Faculty of Science and Engineering

Ali Ahsan, Gulwarina Muska Saleem, Momin Abdurrehman

Database Management Systems - Assignment 3

Database Indexing and Query Optimization in MySQL

May 2025

BS Data Science 2A

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose and Scope

Relational databases are the backbone of data-driven applications, supporting various industries such as online retail, banking, and healthcare. This report examines two foundational techniques for improving database efficiency—indexing and query optimization—within MySQL. It aims to:

1. Explain the theoretical mechanisms behind indexes and optimizer decision-making,
2. Demonstrate practical MySQL configurations and examples,
3. Contrast strategies in two industries with opposing workload profiles (e-commerce and banking).

By the end of this report, the reader should be able to answer both the "why" and the "how" of high-performance database design.

1.2 Indexing Importance and Query Optimization

An *index* in MySQL is a data structure, commonly a B-tree, that maps Key values to Row location, enabling lookups in Logarithmic time rather than full-table scans.(Silberschatz et al. 2019) Proper indexing can reduce Query latency by orders of magnitude, but each index adds storage and Write-maintenance overhead.(Oracle Corporation 2024a) Query optimization complements indexing by building cost-based execution plans that select the most efficient access

paths, join methods, and data-retrieval algorithms.(Chaiken and al. 2008)Together, indexing and optimization allow MySQL to serve large numbers of concurrent reads and writes without performance degradation or inconsistency.

1.3 Overview of Industry Applications

To illustrate how indexing and optimization adapt to different requirements, we compare:

E-Commerce (Read-Heavy)

- **Characteristics:** High volume of `SELECT` queries for product search, filtering, and browsing.
- **Indexing Focus:** Composite and full-text indexes on product attributes (e.g., `(category, product_name)`) to accelerate Range scans and keyword searches.(LoadForge 2023)
- **Optimization:** Use of `EXPLAIN` to verify index usage, covering indexes to avoid table lookups, and External caching (e.g., Redis) to offload repeated queries.(Acceldata 2022)

Banking (Write-Heavy)

- **Characteristics:** Intensive `INSERT`/`UPDATE` operations for transactions, with strict ACID requirements.
- **Indexing Focus:** Minimalist index set (primary and essential foreign keys) to reduce update overhead.(Percona 2022)
- **Optimization:** Batched writes, tuned InnoDB parameters (log-write frequency, log-buffer size), and appropriate Isolation levels.

1.4 Key Concepts and Terminology

1.4.1 Definitions

Index. A Index is a separate data structure that stores Key values–row pointer pairs, enabling the database engine to locate rows without scanning the entire table. In MySQL's InnoDB engine, indexes are implemented as B-tree index structures by default, providing Logarithmic time lookup for Equality query and Range query queries. While indexes dramatically improve

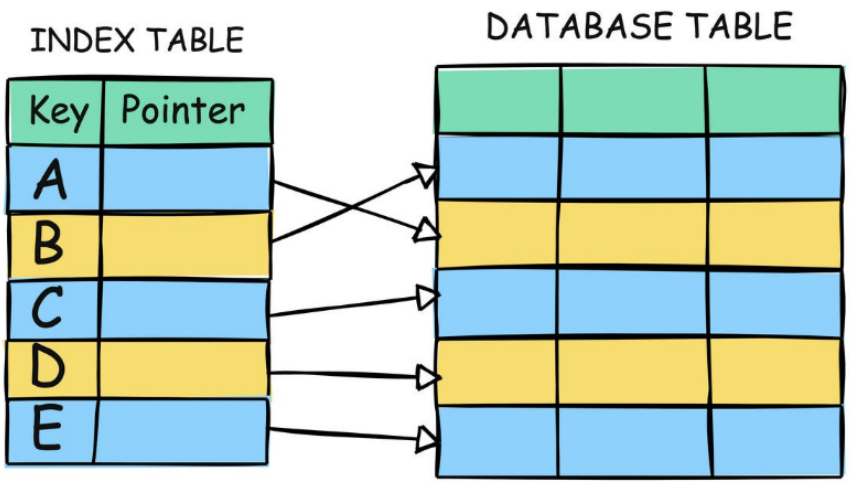


Figure 1.1: Indexing visualized

`SELECT` performance, each additional index incurs storage costs and slows down `INSERT`, `UPDATE`, and `DELETE` operations because the index structures must be maintained on writes.(Silberschatz et al. 2019; Oracle Corporation 2024a)

Query Execution Plan. When MySQL processes a SQL statement, it generates a Query Execution Plan, which shows the steps it will follow to get the requested data. This includes:

- the order in which tables are accessed,
- which Indexes (if any) are used,
- join algorithms,
- operations such as Sorting or Aggregation.

The `EXPLAIN` command reveals the execution plan, showing key metrics (e.g., rows, filtered, type) to help DBAs optimize performance.(Chaiken and al. 2008; Oracle Corporation 2024b)

B-Tree Index. A B-tree index stores sorted key values and pointers to child nodes or rows. InnoDB's B-tree indexes adjust automatically on inserts/deletes, keeping searches fast.(Silberschatz et al. 2019; Oracle Corporation 2024a)

Hash Index. A Hash index uses a Hash function to map key values into Buckets. MySQL supports hash indexes in the MEMORY engine, not InnoDB, though InnoDB maintains internal hash lookups for performance.(Oracle Corporation 2024b; Chaiken and al. 2008)

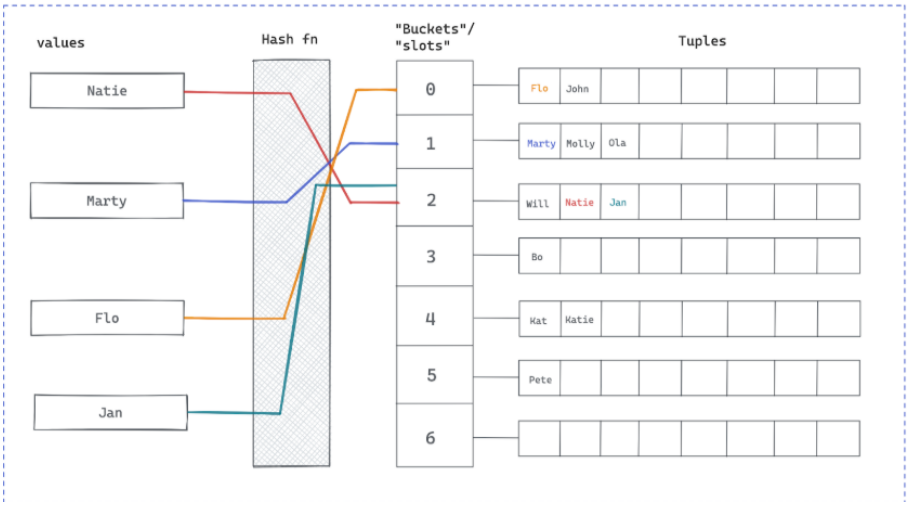


Figure 1.2: Hash Index

1.4.2 Relationship to Database Design Principles

Normalization. Normalization is the process of structuring a database to reduce redundancy and improve Data integrity by dividing data into related tables. Highly normalized schemas require joins, so Foreign key indexes become critical for performance.

Entities and Relationships. Entities represent real-world objects (e.g., Customer, Account), and relationships define how entities relate (e.g., a customer owns multiple accounts). Primary key uniquely identify each row in an entity table, while Foreign keys establish links between tables. Indexes on primary and foreign keys are important: primary key indexes make sure values are unique and allow fast searches, while foreign key indexes make join operations faster when linking related tables. Without good indexing on foreign keys, even simple joins can become slow and require scanning entire tables, which defeats the purpose of having a normalized database.

