aim of this research is to evaluate how a database system capable of supporting multiple storage engines can be leveraged in real-world workloads. This study explores its feasibility for production environments by benchmarking its performance against well-established databases. Specifically, it compares PostgreSQL, MySQL with RocksDB (via Percona Server 5.7), and CockroachDB under a range of workload patterns, such as high read or write operations, while also considering constraints like limited CPU, memory, and I/O bandwidth. To ensure fair and repeatable performance testing across all databases, a consistent and controlled testing environment is crucial. For this purpose, Docker is used to emulate a clean, production-like setup. Docker containers allow for fine-grained resource constraints such as CPU, memory, and network bandwidth, which help simulate real-world deployment conditions accurately. This also ensures that each database instance operates under the same baseline conditions, enabling meaningful comparisons.

For workload generation, Apache JMeter is used with JDBC connectors to simulate real-time query executions and transactions. CockroachDB and PostgreSQL both support the PostgreSQL wire protocol, making them compatible with the same JDBC configuration, while Percona Server 5.7 (with RocksDB as its storage engine) is accessed via the MySQL protocol. To populate the databases with realistic test data, Python scripts are written using the Faker and Pandas libraries, allowing the creation of large and diverse datasets representative of real-world e-commerce applications. To monitor performance metrics such as CPU usage, memory consumption, and query throughput, Prometheus is used in combination with Grafana dashboards and cAdvisor for container-level statistics. For cloud-based benchmarking and scalability evaluation, Amazon ECS (Elastic Container Service) is utilized to deploy containerized database instances in a managed, distributed environment.

In summary, this research explores the need for adaptable database systems capable of handling diverse workloads without the operational overhead of maintaining multiple database technologies. By focusing on storage engines and indexing strategies, the study investigates whether a single database system with support for multiple storage engines can deliver optimal performance across varying conditions. Through comprehensive benchmarking of PostgreSQL, Percona Server with RocksDB, and CockroachDB under different resource constraints and workload patterns, this research aims to provide practical insights into their real-world applicability. The goal is not only to evaluate performance metrics but also to understand the trade-offs and design considerations that can guide engineers in choosing the right database architecture for mixed workload applications.

# Literature Review

Efficient data retrieval and storage are fundamental challenges in the design of modern database systems. Indexing structures play a critical role in addressing these challenges by enabling fast query execution and supporting large-scale data management. Two predominant indexing approaches have emerged over the decades: B-Trees and Log-Structured Merge Trees (LSM Trees). Each of these structures offers unique advantages and trade-offs, reflecting differing priorities in balancing read and write performance. B-Trees have long been the standard indexing mechanism in relational databases, favored for their ability to provide balanced, sorted access to data with logarithmic time complexity for search, insertion, and deletion operations. Their design aligns closely with the characteristics of traditional disk storage by organizing data into nodes, or pages, that match disk block sizes, thereby minimizing costly I/O operations. This alignment ensures that B-Trees maintain efficient lookup times and support range queries naturally, which are common in transactional and analytical workloads. In contrast, LSM Trees were developed to optimize write-intensive workloads, particularly in environments where data ingestion rates are high and random write costs are prohibitive. By employing an append-only approach that batches writes and periodically merges sorted data segments, LSM Trees significantly reduce write amplification and improve throughput. This design, however, introduces complexity in read operations, as queries may need to access multiple data segments and merge results dynamically.

PostgreSQL, often referred to simply as Postgres, is a powerful, open-source object-relational database system known for its stability, extensibility, and standards compliance. It has a rich history dating back to the 1980s and is considered one of the most advanced general-purpose databases in production today. PostgreSQL's origins trace back to the POSTGRES project at the University of California, Berkeley, led by Professor Michael Stonebraker in 1986. It was designed as a successor to the Ingres database system, focusing on support for complex data types and extensibility. The project aimed to address limitations in traditional relational databases by adding object-oriented features. The system evolved over time and was officially released to the public as PostgreSQL 6.0 in 1996, marking the transition from an academic project to a robust, community-driven open-source database.

PostgreSQL is often described as an object-relational database management system (ORDBMS) because of its support for:

* User-defined data types: Users can define custom data types same like OOP objects, but in an SQL which can be used when creating tables.
* Custom functions: Postgres has many built-in functions, but it also lets users to write their own functions based on individual requirements and are particularly useful for repetitive tasks or operations unique to a given workload. Few types of functions are:
  + SQL functions: functions written directly in SQL.
  + Procedural language functions: functions written in languages like PL/pgSQL.
  + Internal functions: built-in functions provided by PostgreSQL.
  + C-language functions: functions written in C for advanced performance and customization.
* Inheritance in table structures: A table able to inherit the structure and properties. The concept Inheritance is taken from OOP, and it works the same way.
* Supports data types which can be found in modern programming languages such as array data types, JSON, HSTORE for storing key-value pairs, etc.

PostgreSQL provides a unique combination of traditional RDBMS features and modern NoSQL-like flexibility. Its advantages include:

* Extensibility: New data types, index methods, and functions can be added without recompiling the core system.
* Rich Data Modelling: Perfect for complex business domains due to support for custom types and inheritance.
* Stability: Over 30 years of development has resulted in a stable and trusted database engine.

Cross-Platform and Cloud-Ready: Supported on all major Operating Systems and offered as a managed service by major cloud providers like Amazon Web Services, Google Cloud Platform, and Microsoft Azure.

PostgreSQL uses a single, monolithic storage engine that is tightly integrated with the rest of the database system. This native engine is designed and optimized for B+ Tree-based indexing, MVCC (Multi-Version Concurrency Control), and ACID compliance. In MySQL, which supports pluggable storage engines like InnoDB, CSV, MyISAM and RocksDB. Key Differences from InnoDB are its engine is not swappable; everything from transaction logs to indexing and storage is part of one single engine, offering deep optimization and tight consistency. PostgreSQL uses WAL for durability, but its implementation is often considered more transparent and developer friendly. It maintains B+ Trees differently, using background processes like the autovacuum daemon to reclaim space and maintain performance.

MySQL is a widely used open-source relational database management system known for its speed, simplicity, and reliability. It is one of the most popular databases for web-based applications, with widespread use across enterprises, startups, and open-source projects. It was originally developed by MySQL AB, a Swedish company, and released in 1995. It quickly gained popularity due to its lightweight nature and performance in web environments. MySQL AB was acquired by Sun Microsystems in 2008, and later by Oracle Corporation in 2010 after Oracle acquired Sun. Over the years, MySQL has evolved significantly, supporting ACID compliance, transaction processing, and a flexible plugin architecture that allows different storage engines to be used under a common SQL interface.

One of the unique architectural decisions in MySQL is its pluggable storage engine framework. This design decouples the SQL layer (responsible for parsing, planning, and executing SQL queries) from the storage layer (which manages data on disk). Here are some of the popular Storage Engines supported by MySQL:

* InnoDB is a general-purpose storage engine that balances high reliability and high performance and is the default MySQL storage engine. It uses B-Trees data structure to store the data in disk, and it is the only engine in MySQL which uses WAL (Write-Ahead-Logging) mode for durability and Point-In-Time Recovery.

A black screen with white text

AI-generated content may be incorrect.

Figure 1 Choosing InnoDB as Storage Engine

* MyISAM is a storage engine for MySQL databases, known for its speed in read operations and simple data management. It was the default storage engine for MySQL until InnoDB was introduced in version 5.5. It was lacking transaction and row-level locking support, because of these reasons, they dropped the support and chose InnoDB as the default engine.