1.5 Types of Indexes

1.5.1 Single-Column Indexes

A single-column index covers one attribute:

How it works

MySQL builds a B-tree index on the indexed column. Lookups like

```
SELECT * FROM products WHERE category = 'Electronics';
```

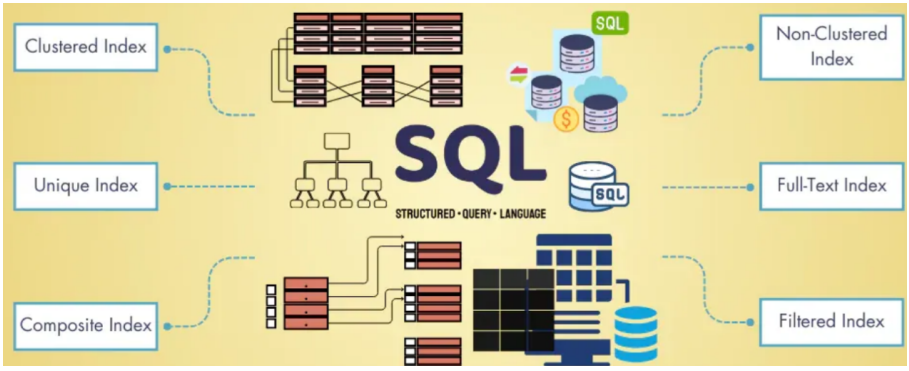


Figure 1.3: Different Types

use the Index to jump directly to matching rows instead of scanning all products.

E-commerce

Use case: Filtering by a single attribute, e.g. category, brand or price range. Benefit: Faster loading when users pick a category. Trade-offs: More indexes mean slower updates. With lots of daily changes, each Index adds extra work.

Banking

Use case: Speeding lookups by Primary key (transaction_id) or by account ID on transactional tables. Benefit: Very fast searches when looking up data by one field, like a specific transaction or account. Trade-offs: Writing new data gets slower with each Index. That's why banks only use the most needed single-column indexes.

1.5.2 Composite (Multi-Column) Indexes

Composite indexes span two or more columns in a defined order:

How it works

A B-tree index keyed on (col1, col2, . . .) supports queries that filter on the left-most prefix:

```
SELECT * FROM products
WHERE category='Electronics' AND price<500;
```

can use a combined Index on (category, price) to handle both conditions in a single index lookup, making the query faster with just one tree traversal.

E-commerce

Example: (category, product_name) lets searches for "Electronics" plus a keyword using one index rather than two separate scans. Benefit: Avoids merging two index results. Overhead:

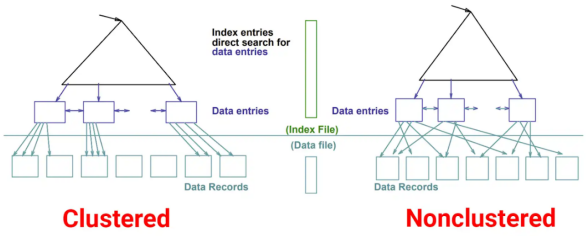


Figure 1.4: Difference between them

Inserting or updating values in indexed columns requires rebuilding the index entry. For frequently changing columns (like price during dynamic pricing), this increases the cost of writes.

Banking

Example: (account_id, transaction_date) to quickly retrieve an account's recent transactions in date order. Benefit: Enables efficient Range scans (e.g., last 30 days) without sorting. Overhead: Since transaction dates are added in order, maintaining the Index is cheaper than for Random keys. However, with thousands of daily writes, there's still some cost involved.

1.5.3 Clustered vs. Non-Clustered Indexes

InnoDB's Primary key is Clustered index; all other indexes are Non-clustered index:

Clustered (Primary Key)

Definition: The table's rows are physically stored in primary-key order. E-commerce: Clustering on product_id (often auto-increment) makes range queries on product IDs (e.g. "show newest products") very fast. Banking: Clustering on transaction_id (sequentially generated) favors Write append performance performance and new transactions are added at the end of the table file.

Non-Clustered (Secondary)

Definition: Secondary indexes store key + PK pointer to find the actual row. E-commerce: Secondary indexes on price or rating allow sorting and filtering without touching the full table. Banking: Secondary indexes (e.g., on account_id) let you find all an account's transactions, but each lookup requires a clustered Primary key lookup.

1.5.4 Full-Text Indexes

Full-text indexes build an inverted index of word→document mappings:

E-commerce

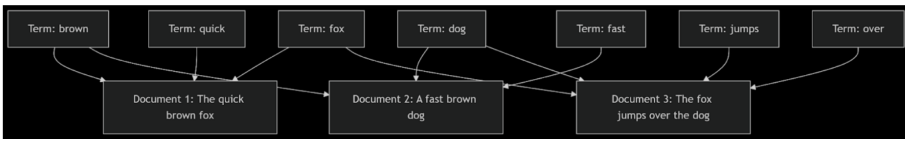


Figure 1.5: Full text index

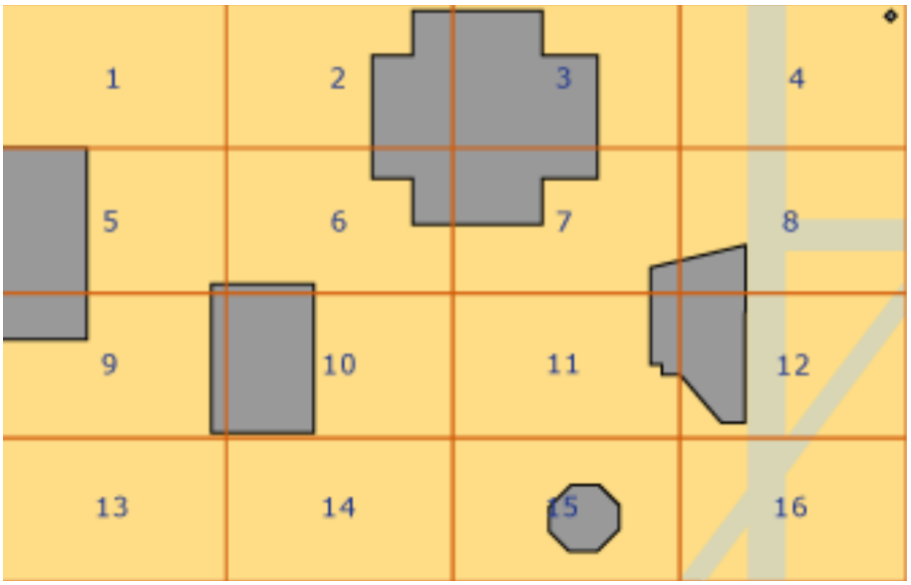


Figure 1.6: Spatial Indexes

Use case: Searching product titles and descriptions by keywords. Benefit: MySQL's FULLTEXT index returns ranked results; supports natural-language and Boolean modes. Example query:

```
SELECT * FROM products
 WHERE MATCH(description)
       AGAINST('+smartphone -refurbished' IN BOOLEAN MODE);
```

Trade-offs: Indexes for big text fields (like product descriptions) use a lot of compute power, but since these descriptions rarely update, the extra effort is worth it for faster searches.

Banking

Use case: Rarely used transaction notes and descriptions are typically short and infrequently searched. Recommendation: Avoid full-text indexes to minimize write overhead.

1.5.5 Spatial Indexes

Spatial indexes use R-trees for geometric data:

E-commerce

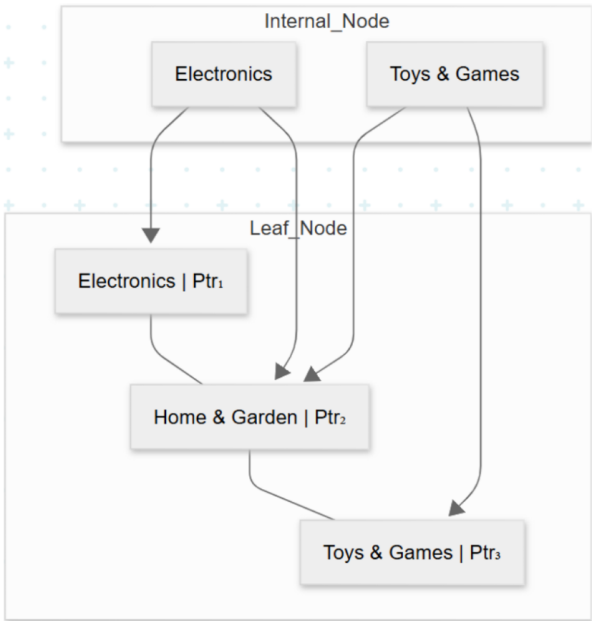


Figure 1.7: B-Tree node structure with sample e-commerce key values (e.g., product categories).

Example: Geo-filter searches like "warehouses within 50 km of user."

Banking

Use case: Checking store locations using map coordinates (spatial data) doesn't require many searches, so it's easy for the database to handle without slowing down. Recommendation: Only add if your application actually performs spatial operations on transactional data.

1.5.6 Summary Table of Index Characteristics

Table 1.1: Read/write trade-offs and use cases for MySQL index types

Index Type	Read Benefit	Write Overhead	E-commerce Fit
Single-Column	Direct equality/range lookups	Medium	Essential for individual filters
Composite	Multi-predicate queries	High	Critical for faceted search
Clustered (PK)	Fast PK-order range scans	Low	Good for "newest" product queries
Non-Clustered	Speeds non-PK lookups/sorting	Medium	Index on price, rating
Full-Text	Keyword search + ranking	High	Ideal for descriptions
Spatial	Efficient geometric queries	Medium	Location-based product searches

1.5.7 Visual Helps

The internal node holds split keys "Electronics" and "Toys and Games," directing lookups: Categories alphabetically before "Electronics" go to the first leaf range (L1).

"Electronics" through just before "Toys and Games" map to the second leaf (L2).

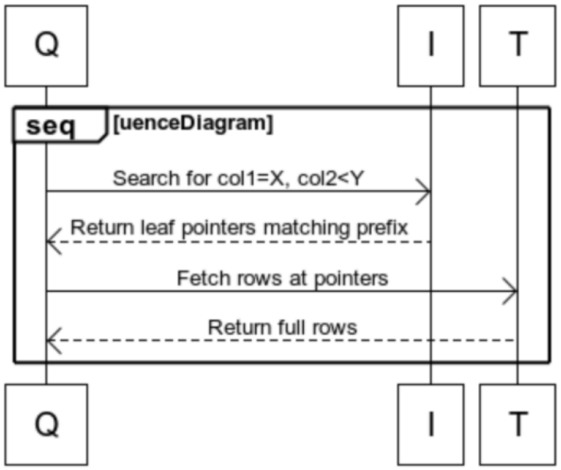


Figure 1.8: Lookup flow using a composite index—single index range scan then pointer fetch

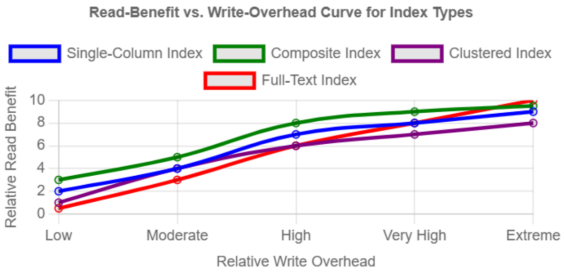


Figure 1.9: Relative write-overhead vs. read-benefit curve for each index type, annotated with e-commerce and banking points.

"Toys and Games" and beyond map to the third leaf (L3), where each leaf entry contains the category name and a pointer to the corresponding table rows.

(the read benefit values are taken just for illustration, their relativity however remains accurate)

Full-text and composite Indexes offer the highest read benefit, but only at steep write costs, making them ideal for read-heavy e-commerce workloads. clustered indexes and single-column indexes show more balanced profiles, hence better suited for transactional systems like banking.

By grounding each index type in our two real-world scenarios, you can see how "one size does not fit all." E-commerce platforms optimize for fast product discovery and user experience. By creating multiple indexes (e.g., on product titles, categories, or tags), they enable instant searches, filters, and recommendations so rich indexing for reads, while banking streamlines indexes for transaction speed and accuracy.

Aspect	E-Commerce	Banking
Primary Goal	Fast reads (user experience)	Fast writes (transaction integrity)
Index Strategy	Many indexes for flexible queries	Minimal indexes to reduce write lag
Data Dynamics	Stable product data	Highly volatile transactional data
Trade-Off	Acceptable write overhead	Prioritize write speed and ACID

Table 1.2: Balancing read vs. write in E-commerce and Banking workloads.

This balance ensures each system meets its core business needs: shoppers find products quickly, while banks process transactions securely.

1.6 Query Optimization Techniques

To ensure efficient query execution in MySQL, the optimizer applies a variety of techniques such as cost models, index selection, and join algorithms that choose the lowest-cost execution plan. Below, we examine each technique, illustrate how it operates in MySQL, and contrast its application in our two industries.

1.6.1 Cost-Based Optimization

Theory:

MySQL's optimizer evaluates multiple execution plans by estimating the "cost" of each in terms of CPU and memory. It uses table and index statistics (row counts, data distribution) to assign numeric costs and selects the plan with the lowest total cost.

MySQL Implementation:

Statistics Gathering: The `ANALYZE TABLE` command collects data about how unique and spread out your table's values are, helping the database speed up searches. Without these stats, the database might choose slow search methods, like scanning the entire table.

Cost Components:

- I/O cost: Time and resources spent reading/writing data from storage, such as a hard drive. (Fetching a page of data is like grabbing a book from a library shelf.)
- CPU cost: Processing power needed to compare, sort, or run calculations on data. (Checking if a username matches or calculating a total price.)
- Network cost: Delay and bandwidth used to sync data across servers in different locations.

(Copying updates from New York to Tokyo servers takes longer than local copies.)

(the analogies above have been taken from ChatGPT for better understanding)

Industry Examples:

E-commerce:

```
EXPLAIN
SELECT p.product_id, p.name, c.category_name
  FROM products p
  JOIN categories c ON p.category_id = c.category_id
 WHERE p.price BETWEEN 100 AND 500
   AND p.in_stock = 1;
```

The optimizer chooses between using a composite index on (`price`, `in_stock`) followed by an index lookup on `category_id`, versus scanning the clustered primary key index on `products` and filtering. Proper statistics ensure the composite-index plan is chosen for large catalogs.

Banking:

```
EXPLAIN
SELECT t.transaction_id, t.amount
  FROM transactions t
 WHERE t.account_id = 12345
   AND t.transaction_date >= '2025-01-01';
```

With a composite index on (account_id, transaction_date), the optimizer can better perform a low-cost range scan. However, if statistics are outdated, it may incorrectly opt for a full table scan, leading to high latency on large transaction tables. This reminds us why it's important to run `ANALYZE TABLE` to keep statistics up to date and ensure optimal query performance.

1.6.2 Index Selection Strategies

Normally, the database picks which index to use on its own. But you can also tell it exactly which one to try by adding commands like `USE INDEX` or `FORCE INDEX`. If you want to let a helper tool decide, something like Percona's `pt-index-usage` can scan your slow-query log and suggest useful indexes.

Online Shopping:

```
SELECT /*+ INDEX(p idx_price_instock) */
  p.product_id, p.name
FROM products p
WHERE p.price < 200
  AND p.in_stock = 1;
```

Bank Reports:

Banks often run big reports at night that join customer records with transaction details. A tool might notice these reports are slow and suggest adding an index on customer_id plus transaction_type. Since these reports run after hours, the tiny extra work every time you add or change data is well worth the much faster report time.

1.6.3 Join Algorithms

MySQL supports several physical join methods. The best method is picked based on estimated row counts and available indexes:

Algorithm	Description
Nested Loop Join	For each row in the outer table, scan or index-lookup matching rows in the inner table.
Merge Join	Requires both inputs sorted; merges them in one pass.
Hash Join	Builds hash table on smaller input, probes with larger input rows (MySQL 8.0 via hints).

Industry Examples:

E-commerce: Nested Loop + Index Join: When you want to match every category (10 000 rows) to its products (100 000 000 rows), the database starts with a category and then does a fast lookup on the products using the category index. Merge Join: You could also sort both tables and merge them in order, but because product lists change, the extra sorting will slow down things more than it helps.

Banking: Hash Join: For the end-of-day process of comparing today's transactions with archived ones, the database can load one table into memory and match records by hashing—this can be quicker if there's enough RAM. In MySQL 8.0, you can force this method with the hint `USE_HASH()`. Nested Loop: For everyday account checks, the system simply uses the account ID index to pull up each account's details instantly.