A black background with white text

AI-generated content may be incorrect.

Figure 2 Choosing MyISAM as Storage Engine

* CSV is a simpler storage engine which is supported my MySQL database, as it stores data in text files using comma-separated values format.

A black background with white text

AI-generated content may be incorrect.

Figure 3 Choosing CSV as Storage Engine

* RocksDB/MyRocks is a MySQL storage engine that integrates with RocksDB. It provides improved flash storage performance through efficiencies in reading, writing, and storing data. It offers significantly better compression, up to 4x more efficient than uncompressed InnoDB, resulting in reduced storage usage. It achieves up to 10x lower write amplification, improving flash storage endurance and write throughput. Additionally, it provides faster replication and data loading by minimizing random reads and bypassing compaction overheads.

A black screen with white text

AI-generated content may be incorrect.

Figure 4 Choosing RocksDB (LSM Tree) as Storage Engine

The engine architecture allows developers and companies to build custom engines optimized for specific workloads. This flexibility became the foundation for the creation of MyRocks.

InnoDB, originally developed by Innobase (later acquired by Oracle), is the default transactional engine in MySQL since version 5.5. Features include ACID compliance with support for transactions, commit, rollback, MVCC (Multi-Version Concurrency Control) for high concurrency, Row-level locking, Clustered Indexing, Foreign key support, Automatic Crash Recovery and Point-In-Time Recovery/Rollbacks using Write-Ahead-Logging (WAL). It uses B+ Trees for primary and secondary indexes. Data is stored in a clustered format, where rows are physically ordered based on the primary key. When compared to PostgreSQL engine, it supports pluggable architecture at runtime. PostgreSQL provides richer data types and object-relational features, whereas MySQL (InnoDB) is more minimalistic in type support.

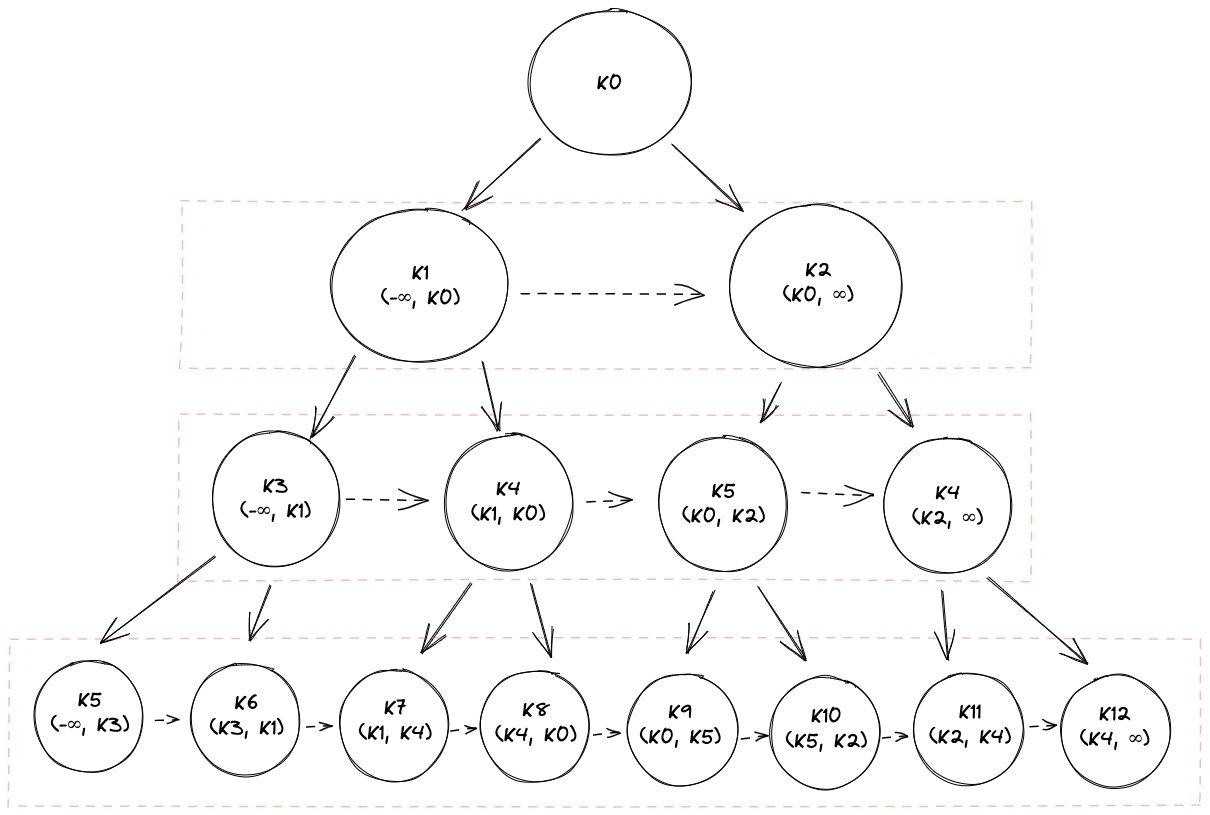


Figure 5 Structure of B-Tree

PostgreSQL and MySQL use B-Tree as in its core storage layer. It is a self-balancing tree data structure that was first introduced by Rudolf Bayer and Edward McCreight in 1972 to efficiently organize and retrieve large volumes of data stored on disk. Its primary purpose is to reduce the number of disk accesses required during search, insertion, and deletion operations, a critical concern given the relatively slow access speeds of magnetic storage compared to memory. Traditional binary search trees, while effective in memory, often degenerate into skewed structures when dealing with large datasets on disk, causing inefficient access patterns and excessive disk I/O. B-Trees overcome these limitations by allowing each node to contain multiple keys and child pointers, resulting in a tree with a high branching factor and shallow height. Internally, B-Trees organize data into pages or nodes that correspond to fixed-size blocks on disk, typically matching the size of a filesystem block or database page (commonly 4KB or 8KB). Each page contains sorted keys and pointers to child pages, enabling efficient in-node binary search and minimizing the number of page reads during traversal.

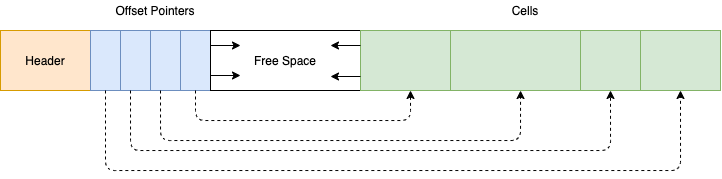


Figure 6 Structure of Page in B-Tree

This design allows B-Trees to handle millions of records with very few disk accesses, even under heavy workloads. In PostgreSQL, B-Trees serve as the default index type and are implemented as B+ Trees, a variant where all actual data entries are stored in leaf nodes linked sequentially to support efficient range scans. PostgreSQL’s B-Tree implementation optimizes for concurrent access using lightweight locking mechanisms and ensures that the tree remains balanced after insertions and deletions through page splits and merges.

It offers several advantages that make them well-suited for indexing in relational databases. Their balanced and sorted structure enables efficient point lookups and range queries with predictable, logarithmic time complexity. The alignment of nodes with disk pages minimizes the number of costly disk I/O operations, resulting in fast and reliable access even for large datasets. Moreover, B-Trees support dynamic updates, allowing insertions and deletions while maintaining balance, which is essential for transactional workloads. However, B-Trees also have inherent limitations. Write operations can become expensive due to page splits, merges, and rebalancing, leading to random disk writes and increased latency under heavy write loads. Additionally, their in-place update nature can cause higher write amplification, which is less optimal for modern storage technologies like SSDs. Despite these drawbacks, their maturity, robustness, and general-purpose efficiency have ensured that B-Trees remain the preferred indexing method in many database systems.