1.6.4 Additional Optimization Techniques

Query Rewriting:

If you don't wrap indexed columns in functions, the database can use the index directly. Bad:

```
WHERE YEAR(date) = 2025
```

Good:

```
WHERE date >= '2025-01-01' AND date < '2026-01-01'
```

Industry Examples:

E-commerce: Rewrite filters on `product_date` to use range scans instead of functions, enabling index use on the date column. **Banking:** Replace `WHERE MONTH(transaction_date)=1` with date-range predicates to avoid full scans on large transaction logs.

Partitioning:

Break a huge table into smaller pieces so each query only looks at the relevant slice. *Industry Examples:*

Banking: Split the `transactions` table by month. Old months can be ignored instantly, making reports and clean-up much faster.

E-commerce: Divide product data by category or region so searches in one area don't scan unrelated items.

Parallelism:

Let the database do several parts of a big scan at the same time.

Industry Examples:

E-commerce: In MySQL 8.0, turn on multiple read threads for daily analytics on sales data to generate reports in minutes.

Banking: Parallelize end-of-day reconciliation queries across multiple CPU cores to expedite batch processing.

1.7 Transaction and Conflict Management

Database indexing and query optimization interact closely with transaction control and concurrency mechanisms. In high-throughput systems, understanding locking behavior, isolation

levels, and mitigation strategies is important for striking a balance.

1.7.1 Locking Mechanisms and Concurrency Control

Theory:

MySQL's InnoDB engine uses row-level locks and two-phase locking (2PL) to enforce ACID properties. Locks can be shared (S-lock) for reads or exclusive (X-lock) for writes. When a transaction modifies a row, it acquires an X-lock on that row; other transactions requesting an S-lock must wait, and vice versa.

Industry Examples:

E-commerce (Read-Heavy):

Typical workload: Many concurrent `SELECT` queries on the `products` and `orders` tables. Impact: Shared locks for reads rarely block each other, so high concurrency is supported. However, long-running reporting queries can hold S-locks that block `UPDATE` or `DELETE`. Mitigation: Use consistent reads (MVCC Snapshots) for reporting—InnoDB's MVCC allows readers to see a snapshot of data without acquiring S-locks, avoiding contention.(Oracle Corporation 2024a)

Banking (Write-Heavy):

Typical workload: Numerous concurrent `INSERT`/`UPDATE` operations on `transactions`. Impact: X-locks on the same rows can lead to lock waits or deadlocks when two transactions modify related rows. Mitigation:

- Keep transactions short (limit scope of row modifications).
- Use consistent key ordering in multi-row updates to prevent circular waits (always update account A then account B).

1.7.2 Impact of Indexing on Transaction Throughput

Indexes accelerate searches but also influence locking behavior:

E-commerce:

Read transactions benefit: An indexed lookup on Primary key acquires minimal row locks (or none, under MVCC), so thousands of customers can browse simultaneously. Write transactions (e.g., updating inventory) must update all relevant indexes. Each index update requires acquiring X-locks on index tree pages, slowing writes if too many indexes exist.

Banking:

With **minimal indexes** (primary key + essential FKs), each `INSERT` on `transactions` only modifies two B-tree structures: the clustered primary index and one foreign-key index. Reducing indexes minimizes lock contention on index pages, increasing throughput.(Acceldata 2022)

1.7.3 Isolation Levels and Their Effects

Isolation levels specify how much Uncommitted work from other transactions can be seen and which anomalies (dirty reads, non-repeatable reads, phantom reads) are allowed. Let's first define the four isolation levels:

Read Uncommitted The weakest isolation level, permitting transactions to read data modified by other transactions but not yet committed. This minimizes locking overhead and maximizes concurrency, but risks dirty reads—reading uncommitted changes that may later be rolled back. *Example:* Transaction A updates a row but hasn't committed; Transaction B using Read Uncommitted sees the uncommitted change. If A then rolls back, B's read was invalid. *Use Case:* Rarely used in practice—only for workloads that can tolerate inconsistency, such as fast, approximate analytics.

Read Committed Allows a transaction to read only data committed by other transactions, eliminating dirty reads. However, it still permits non-repeatable reads: re-reading the same row within your transaction can yield different results if another transaction commits an update in between. *Example:* Transaction B reads a product's quantity as 100. Transaction A then changes it to 50 and commits. If B reads again, it sees 50. *Use Case:* Default in many systems (e.g., PostgreSQL). Balances consistency and performance—suitable for e-commerce and general OLTP workloads.

Repeatable Read Guarantees that any row read during the transaction cannot be changed by other transactions until this one completes, thus preventing dirty and non-repeatable reads. However, phantom reads are still possible: newly inserted rows matching your query may appear on subsequent reads. *Example:* A query for "products under "100" might initially return 10 rows; if another transaction inserts a qualifying product before you re-query, you'll see 11. *Use Case:* Ideal for financial reporting, where consistency of existing data matters, but the appearance of new records is acceptable.

Serializable The strictest isolation level, executing transactions as if they ran sequentially.

Prevents dirty reads, non-repeatable reads, and phantom reads by using rigorous locking or optimistic concurrency control (e.g., true snapshot isolation). *Example:* While Transaction A modifies a set of rows, Transaction B cannot read, update, or insert any rows in that set until A commits. *Use Case:* Reserved for mission-critical operations—bank transfers, inventory control—where absolute data integrity outweighs the cost of reduced concurrency and higher deadlock risk.

Table 1.3: Isolation levels, anomalies, and typical use cases

Isolation Level	**Dirty Read**	**Non-Repeatable Read**	**Phantom Read**	**Typical Use Case**
READ UNCOMMITTED	✓	✓	✓	Rarely used
READ COMMITTED	✗	✓	✓	Banking: moderate
REPEATABLE READ	✗	✗	✓	MySQL default; e-commerce reporting
SERIALIZABLE	✗	✗	✗	Strict consistency

Industry Examples:

E-commerce: Default (`REPEATABLE READ`):Ensures that reporting queries see a consistent snapshot without blocking writes. Phantom reads are permitted but sometimes problematic for product browsing.

Lower level (`READ COMMITTED`): used to reduceSnapshot Overhead (e.g., storage/CPU costs from retaining old data versions) This means that the database retains fewer old versions of data, lowering overhead. However,there's risk of non-repeatable reads (data changing during a transaction).

Banking: `READ COMMITTED`:: Uses shorter-lived snapshots (MVCC), freeing resources faster. It strikes a balance between consistency (no dirty data) and Throughput.

`SERIALIZABLE`: strict locking for critical operations where absolute consistency is required.

1.8 Mitigation Strategies for Conflicts and Bottlenecks

Locking Best Practices:

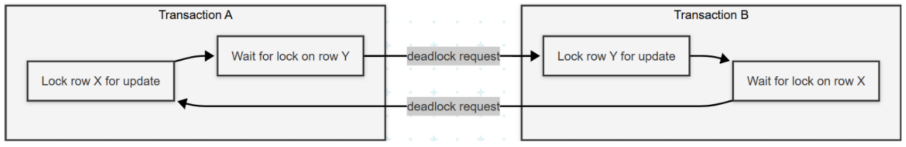


Figure 1.10: Illustrate a deadlock cycle between two transactions updating two accounts, with lock arrows and resolution after one is rolled back. (generated by memeriad io)

- Short Transactions: Break large updates into smaller chunks to reduce lock duration.
- Ordered Access: Standardize the order of row access in multi-row updates to prevent circular waits and Deadlocks.

Index Review and Tuning:

- Regularly audit index usage with tools like Percona's `pt-index-usage` and drop unused Indexes to minimize lock contention on Index pages (Percona 2022).

Deadlock Detection and Handling:

- Configure applications to catch Deadlock errors (ER_LOCK_DEADLOCK) and retry transactions automatically, as recommended by MySQL documentation (Oracle Corporation 2024a).

Partitioning and Sharding:

- Banking: Range-Partitioning the `transactions` table by date to localize locks within partitions, reducing cross-partition contention.
- E-commerce: Shard orders by customer region or hash on `customer_id` to distribute write load across multiple database instances.

1.9 High-Concurrency Indexing

High-concurrency systems must balance query speed with avoiding locks and deadlocks. We will continue comparing our two industries but this time imagining it is the end of the day. Based on this, we recommend optimal transaction isolation levels and Index designs for each.