LSM Tree (Log-Structured Merge Tree) is a write-optimized storage engine architecture designed to handle high-throughput workloads by prioritizing fast inserts and updates. They were first introduced by Patrick O’Neil and colleagues in 1996 to address the inefficiencies of handling high-volume writes on disk-based systems. Unlike traditional B+ Trees, which modify disk pages directly during updates, LSM Trees defer writes in memory and periodically flush them to disk in a sequential, sorted manner. Incoming writes are first captured in an in-memory structure called a MemTable, and for durability, they are also written to a Write-Ahead Log (WAL) on disk. Once the MemTable reaches a predefined size threshold, it is flushed to disk as an immutable Sorted String Table (SSTable). These SSTables are organized into multiple levels, where newer SSTables reside at higher levels and older, larger SSTables move to lower levels. To maintain query performance and manage storage space, LSM Trees rely on a process called compaction, where overlapping SSTables are merged and rewritten, ensuring keys remain sorted and duplicates are removed. This compaction process, although necessary, can be computationally expensive and cause write amplification. To optimize read performance, LSM-based engines incorporate Bloom Filters, a probabilistic data structures that quickly determine whether a key might exist in an SSTable, helping avoid unnecessary disk reads. Despite being highly efficient for write-heavy workloads and sequential scans, LSM Trees are known to have slower point-read performance compared to B+ Trees due to scattered key locations across multiple SSTables and levels. Additionally, frequent compactions and random I/O can lead to higher CPU usage, disk wear, and latency spikes during read-heavy workloads. Nonetheless, the LSM Tree’s ability to handle massive write volumes with minimal disk seeks makes it an ideal choice for modern storage engines used in logging systems, key-value stores, and distributed databases.

A diagram of a diagram

AI-generated content may be incorrect.

Figure 7 Representation of write operation in LSM tree

When a write request is received by a database that uses the Log-Structured Merge Tree (LSM Tree) architecture, the operation is not immediately written to its destination on disk. Instead, to ensure durability and prevent data loss in the event of a system crash or power failure, the incoming write is first recorded in a Write-Ahead Log (WAL) — a sequential log file stored on disk that captures every modification before it is applied to the actual database. This step guarantees that even if the system crashes before the data is permanently written to storage, the database can recover and replay the WAL to restore the state. Simultaneously, the write is applied to an in-memory data structure called the Memtable, which temporarily holds recent writes in a sorted order. The Memtable is typically implemented using a balanced binary search tree (like a red-black tree or skip list), allowing for fast inserts, updates, and lookups in memory. The Memtable acts as a staging area for data before it is flushed to disk. Once the Memtable grows beyond a certain size threshold — which is configurable based on memory availability and workload characteristics — the database performs a flush operation. This involves freezing the current Memtable, sorting its contents (if not already sorted), and writing the data to disk as a Sorted String Table (SSTable). SSTables are immutable, disk-resident files that store key-value pairs in a sorted format, enabling efficient range queries and reducing the need for frequent random disk I/O. This pipeline, WAL for durability, Memtable for fast in-memory buffering, and SSTables for persistent storage, is central to how LSM-based databases achieve high write throughput. Over time, as more SSTables accumulate on disk, background processes perform compactions to merge and reorganize SSTables, reduce storage overhead, and maintain read efficiency.

A diagram of a flowchart

AI-generated content may be incorrect.

Figure 8 Representation of a Read operation in LSM Tree with Bloom filter

When a read request is made in an LSM tree-based database, the system first attempts to find the requested key in the in-memory Memtable, where the most recent writes are stored. If the key is present, the corresponding value can be returned almost instantly, taking advantage of the high-speed memory access. This in-memory lookup is extremely efficient and is typically the fastest possible retrieval path in the database. However, if the key is not found in the Memtable, the database then proceeds to search the on-disk SSTables, starting from Level 0 and moving deeper through Level 1, Level 2, and so on. These levels represent increasingly older data, and the SSTables within them are typically larger and more compacted as the level number increases. Since each SSTable is sorted, a binary or logarithmic search can be used within a file to look for a key. However, even with sorted data, reading from disk is still significantly slower than accessing memory, especially when many SSTables must be examined. To optimize read performance, LSM tree-based systems incorporate Bloom filters — a probabilistic, space-efficient data structure designed to quickly test whether a key might exist in a dataset. While Bloom filters can occasionally yield false positives, they never produce false negatives, which means they can safely rule out the absence of a key in each SSTable or Memtable. By implementing Bloom filters at each level of the LSM tree, the system can first check whether the key possibly exists in any SSTable at a particular level. If the Bloom filter indicates that the key does not exist, the system can skip reading that file entirely, saving valuable disk I/O operations. This significantly reduces the number of SSTables that need to be scanned, making reads faster and more efficient overall. Interestingly, Bloom filters are also applied to the Memtable, even though it resides in memory. This is because checking a Bloom filter is often faster than traversing a balanced tree (like a skip list or red-black tree), especially for large in-memory structures. As a result, Bloom filters play a vital role in making read operations in LSM tree-based databases both scalable and performant, even under heavy workloads or large data volumes.

MyRocks, developed by Meta (Facebook), is a pluggable storage engine for MySQL built on top of RocksDB. It was designed to address the inefficiencies of B+ Trees in write-heavy workloads, focusing on optimizing write performance, space efficiency, and effective utilization of modern storage hardware such as SSDs and NVMe drives through background compaction and reduced write amplification. MyRocks integrates seamlessly with MySQL by leveraging the pluggable storage engine API, allowing it to reuse the MySQL SQL layer while introducing a completely different storage engine underneath. It is officially supported in Percona Server for MySQL 5.7, a community-driven fork known for performance and enterprise-grade features. To enable MyRocks in MySQL, it must be manually loaded during server startup using the following argument *--plugin-load-add=ha\_rocksdb.so*. This loads the ha\_rocksdb.so shared object for RocksDB into Percona Server.

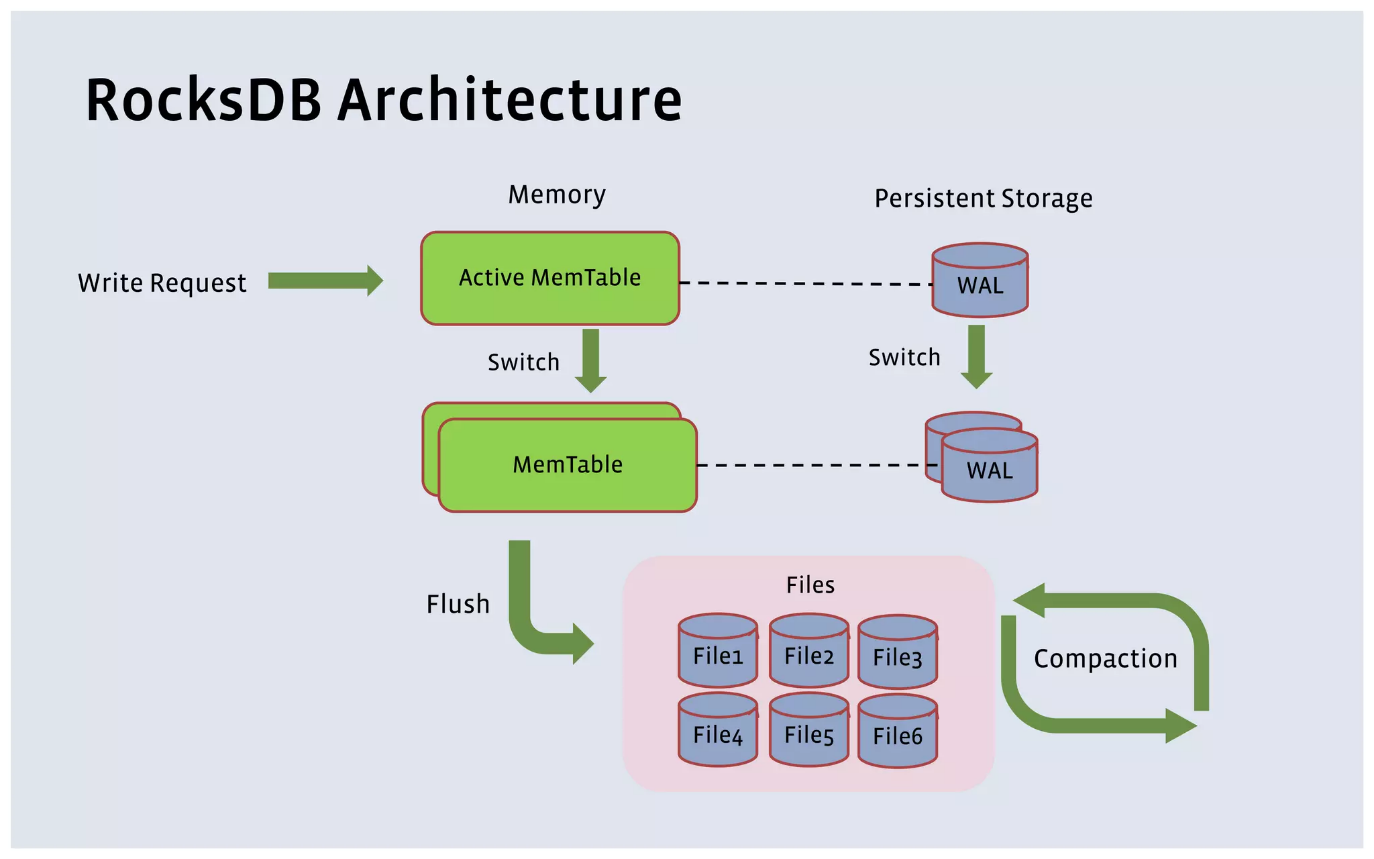


Figure 9 RocksDB Architecture Diagram

LSM Trees are optimized for large datasets that can handle from Gigabytes to Petabytes with billions of rows and hundreds of columns. These are some of the advantages of using MyRocks inside MySQL, but one big downside is mixing up the foreign keys are not allowed (not supported), which means one table uses InnoDB and another uses RocksDB and linking both using a foreign key is not allowed, as it is mentioned in official documentation that transactions cannot happen in multiple storage engines. MySQL, with its pluggable engine architecture, remains a versatile RDBMS for many workloads. The default InnoDB engine serves general-purpose applications well, offering transactional support and B+ Tree indexing. However, by leveraging the same SQL interface, MySQL allows for innovations like MyRocks, which swaps the underlying storage with an LSM-tree-based engine optimized for modern high-write, SSD-backed applications. This modularity is what allows Percona Server to support both B+ Tree and LSM Tree engines within the same MySQL binary, offering a unique opportunity to compare the performance of both paradigms under identical workloads, a crucial feature leveraged in this research.

Pebble is a high-performance key-value storage engine implemented in Go by Cockroach Labs which uses LSM tree in its storage layer. Designed as a compatible subset of RocksDB, it was introduced as an alternative to RocksDB to eliminate dependency on C++ and simplify integration with CockroachDB’s primarily Go-based codebase. Pebble became the default storage engine for CockroachDB starting with version v20.2, having been introduced in v20.1. Internally, Pebble inherits core LSM Tree concepts such as skiplist-based memtables, SSTable-based persisted storage, leveled compaction, merge operators, range deletion tombstones, and table-level Bloom filters. While Pebble supports RocksDB’s file formats for interoperability, it intentionally omits features beyond CockroachDB’s needs—for instance, it does not implement column families, backup or restore, universal compaction style, or some SSTable formats. Pebble offers several improvements over RocksDB such as faster reverse iteration, better concurrency via an optimized commit pipeline, seamless iteration over indexed batches, and enhanced compaction performance under heavy write load through sublevels and flush splitting. Its modular and Go-native codebase makes maintenance and enhancement easier than working with a large C++ dependency.

LSM Trees are widely adopted in write-heavy, large-scale systems including RocksDB, LevelDB, Cassandra, HBase, CockroachDB, and more. They excel in scenarios such as log ingestion, time-series storage, and high-throughput key-value workloads due to their ability to convert random writes into efficient sequential writes, reducing stress on SSDs and maximizing throughput. A significant evolution in LSM Tree design is the concept of separating keys and values, as introduced by the WiscKey architecture. In WiscKey, the LSM Tree only stores keys and pointers to values, while the actual value data is stored separately in a large append-only log (vLog). This design dramatically reduces write amplification because compacting SSTables involves only the smaller keys, not the full values. It improves SSD performance and extends device lifespan by minimizing unnecessary rewrites. Key-value separation introduces challenges, particularly in range queries (where value lookups become random) and in maintaining consistency and garbage collection across separate key and value stores, but WiscKey addresses these using SSD parallelism and an online garbage collector that avoids blocking foreground operations. Taken together, LSM Trees and their variants provide a powerful indexing mechanism for write-heavy workloads, at the cost of more complex read paths and compaction overhead. Their design complements modern storage technologies and scales effectively, making them an essential alternative to B‑Tree–based indexing structures in high-throughput, distributed, and SSD-oriented environments.

This literature review has presented a comprehensive exploration of two foundational storage engine architectures: B-Trees and Log-Structured Merge Trees (LSM Trees). B-Trees, introduced in the 1970s, remain a dominant indexing mechanism in traditional relational databases like MySQL (InnoDB) and PostgreSQL due to their balanced structure, efficient range queries, and predictable update performance. However, they face challenges with write amplification and random I/O operations, especially in write-heavy or SSD-optimized systems. To address these limitations, LSM Trees emerged as a compelling alternative. They organize writes sequentially using in-memory structures like skip lists (memtables) and immutable on-disk Sorted String Tables (SSTables), drastically improving write throughput. Modern implementations, such as RocksDB and its Go-based counterpart PebbleDB, further refine LSM Tree architectures with features like leveled compaction, Bloom filters, and key-value separation to optimize for SSD performance. PebbleDB, in particular, stands out as a production-grade, RocksDB-compatible engine integral to CockroachDB, offering high concurrency and efficient resource utilization. Through this comparative analysis, it is evident that each indexing approach brings distinct trade-offs. B-Trees excel in read-heavy workloads with frequent range scans, whereas LSM Trees outperform in high-throughput write scenarios and are better suited for modern storage hardware. This foundational understanding of their internal mechanics, use cases, and real-world implementations sets the stage for further investigation into hybrid or adaptive systems that can dynamically leverage the strengths of both structures depending on workload characteristics.