1.9.1 Consistency Challenges under Heavy Load

Lock Contention on Index Pages

With many simultaneous writes (e.g., inventory updates or transaction inserts), multiple transactions may attempt to modify the same b-tree index nodes, causing waits or latch contention.

InnoDB's page-level latches protect b-tree index structures; heavy hot-spotting on upper nodes (root or first-level pages) can throttle performance (Oracle Corporation 2024a).

Deadlocks in Multi-Row Operations

Complex transactions that update multiple rows or tables (e.g., adjusting stock and order status together) may acquire locks in different orders, leading to circular waits.

InnoDB detects deadlocks and rolls back one transaction, but repeated deadlocks can harm Throughput (Percona 2022).

MVCC Snapshot Overhead

MySQL's MVCC Snapshots maintains older row versions to serve consistent reads without locking. Under extreme concurrency, long-running snapshots can bloat the Undo Log, requiring more disk I/O and cleanup (Oracle Corporation 2024a).

1.9.2 Case Comparison

Scenario	Workload Characteristics	Primary Concurrency Issue	Typical Mitigation
E-commerce Peak Sale	Thousands of simultaneous `SELECT` (product views) and bursts of `UPDATE` (inventory decrements)	Reader–writer contention on product and inventory Indexes	Use MVCC Snapshots consistent reads; isolate write-only tables; offload reads to Replicas; increase InnoDB buffer pool
Banking End-of-Day Processing	Bulk `INSERT` of settlement records, reconciliations, batch analytics queries	hot-spotting on transaction table root pages; long-running analytics queries blocking writes	Partitioning transactions by date; run analytics on read-only Replica; tune `innodb_log_file_size` and `log_buffer_size`

Table 1.4: High-concurrency scenarios and their main challenges.

1.9.3 Recommended Isolation Level and Index Strategies

E-commerce Peak Sale Isolation Level: `REPEATABLE READ` (MySQL default) to provide snapshot consistency for concurrent readers without locking writes.

Index Strategy:

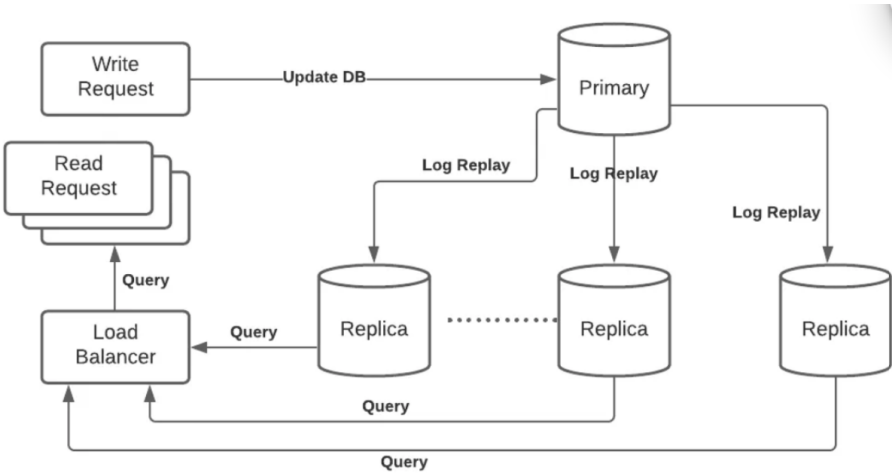


Figure 1.11: Primary–Replica Traffic Split: This diagram illustrates a system where the primary database handles all write operations, while multiple replicas handle read operations. This architecture enhances read scalability and provides redundancy, though it introduces eventual consistency due to replication lag

- Read-Optimized Replica: Direct all `SELECT` traffic to read replicas, each with identical Indexes to the primary.
- Minimal Write Indexes on Hot Tables: On core inventory tables, retain only essential Indexes (e.g., primary key on `product_id`, Index on `inventory_id`) to lessen write-side contention.

Banking End-of-Day Processing Isolation Level: `READ COMMITTED` to reduce undo-log growth and avoid long snapshots; prevents dirty reads while permitting short repeatable reads.

Index Strategy:

- Partitioned Primary Key: Range-Partitioning transactions by month, which distributes inserts across partitions and reduces b-tree index hot-spotting on a single root node.
- Deferred Index Builds: If possible, defer non-critical Index maintenance (e.g., rebuild secondary Indexes offline) to avoid interfering with bulk inserts.
-

1.10 Normalization and Index Design

Normalization structures data into related tables to eliminate redundancy, but it also influences how indexes should be designed. Below, we explore how each normal form affects index strategy

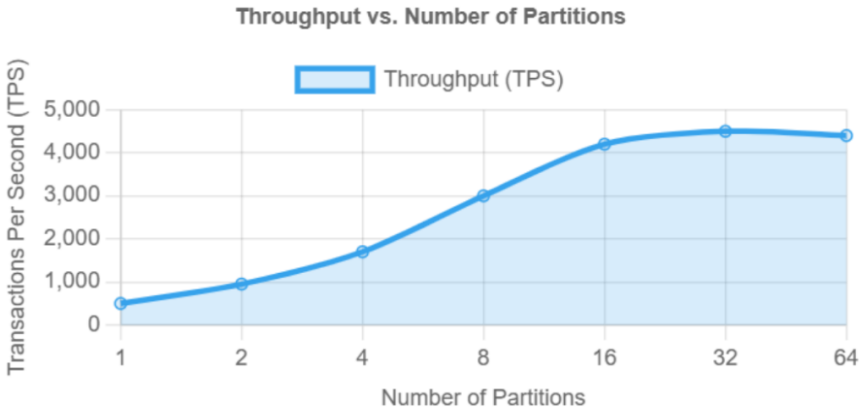


Figure 1.12: Throughput Scaling with Table Partitioning—demonstrates how increasing the number of Partitionings on a `transactions` table can enhance Throughput (TPS), especially in high-concurrency workloads. However, after a certain point, the gains taper off due to coordination overhead.

and compare optimal designs in e-commerce and banking schemas.

1.10.1 Effects of Normalization on Index Design

First Normal Form (1NF): Definition: Eliminate repeating groups; each column holds atomic values.

Index Implications: One-to-many relationships (e.g., a product with multiple tags) become separate tables (e.g., `product_tags`). Indexes on foreign-key columns (e.g., `product_id` in `product_tags`) are essential to join back to the main table.

Second Normal Form (2NF): Definition: Eliminate Partial Dependencys; non-key columns no longer depend on part of the composite primary key.

Index Implications: Composite primary keys (e.g., (`order_id`, `product_id`) in an order-detail table) require clustered composite indexes.

Third Normal Form (3NF) and Beyond: Definition: Eliminate Transitive Dependencys; non-key columns no longer depend on another non-key column, which in turn depends on the primary key.

Index Implications: More lookup tables (e.g., `category`, `brand`) mean more joins. Foreign-key indexes on each relationship column are essential to prevent table scans on joins across multiple normalized tables.

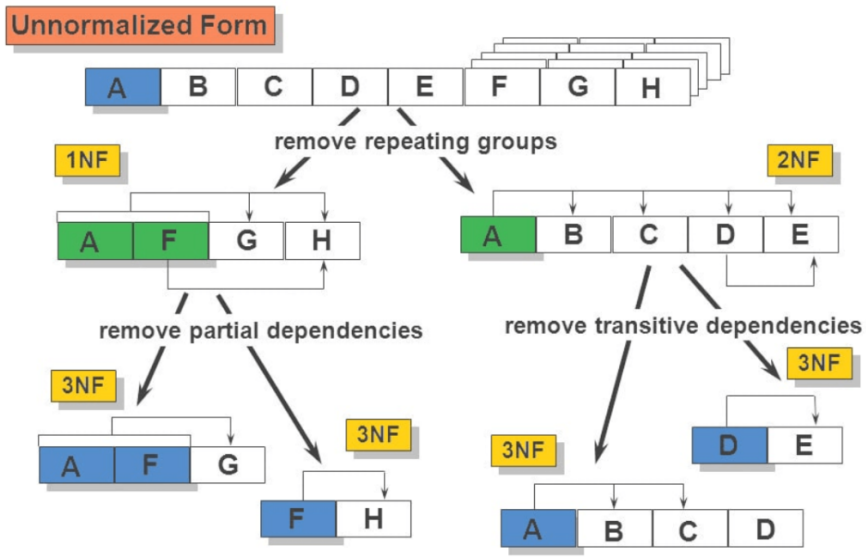


Figure 1.13: ALL types of Normalization

1.10.2 Optimal Index Designs for Normalized Schemas

Normal Form	Table Example	Primary Key	Recommended Indexes
1NF	`product_tags`	(`product_id`, `tag_id`)	Composite PK index; secondary index on `tag_id` for reverse lookup
2NF	`order_details`	(`order_id`, `product_id`)	Clustered composite index on PK; index on `product_id` and `quantity` filter
3NF	`transactions`, `accounts`, `customers`	`transaction_id`	PK on `transaction_id`; FK indexes on `account_id`, `customer_id`

Table 1.5: Index recommendations by normal form in normalized schemas.