# Methodology

Modern applications, especially data-intensive systems, demand storage engines that can handle diverse and unpredictable workloads efficiently. In such systems, B+ Trees and Log-Structured Merge (LSM) Trees have become the foundational indexing structures due to their performance characteristics. B+ Trees are well-suited for read-heavy workloads and support range queries efficiently, whereas LSM Trees are optimized for write-heavy operations due to their sequential write behavior and compaction strategies. Traditional relational databases typically rely on a single storage engine optimized for general-purpose performance. However, this one-size-fits-all approach can fall short in addressing the nuanced performance needs of modern applications. Today, applications often resort to polyglot persistence - using different databases or storage engines for different components of their architecture. This separation of concern introduces complexity in maintenance, consistency, and integration. A promising alternative is the concept of a hybrid or multi-engine architecture within a single database, where both B+ Trees and LSM Trees coexist and dynamically switch based on workload characteristics. This research aims to explore the feasibility and performance impact of such a hybrid indexing strategy. The primary objective is to understand whether dynamically switching between B+ Tree and LSM Tree structures, based on query patterns, can yield performance gains without adding unnecessary complexity or overhead.

This research adopts an experimental methodology. It does not attempt to simulate or replicate a full-scale production deployment. Instead, the focus is purely on benchmarking engine-level performance under controlled and consistent conditions. To isolate indexing performance, only core operations—reads (SELECTs) and writes (INSERTs)—are evaluated. Operations like UPDATE and DELETE often follow similar access paths as inserts and hence exhibit comparable performance trends in most engines. Modern applications often use secondary indexes to improve lookup speed on non-primary key columns such as user emails or product names. Accordingly, indexes are applied to frequently queried fields in the datasets. Benchmark queries include point lookups, multi-table joins, and aggregate functions, simulating the workload patterns seen in real-world systems. These tests aim to stress the indexing structures and assess their response under varied query types, rather than evaluating full database functionality such as transactions or concurrency control. By focusing solely on indexing engine performance under different workloads, this research offers a meaningful comparison of how B+ Tree and LSM Tree structures behave independently, and whether hybrid approaches can deliver balanced, optimal performance.

To conduct a fair and controlled comparison of indexing performance between B+ Trees and LSM Trees, this research uses three distinct relational database systems, each chosen for their support of specific storage engines:

## Comparison of PostgreSQL, MySQL (InnoDB & MyRocks), and CockroachDB (PebbleDB)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | PostgreSQL (B+ Tree) | MySQL (InnoDB) | MySQL (MyRocks) | CockroachDB (PebbleDB) |
| Release Year | 1996 | 1995 | 2016 Forked MySQL | 2015, PebbleDB in 2020 |
| Storage Engine | B+ Tree | B+ Tree | LSM Tree | LSM Tree |
| Programming Language | C | C/C++ | C++ | Go |
| Pluggable Storage Engine | No | Yes | Yes | No |
| Transactional Support | Full ACID | Full ACID | Yes (with some limitations) | Full ACID with distributed transactions |
| Write Performance | Moderate | Moderate | Very Fast | Fast (GC allocation limitations) |
| Read Performance | High (especially random reads) | High | Moderate | Moderate |
| Data Compression | Yes | Yes (Enabled by Default) | Better compression | Integrated in Pebble (zstd/snappy) |
| SQL Compatibility | PostgreSQL standard | MySQL standard | MySQL standard | PostgreSQL-compatible |
| Distributed | No | No | No | Yes (Native Support) |

Table 1 Comparison b/w PostgreSQL, InnoDB, MyRocks and Pebble Storage Engine

## Data Generation and Preprocessing

In this experimental research, synthetic data is used instead of real-world datasets. This decision stems primarily from legal and ethical considerations, especially under strict data privacy regulations such as the UK General Data Protection Regulation (UK-GDPR). Using real user data, even in anonymized form, may introduce potential legal liabilities, consent issues, and data handling complexities. To circumvent these concerns while ensuring flexibility in dataset design, fake data was generated programmatically. To achieve this, the Faker library (Python) was used as the core data generation tool. Faker is open source, released under the MIT License, which explicitly permits free usage, modification, and distribution, including for academic and commercial purposes. This makes it a legally safe and efficient choice for simulating realistic datasets. The simulated dataset emulates an e-commerce application operated by a Small or Medium Enterprise (SME). The rationale behind choosing an SME-scale setup is twofold:

* Cost and Efficiency: Large-scale enterprise datasets demand significantly more storage, processing power, and time. For an experimental research context with limited infrastructure (Docker environments with constrained resources), an SME-level dataset strikes a balance between realism and manageability.
* Representative of Modern Workloads: Many modern applications, including startups, mid-scale SaaS platforms, and mobile commerce apps, operate at SME scale and often rely on single-node or small-cluster databases. Testing indexing strategies in this environment makes the results more applicable to such use cases.

The dataset was designed to reflect a typical e-commerce domain, consisting of interrelated entities such as Users, Products, Category, Orders, Order Entry, Transactions, Reviews.

**Users Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| Id | Int | Unique Identifier for User |
| first\_name | varchar | First name of the User |
| last\_name | varchar | Last name of the User |
| email | varchar | Email Address of the User |
| password | varchar | Hashed Password of the User |
| picture | text | Profile picture URL |
| address | varchar | Home Address of the User |
| phone | varchar | Phone Number with Country code of the User |
| notifications\_allowed | boolean | Whether the User opted to receive notifications. |
| date\_joined | datetime | Date and Time when the user details inserted into the system |

Table 2 Users Entity

**Products Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| product\_id | int | Unique Identifier of the Product |
| product\_name | varchar | Name of the Product |
| product\_description | varchar | Short Description of the Product |
| product\_image | text | Image URL of the Product |
| price | decimal | Original Price of the Product |
| discount\_percent | smallint | A 2-digit percentage which represents discount in percentage |
| category | int | To which category this product belongs, and it references category table |
| date\_created | datetime | Date and Time when the product details stored into the system |

Table 3 Products Entity

**Category Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| id | int | Unique Identifier of the Category |
| category | varchar | Name of the category |

Table 4 Category Entity

**Orders Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| product\_id | UUID | Unique Identifier of the Order |
| total\_price | decimal | Total price before discount |
| discounted\_price | decimal | Total price after discount applied |
| user\_id | int | User ID who owns the order |
| date\_created | datetime | Date and Time when the order stored into the system |

Table 5 Orders Entity

**Order Entry**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| order\_entry\_id | uuid | Unique Identifier of the Order Entry |
| order\_id | uuid | Order which is associated to |
| product\_id | int | Product which is associated to |
| price\_per\_unit | decimal | What was the total price per unit |
| quantity | int | Total quantity |
| total\_price | decimal | Total price with all quantity |
| discounted\_price | decimal | Total price after discount |
| discount\_percent | smallint | A 2-digit percentage which represents discount in percentage |

Table 6 Order Entry Entity

**Transactions Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| transaction\_id | uuid | Unique Identifier of a Transaction |
| order\_id | uuid | Order which is associated to |
| user\_id | int | User who did the transaction |
| amount | decimal | Total amount paid |
| payment\_method | enum | Payment method the user did with |
| status | enum | Payment Status |
| date | datetime | Date and Time when the transaction happened |

Table 7 Transactions Entity

**Reviews Entity**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| review\_id | uuid | Unique Identifier of a Review |
| user\_id | int | User who posted review |
| product\_id | int | Product to which the review posted on |
| rating | smallint | Rating given by user on scale of 5 |
| review\_text | text | Review text given by user |