1.10.3 Denormalization Trade-offs and Considerations

Denormalization can improve read performance by reducing joins at the cost of redundancy and more complex writes.

E-commerce Denormalization: Example: Storing `category_name` and `brand_name` directly in the `products` table (in addition to `category_id`/`brand_id`).

Benefit: Product listing queries avoid joins, reducing latency.

Drawback: Updates to category or brand names must propagate to all products, increasing write complexity and risk of inconsistency.

Banking Denormalization: Example: Embedding `account_balance` within the `customers` table, updated on each transaction.

Benefit: Fast read of account summary without joining `accounts` and `transactions`.

Drawback: Risk of Balance Drift if updates fail. Balance Drift occurs when a database transaction (e.g., transferring money between accounts) fails mid-process, leaving data in an inconsistent state. For example: if a bank transfer debits one account but crashes before crediting the other, total balances no longer match. This makes rollback harder because the system must track and reverse all partial updates.

1.11 Detailed Case Study: E-commerce vs. Banking

In this section, we'll have a final walk through of our two realistic scenarios: one in an E-commerce environment (read-heavy, catalog/search focus) and one in a Banking system (write-heavy, transactional focus) to illustrate how the indexing and optimization concepts discussed earlier play out in practice.

1.11.1 E-commerce Case Study

Scenario: A large online retailer maintains a product catalog of 5 million items. During a flash sale, read traffic spikes to 10 000 queries per second, primarily searches by category, keyword, price range, and geolocation.

1. Schema & Workload Characteristics Tables:

- `products(product_id PK, name, description, category_id, price, location POINT)`
- `categories(category_id PK, name)`
- `inventory(product_id FK, stock_level)`

Queries:

- **Full-text search** on description full-text index)
- **Range queries** on price btree index
- **Spatial queries** on location (e.g., "find nearest store") (spatial index)

Index Type	Definition	Purpose
B-tree on `category_id`	Non-clustered single-column btree index	Fast equality filtering for category lookups
Composite on (`price`, `category_id`)	Non-clustered composite Index	Efficient "price range within category" scans
Full-text on `description`	InnoDB full-text index	Keyword search, ranking by relevance
Spatial on `location`	spatial index (R-tree)	Proximity searches for geofencing or store locator

Table 1.6: E-commerce indexing strategy.

2. Indexing Strategy Rationale:

- B-trees excel at ordered and range scans (price, category).
- Composite indexes reduce I/O by covering multi-predicate `WHERE` clauses.
- Full-text is specialized for natural language search, automatically building inverted indexes.
- Spatial indexes accelerate geometric queries by partitioning space (grid/quadtree).

3. Query Execution & Optimization

```
EXPLAIN
SELECT p.name, p.price
  FROM products p
 WHERE MATCH(p.description) AGAINST('wireless earbuds')
   AND p.price BETWEEN 50 AND 200
   AND p.category_id = 3
 ORDER BY p.rating DESC
 LIMIT 20;
```

Uses full-text index for MATCH... AGAINST. Applies composite index (price, category_id) for residual filtering. Filesort on rating (unless a covering index is added).

```
EXPLAIN
SELECT p.name, c.name AS category, i.stock_level
  FROM products p
  JOIN categories c USING (category_id)
```

```
  JOIN inventory i USING (product_id)
 WHERE p.category_id = 3
   AND i.stock_level > 0;
```

MySQL chooses a nested-loop join, driving from the smaller categories table, using index lookups into products and inventory. If inventory were large, a hash join (MySQL 8.0+) might be more efficient, building an in-memory hash on inventory keyed by product_id.

4. Normalization & Denormalization Normalized: Ensures no duplicated data across products and categories.

Denormalization: Could add category_name directly into products to avoid a join under extreme read load—but at the cost of write complexity (must update both tables on category rename).

Key Takeaway: E-commerce workloads prioritize read throughput. Aggressive indexing (including specialized indexes) and selective denormalization often pay dividends.

1.11.2 Banking Case Study

Scenario: A retail bank processes 50 000 transactions per minute during business hours, with an end-of-day batch that reconciles all accounts. Consistency, atomicity, and throughput under concurrency are paramount.

1. Schema & Workload Characteristics Tables:

- `accounts(account_id PK, balance DECIMAL, status)`
- `transactions(txn_id PK, account_id FK, amount, txn_time, type)`
- `audit_log(log_id PK, txn_id FK, change_time, details)`

Workload:

- High-volume `INSERT`s into transactions
- Frequent `UPDATE`s on accounts.balance
- `SELECT`s for statement generation

Index Type	Definition	Purpose
Clustered on `transactions(txn_id)`	Primary key dictates physical order (clustered index)	Fast `INSERT` and sequential scan for daily reconciliation
Non-clustered on `account_id`	Secondary index on `transactions(account_id)` (non-clustered index)	Quickly retrieve all transactions for a given account
Hash index (MySQL 8+ MEMORY)	In-memory temporary hash index	Speed up aggregation during end-of-day reports in MEMORY engine
Composite on (`account_id`, `txn_time`)	Non-clustered composite composite index	Efficient date-range queries for statement cycles

Table 1.7: Banking indexing strategy.

2. Indexing Strategy Rationale:

- Clustered index on the primary key ensures `INSERT`s are append-only and efficient.
- Secondary non-clustered index supports frequent lookups by `account_id`.
- Composite index covers queries like "all transactions for account X between dates Y and Z" (Percona 2022).

3. Concurrency Control & Transactions Locking Mechanisms:

- InnoDB uses row-level locks by default, minimizing contention.
- High write concurrency can still lead to lock waits—mitigated via short transactions and proper index usage to avoid locking large ranges (Oracle Corporation 2024a).

Isolation Levels:

Level	Phenomena Prevented	Use Case
`READ COMMITTED`	Dirty reads	Standard transactional consistency, high throughput
`REPEATABLE READ` (default)	Non-repeatable/phantom reads	Safeguard balance consistency during processing
`SERIALIZABLE`	All anomalies	Critical batch jobs requiring strict correctness

Table 1.8: Isolation levels in banking workflows.

Deadlock Avoidance:

- Acquire locks in a consistent order (e.g., always update `accounts` before `transactions`).
- Keep transactions short and narrow (touch as few rows as possible).

4. Impact of Indexing on Throughput Proper indexing avoids full-table scans on `transactions`, which under heavy `INSERT` load would block insert threads and escalate locks. Composite indexes ensure reads for reconciliation don't scan the entire table, reducing contention with ongoing writes (Percona 2022).

Key Takeaway: Banking systems demand a balance between write performance and strict consistency. Index design must minimize lock contention while supporting fast reads for reconciliation.

1.11.3 Comparative Summary

Aspect	E-commerce	Banking
Primary Workload	Read-heavy (search, catalog)	Write-heavy (transactions)
Critical Index Types	full-text index, spatial index, btree index ranges	clustered index (PK), non-clustered index (FK), composite index
Join Patterns	Multi-table reads (`products` → `categories`)	Joins mainly for reporting/audit logs
Concurrency Strategy	Snapshot isolation; read replicas	row-level locks; careful lock ordering
Normalization Trade-offs	Denormalize for speed (cache category)	Strict normalization to ensure ACID
Optimization Tools	Query planner hints, `EXPLAIN`, optimizer statistics	Isolation tuning, deadlock diagnostics

Table 1.9: Side-by-side comparison of e-commerce vs. banking workload characteristics.