**Entity Relationship Diagram**

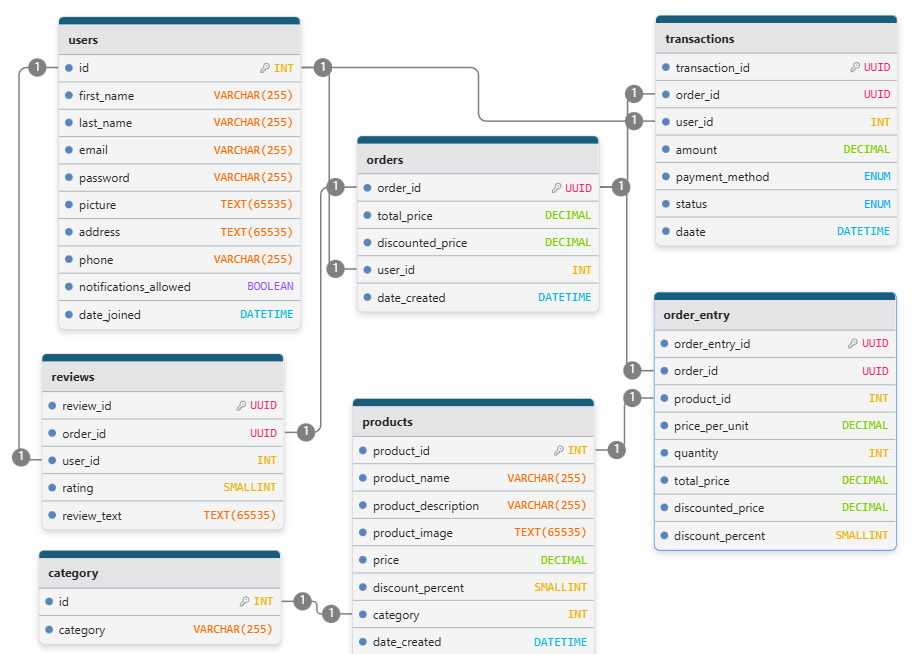


Figure 10 Entity Relationship Diagram of all Entities with Relationship

## Data Loading Process

In the data loading phase, tables were created in the database schema without including foreign key constraints during the initial creation. While the necessary columns for maintaining relational integrity were present, the foreign key relationships themselves were deliberately excluded to simplify the bulk data import process and to avoid potential constraint violations or performance overheads during the insertion stage. The dataset, previously generated and exported into individual CSV files, was then imported into the database using TablePlus, a modern SQL editor and data visualization tool that supports seamless data import/export operations. Each CSV file was loaded into its corresponding table carefully, ensuring the mapping of columns was consistent. After the data import, a validation step was carried out where each table was checked for completeness and correctness using the SQL command:

SELECT count(\*) FROM <TABLE\_NAME>;

This provided a quick verification of the total number of rows per table to ensure that the expected number of records had been successfully loaded without any data corruption or missing entries. Once the data was successfully loaded into the database, foreign key constraints were added post-import to enforce relational integrity across the schema. This approach helped avoid performance bottlenecks and constraint-related errors during the initial bulk data load. The constraints were added using the ALTER TABLE SQL command, following the syntax:

ALTER TABLE TABLE\_NAME

ADD CONSTRAINT CONSTRAINT\_NAME

FOREIGN KEY (FIELD)

REFERENCES FOREIGN\_TABLE\_NAME (FOREIGN\_FIELD);

Each foreign key was defined carefully to match the intended relational design of the e-commerce schema. After adding the constraints, simple JOIN queries were executed between related tables to verify that the relationships were enforced correctly and that data integrity was maintained. This post-validation step ensured that the schema now reflected a fully normalized and referentially consistent dataset, ready for benchmarking across different database engines.

To further improve query performance, especially during benchmarking, indexes were created on frequently accessed fields using the following SQL command:

CREATE INDEX idx\_index\_name ON TABLE (FIELD);

Creating indexes at this stage, after the bulk insert phase, allowed maximum insert throughput during the initial load while still enabling faster lookups and join operations during the evaluation phase. This step finalized the data preparation process, resulting in a well-structured and performance-optimized dataset suitable for running comparative benchmarks across different database systems.

## Workload Scope and Benchmark Focus

The primary focus of this benchmarking study is to evaluate the performance of database storage engines, particularly examining their behavior under read and write-heavy workloads. The scope of the benchmark is deliberately limited to *SELECT* and *INSERT* operations, as they represent the majority of activity in typical Online Transaction Processing (OLTP) applications. Although *UPDATE* and *DELETE* queries were initially considered, they were excluded from the final testing phase due to their similar performance characteristics to inserts and the added complexity in designing controlled benchmarks for such operations.

**Read Performance (SELECT Queries)**

To comprehensively assess the read efficiency, a variety of SELECT queries were executed, including:

* Point lookups (e.g., querying a specific user or order by primary key)
* Range queries (e.g., retrieving all orders within a date range)
* Aggregations (e.g., calculating total sales per user or product category)
* Multi-table joins (e.g., joining users, orders, and transactions for full purchase history)

This diversity of queries helps simulate real-world application patterns and provides insights into the read latency, query execution time, and the effectiveness of indexes and caching mechanisms employed by the storage engine.

**Write Performance (INSERT Queries)**

The write performance tests focus on bulk insert operations, where data is inserted in chunks of 10,000 records using Python's psycopg2 library for PostgreSQL and compatible interfaces for other engines. A dedicated dataset was generated for this purpose to avoid overlap with data used for read queries. The following tables were used during the write performance benchmarks: users, orders, order\_entry, transactions. Each write query was designed to emulate OLTP-like operations common in e-commerce and financial systems. These operations simulate high concurrency insert workloads with realistic data distributions and field types.

Performance metrics collected during these tests include:

* CPU utilization
* RAM consumption
* Disk I/O activity
* Network throughput
* Requests per second (RPS)
* Overall Latency

These metrics provide a quantitative basis for comparing storage engines under high-throughput conditions and evaluating their suitability for real-world deployments involving write-intensive workloads.

## Benchmarking Setup: Deployment, Monitoring, and Tooling

To ensure consistency, control, and repeatability across all database performance tests, the entire benchmarking infrastructure was containerized using Docker, with orchestration handled via Docker Compose. This setup allowed multiple database systems and monitoring tools to be spun up simultaneously in isolated environments, each configured with specific resource constraints to simulate realistic deployment conditions. Each database was run in its own container, with dedicated persistent volumes mounted from the host machine to ensure that data remained intact across container restarts or failures. The images used were the following:

* PostgreSQL was deployed using *postgres:bullseye*, which provides the latest stable release at the time of testing.
* CockroachDB was deployed using the official *cockroachdb/cockroach:v25.2.2* image.
* MyRocks (via MySQL) was deployed using *percona/percona-server:5.7*, a version that supports the RocksDB-based MyRocks engine.

These containers exposed their database ports (e.g., 26257 for CockroachDB, 5432 for PostgreSQL, and 3306 for MyRocks) to the host machine, allowing external connection using tools like TablePlus for manual inspection, and programmatic access via scripts using Python (e.g., through psycopg2 for PostgreSQL) and JMeter.

To monitor the system-level metrics, Google’s cAdvisor (Container Advisor) was used as the core agent. cAdvisor runs as a container itself and exports real-time performance metrics for all containers on the host system. It collects various metrics such as CPU usage, Memory usage, Disk I/O, Network traffic, Container metadata and more.

These metrics were scraped at regular intervals using Prometheus, an open-source time-series database designed for handling metric data. Prometheus was configured with appropriate scrape\_interval value to balance between resolution and performance overhead. All data collected was stored on a persistent Prometheus volume, enabling historical metric analysis and preventing data loss between test runs.