Conclusion: By examining these two industries side by side, we see how index selection, query optimization, normalization, and concurrency control must be tailored to workload characteristics: E-commerce leans into specialized and covering indexes plus selective denormalization to accelerate reads.

Banking emphasizes atomic writes, lock efficiency, and range or hash indexes that support high-volume inserts and safe, concurrent access.

These case studies demonstrate the practical application of B-trees, hashing, composite indexes, optimizer statistics, join algorithms, and transaction isolation and are all within the context of modern, high-performance database systems.

1.12 Analysis and Insights

In this section, we construct the opposing optimization strategies of our e-commerce and banking case studies, draw out critical trade-offs, link back to our CLOs, and surface emerging trends.

1.12.1 Read vs. Write Trade-Offs

1. How More Indexes Affect Updates E-commerce:

- Uses many indexes (composite, full-text index, covering) to speed up searches.
- Downside: Every extra index slows down updates.

Banking:

- Keeps indexes to a minimum (just a primary key plus one composite Index).
- Benefit: Insert speed improves.
- Drawback: Even adding a single extra index can slow inserts.

Takeaway: If your system mostly serves reads, you can afford more indexes to make queries fast. But if it mostly handles writes, too many indexes will hurt your write performance.

2. How Schema Design Affects Queries Normalized (Banking)

- Data is split cleanly into separate tables (3NF).
- Ensures accuracy and avoids duplicates, but requires more joins when you run reports.
- Solution: Add carefully chosen composite indexes and split old data by month for fast lookups across tables (Elmasri and Navathe 2015).

Denormalized (E-commerce)

- Puts extra fields (like category name or average rating) directly into the main product table.
- Speeds up listing pages by cutting out joins.
- Trade-off: Updates become more complicated and risk data drift if not synchronized.

Takeaway: Normalized schemas protect data consistency but need extra work at query time. Denormalized schemas boost read speed but make updates trickier. Choose based on whether your workload is read-heavy or write-critical.

1.12.2 Connection to CLOs

- **CLO 1 (Key Concepts):** We've demonstrated how B-trees, hash indexes, execution plans, and normalization principles support practical index and query optimization choices (Silberschatz et al. 2019).
- **CLO 3 (Transactions & Conflicts):** Our analysis of lock contention, isolation-level impacts, and index maintenance overhead shows the interplay between optimization and concurrency control in high-volume environments (Percona 2022).

1.12.3 Emerging Trends

1. Auto-Indexing and Machine-Assisted Tuning MySQL's Autopilot Indexing leverages machine learning to analyze query and Data Manipulation Language (DML) workloads, automatically recommending or creating candidate indexes. Early users have seen end-to-end query latency reductions of 10–15 % in environments with rapidly shifting access patterns (Oracle Corporation 2024a).

2. Hybrid Storage Engines Modern platforms combine row-oriented and column-oriented storage within the same ecosystem to support both Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) workloads:

- MyRocks: A RocksDB-based, LSM-tree engine optimized for write-heavy, key-value access patterns (O'Neil et al. 1996).
- MariaDB ColumnStore: A massively parallel columnar engine that ingests change-data-capture streams, enabling near-real-time analytics without separate ETL Pipelines (Informatica n.d.).

3. Serverless & Multi-Region Architectures Cloud-native MySQL offerings—such as Amazon Aurora Serverless and Aurora Global Database—dynamically scale compute and storage across regions. Key capabilities include:

- Auto-scaling read replicas to handle demand spikes.
- Global clusters for cross-region disaster recovery and low-latency reads.
- Configurable consistency models, from eventual consistency for catalogs to strong repli-

cation for ACID-critical systems (Services n.d.).

E-commerce platforms exploit eventual consistency to serve product catalogs quickly, whereas banking systems rely on strict replication guarantees to maintain ACID properties (Services n.d.).

1.13 Conclusion

In this report, we have explored the full spectrum of MySQL indexing and query optimization starting from fundamental concepts and index types to advanced techniques, transactional concerns, and real-world case studies in two contrasting industries. Below, we synthesize our key findings, and reflect on their implications.

1.13.1 Key Takeaways

- **Indexing Is Workload-Dependent:** Read-heavy environments (e-commerce) benefit from a rich index portfolio including composite, covering, and full-text indexes to drive search and listing queries; write-heavy environments (banking) succeed with a minimalist index set—primary keys plus a single composite index—to sustain high TPS and minimize lock contention (Acceldata 2022; Percona 2022).
- **Query Optimization Leverages Cost Models and Execution Plans:** Up-to-date statistics (via `ANALYZE TABLE`) are essential for accurate cost estimation and plan selection; techniques such as index hints, partition pruning, and join-algorithm tuning further refine performance (Chaiken and al. 2008; Oracle Corporation 2024a).
- **Transaction Management and Concurrency Control Are Integral:** Isolation levels (`READ COMMITTED` vs. `REPEATABLE READ`) and lock-avoidance strategies MVCC Snapshots, short transactions, ordered key access) directly influence throughput and latency; in high-concurrency scenarios, combining Partitioning and deferred index operations mitigates bottlenecks (Oracle Corporation 2024a; Percona 2022).
- **Normalization vs. Denormalization:** Normalized schemas ensure data integrity but require index-backed joins; denormalization reduces join overhead at the cost of more complex writes. The choice hinges on whether read latency or transactional integrity is paramount (Elmasri and Navathe 2015).

- **Continuous Monitoring and Iteration:** Real-time monitoring (slow-query logs, `EXPLAIN`-based alerts), periodic index audits, and adaptive tooling (auto-indexing features) are non-negotiable for sustaining optimal performance as data volumes and query patterns evolve (LoadForge 2023; Oracle Corporation 2024a).

1.13.2 Closing Reflection

Effective MySQL indexing and query optimization demand a deep understanding of both database internals b-tree indexes, cost models, MVCC Snapshots and application-level requirements (read vs. write priorities, consistency mandates). By systematically aligning index design, optimization techniques, and transaction controls with workload characteristics—as demonstrated in our e-commerce and banking case studies—organizations can achieve significant performance gains without overprovisioning hardware. As data scales and workloads evolve, the practices outlined here will serve as a robust framework for maintaining responsive, reliable, and efficient database systems.

Glossary

ACID Guarantees Atomicity, Consistency, Isolation, Durability.

Aggregation Summary functions like SUM, COUNT, or AVG.

B-tree index A balanced tree data structure that supports sorted lookup and range queries.

Balance Drift An inconsistency that arises when denormalized account balances become out of sync due to partial transaction failures..

Buckets Storage slots in a hash table for keys with the same or similar hash value.

Clustered index Rows physically stored in primary key order in InnoDB.

Data integrity Ensuring correctness and consistency of data over time.

Data Manipulation Language (DML) A subset of SQL used to manage and manipulate data within relational databases, including INSERT, UPDATE, DELETE, and SELECT commands..

Deadlock A situation where two or more transactions are waiting for each other to release locks, preventing progress..

Entities Real-world objects modeled as tables in a database.

Equality query Query fetching an exact value match (e.g., WHERE id = 5).

ETL Pipeline Processes that extract data from various sources, transform it into a suitable format or structure, and load it into a target system such as a data warehouse..

External caching Storing frequently accessed data outside the database for faster reads.

Foreign key A key in one table that refers to the primary key in another table.

Hash function A function that maps data to fixed-size numbers for fast access.

Hash index Index using a hash function to map keys into buckets for fast lookups.

Index A data structure that speeds up row access using key–row pointer pairs.

InnoDB MySQL's default storage engine supporting transactions and row-level locking.

Isolation levels READ COMMITTED vs. REPEATABLE READ transaction settings.

Key values Unique keys paired with stored data.

Logarithmic time Time complexity growing as $\log(n)$.

LSM-tree A log-structured merge-tree data structure designed for high write-throughput environments, organizing data into multiple levels and merging in-memory writes to on-disk structures..

MVCC Snapshots Private database state views under Multi-Version Concurrency Control, isolating reads from writes.

Non-clustered index Secondary index that maps key to PK pointer, requiring two lookups.

Normalization Database design technique to reduce redundancy and improve integrity.

Online Analytical Processing (OLAP) A computing approach that enables multidimensional analysis of large volumes of data, facilitating complex queries and data modeling..

Online Transaction Processing (OLTP) Systems that manage real-time transaction processing, supporting high volumes of short, atomic operations such as insertions, updates, and deletions..

Partial Dependency A situation in a composite primary key where a non-key column depends on only part of the key rather than the whole..

Partitioning Splitting a table into smaller parts to limit scan scope.

Primary key Unique identifier for rows in a table, usually indexed.

Query Execution Plan The series of steps MySQL takes to execute a SQL query.

Query latency Delay between query submission and result.

Random keys Values inserted in unpredictable order, making index updates costly.

Range query Query fetching values within limits (e.g., WHERE age BETWEEN 10 AND 20).

Range scans Retrieving rows within a specified key range.

Replica A copy of the primary database that receives updates asynchronously to serve read-only workloads..

Row location Physical placement of a row in storage.

Shard A physical subset of data distributed across servers by sharding key.

Snapshot Overhead Extra cost of maintaining and accessing MVCC snapshots, including version tracking and cleanup.

Sorting Arranging data in a specific order (ascending or descending).

Throughput Rate at which a system completes work, typically measured in operations per second.

Transitive Dependency A dependency in which a non-key column depends on another non-key column, which in turn depends on the primary key..

Uncommitted work Changes made by an active transaction not yet committed or rolled back.

Undo Log A transactional log used by InnoDB to store previous versions of data for rollback and MVCC snapshot creation..

Write append performance Speed of adding new data at the end of a table or index, fastest with sequential data.

Write-maintenance overhead Cost of maintaining indexes on writes.

References

Acceldata (2022). *Database optimization in e-commerce: A case study*. Acceldata Engineering Blog.

Chaiken, Robin and al., et (2008). "Extensible query optimization in Starburst". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*.

Elmasri, Ramez and Navathe, Shamkant B. (2015). *Fundamentals of Database Systems*. 7th ed. Pearson.

Informatica (n.d.). *What is an ETL Pipeline?* Informatica. Accessed n.d.

LoadForge (2023). *Implementing full-text search in MySQL for product catalogs*. LoadForge Insights.

O'Neil, Patrick, Cheng, Edward, Gawlick, Dieter, and O'Neil, Elizabeth (1996). "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4, pp. 351–385. DOI: 10.1007/s002360050048.

Oracle Corporation (2024a). *MySQL 8.0 Reference Manual*. Oracle.

– (2024b). *MySQL 8.0 Reference Manual*. URL: https://dev.mysql.com/doc/refman/8.0/en/.

Percona (2022). *Index maintenance for write-heavy workloads in MySQL*. Percona Blog.

Services, Amazon Web (n.d.). *What is OLAP?* aws.amazon.com. Accessed n.d.

Silberschatz, Abraham, Korth, Henry F., and Sudarshan, S. (2019). *Database System Concepts*. 7th ed. McGraw-Hill Education.

Appendix Chapter

This will include diagrams that help us understand the concepts better.

Normalized Structure

StudentID	StudentName
1	Alice
2	Bob

ClassID	ClassName	TeacherName
C101	Math	Mr. Smith
C102	Science	Mrs. Johnson

StudentID	ClassID
1	C101
2	C102

ClassID	Subject
C101	Algebra
C101	Geometry
C102	Physics
C102	Chemistry

Figure 14: Normalized Structure

Denormalized Structure

StudentID	StudentName	ClassName	TeacherName	Subject
1	Alice	Math	Mr. Smith	Algebra
1	Alice	Math	Mr. Smith	Geometry
2	Bob	Science	Mrs. Johnson	Physics
2	Bob	Science	Mrs. Johnson	Chemistry

Figure 15: Denormalised Structure

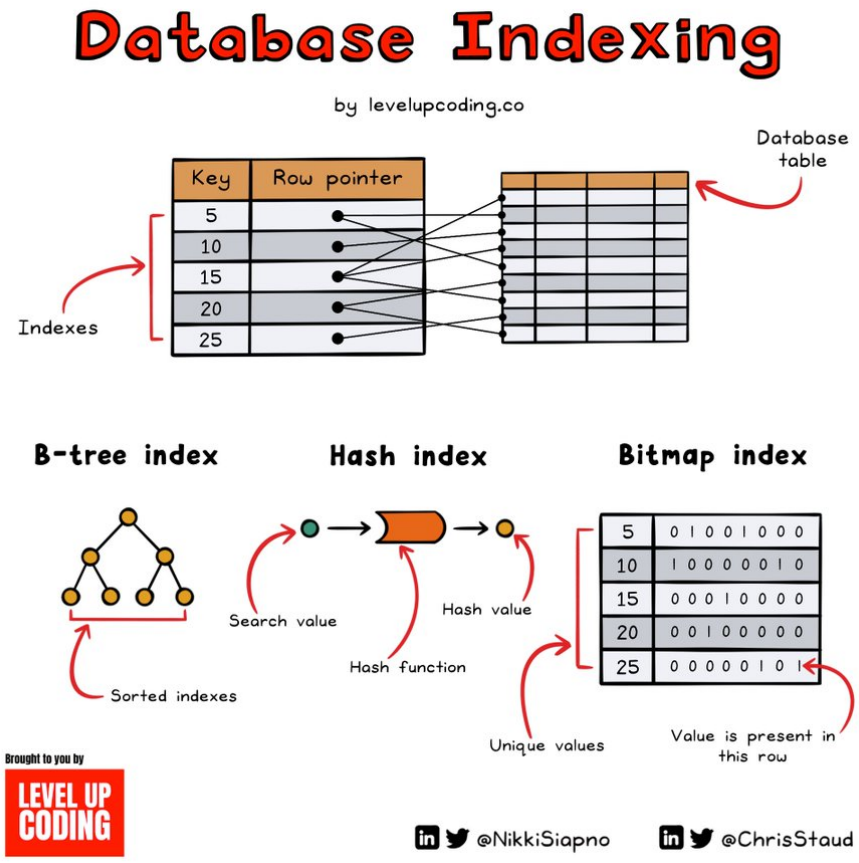
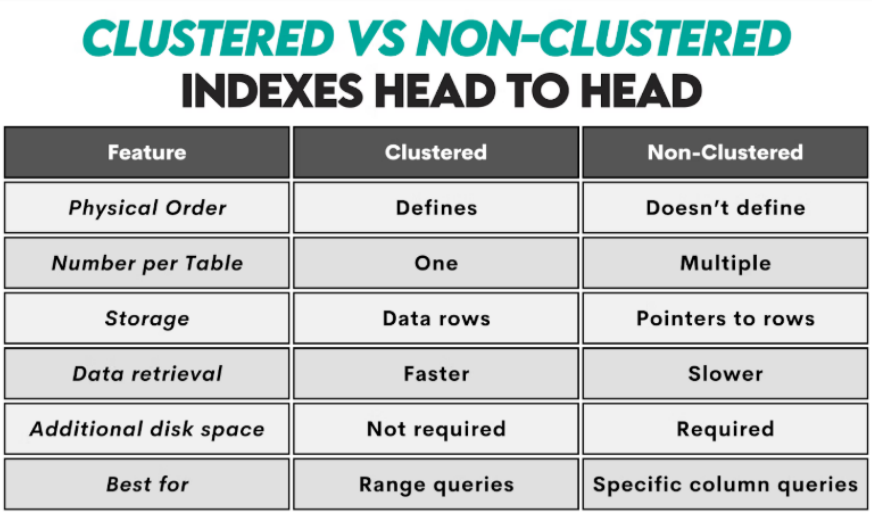


Figure 16: Database Indexing



Feature	Clustered	Non-Clustered
Physical Order	Defines	Doesn't define
Number per Table	One	Multiple
Storage	Data rows	Pointers to rows
Data retrieval	Faster	Slower
Additional disk space	Not required	Required
Best for	Range queries	Specific column queries

Figure 17: Summarised Comparison