To visualize and analyze these metrics, Grafana was employed as the front-end dashboarding tool. Grafana was configured to connect directly to Prometheus as its data source. Custom dashboards were created to track real-time metrics per container, such as CPU usage, memory spikes, disk I/O, and network throughput. These dashboards provided visual cues and validation to interpret the performance behaviours of each database under different workloads. This full-stack monitoring pipeline, cAdvisor → Prometheus → Grafana — allowed for deep, real-time visibility into resource usage. More importantly, it ensured that benchmarking was not solely reliant on query timings or internal metrics from the database engines but also backed by system-level telemetry that reflected true cost and performance.

In addition to local containerized testing, the benchmarking infrastructure was also deployed on the cloud using Amazon Web Services (AWS). This enabled real-world performance testing in a distributed environment with variable latency and more scalable resources, further validating the findings observed in the local setup. For this, Amazon ECS (Elastic Container Service) was selected over traditional EC2-based deployment models. Using ECS eliminated the need to manually install and configure database servers on raw EC2 instances. Instead, container images were built and pushed to Amazon Elastic Container Registry (ECR) or pulled directly from public Docker registries. These images were then deployed as services within ECS Fargate launch type, which abstracts away the server management entirely and offers a serverless experience for container deployment.

Each database (PostgreSQL, MyRocks, and CockroachDB) was deployed in separate ECS tasks, each with dedicated CPU and memory limits defined in the task definitions. This ensured resource isolation across databases and enabled better control when benchmarking under constrained cloud environments. For network access, each ECS task was launched within a public subnet of a VPC (Virtual Private Cloud) and assigned a public IP address. This made the databases directly accessible over the internet, which was necessary for benchmarking tools running outside AWS or from local environments.

Security Groups were configured as firewalls to restrict inbound and outbound traffic. Only essential ports were exposed for external access:

* PostgreSQL → Port 5432
* MyRocks (MySQL) → Port 3306
* CockroachDB → Port 26257

All other ports remained closed, and access was restricted to specific IP addresses and regions for security. These services were deployed in the London region (eu-west-2) to reduce latency from the local testing environment.

For local benchmarking, Docker Compose was used to deploy PostgreSQL, MyRocks (MySQL), and CockroachDB with resource constraints and host-mounted volumes for data persistence. Metrics were collected using cAdvisor, stored in Prometheus, and visualized in Grafana for monitoring CPU, RAM, disk I/O, and network usage.

For cloud deployment, the same database containers were hosted on AWS ECS in the London region, allowing scalable and managed deployment without manual setup. Public IPs and restricted security groups were used to control access. This hybrid setup provided a balance between controlled local testing and realistic cloud performance evaluation.

# Implementation

This section outlines the complete technical setup used to conduct benchmarking experiments on different database engines. It covers the local and cloud-based environments used to deploy and manage the databases, as well as the tooling and techniques adopted to monitor, simulate, and measure performance metrics. The primary goal of this phase was to build a reproducible and resource-constrained infrastructure that allows fair comparison between B+ Tree and LSM Tree-based storage engines under realistic workloads.

To achieve this, a range of tools and technologies were used across the stack. Python and the Faker library were used for generating realistic synthetic datasets. The databases tested include PostgreSQL, Percona Server (MyRocks engine), and CockroachDB, each deployed using Docker containers managed with Docker Compose. For metrics collection and observability, Google’s cAdvisor was used to expose container-level metrics in Prometheus format, while Prometheus served as the time-series database to store these metrics. Grafana was used to create real-time dashboards and visualizations. Load generation and stress testing were conducted using Apache JMeter to simulate concurrent queries. For cloud-based experiments, Amazon Web Services (AWS) was used, specifically AWS ECS (Elastic Container Service), to deploy the database containers in a scalable and isolated manner with access controlled via security groups. This setup ensures consistent data collection and performance monitoring for a comprehensive evaluation of indexing strategies across different database engines.

## Data Generation Using Python and Faker

To simulate realistic workloads for benchmarking different storage engines, a synthetic dataset was generated using the Python faker library. The aim was to mimic typical Online Transaction Processing (OLTP) scenarios such as user registrations, orders, and transactions. Python was chosen for its scripting ease and speed, and faker proved highly suitable due to its ability to generate large-scale, randomized, and human-like data (names, addresses, emails, timestamps, etc.).

The generated data was stored in CSV files, which were later used to bulk load data into the target databases. The script was designed to be generalized, allowing the developer to plug in the desired column schema, target file name, and the size of the dataset to be generated.

Below is a simplified version of the script used to generate the users dataset:

import faker

import random

import csv

import os

fakerGen = faker.Faker()

columns = ['first\_name', 'last\_name', 'email', 'phone']

filename = "users.csv"

fileExist = os.path.isfile(filename)

with open(filename, "a", newline='') as csvFile:

    writer = csv.DictWriter(csvFile, fieldnames=columns)

    if not fileExist:

        writer.writeheader()

    for count in range(100):  # Generates 1 million rows (100 x 10,000)

        dataList = []

        for \_ in range(10\_000):

            data = {

                'first\_name': fakerGen.first\_name(),

                'last\_name': fakerGen.last\_name(),

                'email': fakerGen.email(),

                'phone': fakerGen.phone\_number()

            }

            dataList.append(data)

        writer.writerows(dataList)

        print(f"Bulk {count + 1} inserted")

The script structure allows it to be reused for different datasets (e.g., orders, transactions, products) by simply modifying the columns, file name, and data fields. Each CSV file generated contained between 1 to 5 million records, depending on the target table. This method ensured a consistent, scalable, and reproducible dataset for benchmarking across all tested storage engines.

## Local Deployment with Docker

To facilitate repeatable and isolated benchmarking, all required components for the testing environment were deployed locally using Docker Compose. This approach allowed for quick provisioning, controlled resource allocation, and persistence of data across container restarts. The environment consisted of six main services — three database containers, three monitoring stack components, all connected via a single user-defined bridge network.

Below is the breakdown of each service and its role in the benchmarking setup:

* Grafana
  + Image: grafana/grafana
  + Purpose: Grafana acted as the main visualization layer for all metrics collected during the tests. It queried Prometheus as a data source and displayed real-time graphs for CPU usage, memory consumption, disk I/O, and network throughput of the database containers.
  + Configuration:
    - Port Mapping: 3000:3000 made the Grafana dashboard accessible via <http://localhost:3000>.
    - Volume: Docker volume is mounted on ./grafana\_data:/var/lib/grafana to persist the dashboard configurations, so they remained available even if the container was restarted.
    - Dependency: Waited for the Prometheus container to start before launching.
* Prometheus
  + Image: prom/Prometheus
  + Purpose: Collected and stored time-series metrics from cAdvisor and exposed them for querying and visualization.
  + Configuration:
    - Port Mapping: 9090:9090 provided access to Prometheus’s UI at <http://localhost:9090>.
    - Volume:
      * ./prometheus.yml:/etc/prometheus/prometheus.yml:ro mounts the Prometheus configuration file in read-only mode.
      * ./prometheus\_data:/prometheus ensured all historical metric data was stored persistently.
    - Dependency: Depended on cAdvisor for metric collection.
* cAdvisor (Container Advisor)
  + Image: gcr.io/cadvisor/cadvisor:latest
  + Purpose: Monitored and exposed performance metrics for all running containers on the host machine, such as CPU usage, memory consumption, disk statistics, and network activity.
  + Configuration:
    - Port Mapping: 8080:8080 exposed cAdvisor’s UI for raw metric inspection.
    - Volumes: Mounted key host system directories to allow cAdvisor to gather low-level container metrics:
      * /rootfs for root filesystem access.
      * /var/run for Docker runtime socket.
      * /sys for system information.
      * /var/lib/docker for container-level data.
* MyRocks (Percona Server with RocksDB & InnoDB Engine)
  + Image: percona/percona-server:5.7
  + Purpose: Deployed MySQL with MyRocks storage engine enabled, serving as the LSM-tree based storage engine under test.
  + Configuration:
    - Port Mapping: 3306:3306 allowed connections from the host machine’s MySQL clients.
    - Environment Variable: MYSQL\_ROOT\_PASSWORD=root set the root account password.
    - Command: *--plugin-load-add=ha\_rocksdb.so* loaded the RocksDB storage engine plugin.
    - Volume: ./rocksdb\_data:/var/lib/mysql persisted MyRocks database files.
    - Resource Constraints: Limited to 1 CPU core, 1GB of RAM, and 1GB of shared memory for fair benchmarking.
* PostgreSQL
  + Image: postgres:bullseye
  + Purpose: Served as the baseline B-Tree based storage engine for benchmarking comparisons.
  + Configuration:
    - Port Mapping: 5432:5432 allowed PostgreSQL client connections from the host.
    - Environment Variable: POSTGRES\_PASSWORD=postgres set the default superuser password.
    - Volume: ./postgres\_data:/var/lib/postgresql/data ensured persistent storage.
    - Resource Constraints: Same as MyRocks for consistency — 1 CPU, 1GB RAM, 1GB shared memory.
* CockroachDB
  + Image: cockroachdb/cockroach:v25.2.2
  + Purpose: Provided a modern, distributed SQL database for additional performance benchmarking.
  + Configuration:
    - Port Mapping: 26257:26257 exposed the SQL port to the host.
    - Command: start-single-node --insecure launched the database in single-node development mode without TLS encryption.
    - Volume: ./cockroachdb\_data:/cockroach/cockroach-data persisted all database files.
    - Resource Constraints: Same limits as MyRocks and PostgreSQL.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure 11 Architecture of Containers running in Docker

**Networking**

All services were connected through a custom Docker bridge network named *net*. This allowed inter-container communication using service names as hostnames (e.g., prometheus could scrape metrics from cadvisor via http://cadvisor:8080). The bridge network ensured isolation from other Docker networks on the host machine.

The below code represents the docker-compose file which used to run all the containers. The command used to run this is *docker-compose up -d* which runs in detached mode and doesn’t block the CLI.

services:

  grafana:

    image: grafana/grafana

    container\_name: grafana\_bench

    networks:

      - net

    ports:

      - 3000:3000

    volumes:

      - ./grafana\_data:/var/lib/grafana

    depends\_on:

      - prometheus

  prometheus:

    image: prom/prometheus

    container\_name: prometheus

    networks:

      - net

    ports:

      - 9090:9090

    volumes:

      - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro

      - ./prometheus\_data:/prometheus

    depends\_on:

      - cadvisor

  cadvisor:

    image: gcr.io/cadvisor/cadvisor:latest

    container\_name: cadvisor

    networks:

      - net

    ports:

      - 8080:8080

    volumes:

      - /:/rootfs:ro

      - /var/run:/var/run:rw

      - /sys:/sys:ro

      - /var/lib/docker:/var/lib/docker:ro

  myrocks:

    image: percona/percona-server:5.7

    container\_name: myrocks\_btree\_and\_lsmtree

    ports:

      - 3306:3306

    environment:

      - MYSQL\_ROOT\_PASSWORD=root

    command: --plugin-load-add=ha\_rocksdb.so

    volumes:

      - ./rocksdb\_data:/var/lib/mysql

    cpus: 1

    mem\_limit: 1G

    shm\_size: 1G

    networks:

      - net

    restart: always

  postgres:

    image: postgres:bullseye

    container\_name: postgres\_btree

    ports:

      - 5432:5432

    environment:

      - POSTGRES\_PASSWORD=postgres

    volumes:

      - ./postgres\_data:/var/lib/postgresql/data

    cpus: 1

    mem\_limit: 1G

    shm\_size: 1G

    restart: always

    networks:

      - net

  cockroachdb:

    image: cockroachdb/cockroach:v25.2.2

    container\_name: cockroachdb

    ports:

      - 26257:26257

    command: start-single-node --insecure

    volumes:

      - ./cockroachdb\_data:/cockroach/cockroach-data

    cpus: 1

    mem\_limit: 1G

    shm\_size: 1G

    restart: always

    networks:

      - net

networks:

  net:

    driver: bridge

This containerized setup provided:

* Reproducibility: Easily recreated the test environment with a single docker-compose up.
* Persistence: Database and monitoring data survived restarts.
* Resource Control: Ensured fair benchmarking by applying identical CPU and memory limits to database containers.
* Centralized Monitoring: Grafana, Prometheus, and cAdvisor worked together to capture and visualize the system’s performance under different workloads.

## Cloud Deployment Using AWS

This section describes the process of deploying benchmark database containers in a cloud environment using Amazon Web Services Elastic Container Service (AWS ECS). ECS was chosen for its managed container orchestration, allowing database instances to run without the manual overhead of installing and configuring dependencies on bare-metal EC2 servers. The deployment followed a structured process:

A screenshot of a video game

AI-generated content may be incorrect.

Figure 12 AWS Test Setup Architecture

* Setting the AWS Environment
  + Region Selection: Deployment was carried out in the London region (eu-west-2) to ensure low-latency connectivity for testing purposes.
  + Service Selection: Chose Amazon ECS from the AWS console.
* Creating the Task Definition
  + A task definition in ECS acts as the blueprint for running containers.
  + Key configurations made:
    - Name: Descriptive identifier for the task.
    - Launch Type: Fargate (serverless managed container service) selected for simplicity.
    - Operating System: Linux.
    - CPU Architecture: x86 selected for compatibility with database images.
    - Resource Allocation: 1 vCPU and 2 GB RAM per container.
    - IAM Role: Default ecsTaskExecutionRole since AWS API access was not required.
    - Container Details:
      * Name & Image: Image name provided directly (defaults to Docker Hub if not prefixed with repository URL).
      * Port Mapping: Configured to expose required database ports using TCP.
      * Environment Variables: Set only when necessary for database configurations.
      * Logging: Disabled CloudWatch logging to reduce cost; CPU and memory metrics were sufficient.
    - Storage: Allocated 21 GB ephemeral storage (minimum requirement). EBS volumes could also be attached if persistent storage was needed.
* Creating the ECS Cluster
  + The ECS cluster manages the lifecycle of containers.
  + Configuration included:
    - Cluster Name: Unique identifier for the deployment.
    - Infrastructure: Fargate for full AWS-managed execution.
    - Monitoring: Enabled basic metrics (CPU, memory, disk I/O, network I/O).
* Running a Container (Task Execution)
  + Navigate to the ECS cluster and open the Tasks tab.
  + Select Run New Task.
  + Choose the previously created Task Definition.
  + Set the Desired Task Count (set to 1 for continuous testing; can be set to 0 to pause until traffic arrives, a cost-saving feature in Fargate).
  + Configure Networking:
    - Selected appropriate VPC and subnets.
    - Assigned security groups to control inbound/outbound traffic.
    - Enabled Public IP for direct database access (alternatively, a Load Balancer can be used).
  + Optionally override CLI commands or entrypoint (not required here).
  + Launch the task and wait until its status changes from Pending to Running.