MySQL Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What MySQL is, and how the server works 3
Installing and starting MySQL 8.4 LTS 4
Connecting with the `mysql` client 5
Users, host patterns, and least privilege 7
Loading the posts database and choosing column types 9
JSON, `FULLTEXT` search, and upserts 12
CTEs and window functions in MySQL 8.4 16
InnoDB transactions, locking, and implicit DDL commits 18
Indexes, `EXPLAIN`, and `EXPLAIN ANALYZE` 21
Using MySQL from Node.js (`mysql2`) 23
Observability, storage internals, and server tuning 27
Backups, restores, and migrations 30
Production readiness checklist 33
MySQL vs PostgreSQL vs MariaDB 33

Preface

This is the MySQL Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

This book is about MySQL 8.4 LTS itself: the server, `mysql` client commands, user accounts with host patterns (`'user'@'host'`), `utf8mb4` and collations, InnoDB storage and `REPEATABLE READ` transactions, the silent inline `REFERENCES` trap, implicit commits on DDL, JSON and `FULLTEXT` search, `EXPLAIN` and `EXPLAIN ANALYZE` indexes, Node.js with `mysql2`, `mysqldump` backups, and how MySQL compares to PostgreSQL and MariaDB.

It assumes you know SQL. If `SELECT`, `JOIN`, and `GROUP BY` are new to you, start with my SQL Handbook. This book focuses on what is specific to MySQL.

We use the same real dataset as the SQL, PostgreSQL, and SQLite handbooks: the 2,178 published posts of my blog (over 2 million words of Markdown and their JSON frontmatter) plus 200,000 `post_views` rows so `EXPLAIN ANALYZE`, covering indexes, `FULLTEXT` search, and `mysqldump` show real, measurable numbers.

Every command, timing, and output in this book comes from a real run on my Mac, in November 2026, on these versions:

Software	Version
MySQL	8.4.13 LTS (Homebrew on macOS)
Node.js	22.23.5
mysql2 (Node.js driver)	3.24.6

What MySQL is, and how the server works

MySQL is an open source relational database server. Like PostgreSQL, a dedicated server daemon (`mysqld`) owns the data directory, listens on a TCP port (3306 by default) and a local Unix socket, authenticates users, executes SQL, and coordinates concurrent clients.

One architectural detail sets MySQL apart from PostgreSQL: **MySQL uses threads instead of processes**, and it separates the SQL layer from **storage engines**.

- **InnoDB** is the default storage engine (since MySQL 5.5) and the only one you should use for application tables. It provides ACID transactions, row-level locking, crash recovery, and foreign keys.
- **Database and schema are synonyms in MySQL.** `CREATE DATABASE blog` and `CREATE SCHEMA blog` do the same thing, and a single connection can join tables across databases (`SELECT ... FROM blog.posts JOIN analytics.views ...`).

Let's inspect a running MySQL 8.4.13 server:

```
SELECT VERSION() AS version,
       @@default_storage_engine AS engine,
       @@transaction_isolation AS isolation,
       @@character_set_server AS charset,
       @@collation_server AS collation;
```

```
+---------+--------+-----------------+---------+--------------------+
| version | engine | isolation       | charset | collation          |
+---------+--------+-----------------+---------+--------------------+
| 8.4.13  | InnoDB | REPEATABLE-READ | utf8mb4 | utf8mb4_0900_ai_ci |
+---------+--------+-----------------+---------+--------------------+
```

Notice two defaults that differ from PostgreSQL right out of the gate:

1. Default isolation level is **`REPEATABLE-READ`** (PostgreSQL defaults to `READ COMMITTED`).
2. Default collation `utf8mb4_0900_ai_ci` is **case-insensitive** (`ci`) and accent-insensitive (`ai`), so `WHERE slug = 'NODE-MYSQL'` matches `'node-mysql'`.

Installing and starting MySQL 8.4 LTS

MySQL 8.4 is the current Long-Term Support (LTS) release line, which makes it the right default for production servers and local development.

macOS with Homebrew

Install the versioned 8.4 formula:

```
brew install mysql@8.4
```

Because `mysql@8.4` is keg-only, add its `bin` folder to your `PATH` in `~/.zshrc`:

```
export PATH="/opt/homebrew/opt/mysql@8.4/bin:$PATH"
```

Start it as a background service with `brew services start mysql@8.4`, or check its status with `brew services list`.

Running a temporary local instance

For this book, I kept my system clean by initializing a standalone data directory on port `3318`:

```
mysqld --initialize-insecure --datadir=/tmp/myb/data
mysqld --datadir=/tmp/myb/data --port=3318 --socket=/tmp/myb/mysql.sock \
  --mysqlx=0 --log-error=/tmp/myb/mysqld.log --daemonize
```

Check that it's alive with `mysqladmin ping`, and stop it cleanly later with `mysqladmin shutdown`:

```
mysqladmin --socket=/tmp/myb/mysql.sock -u root ping
mysqladmin --socket=/tmp/myb/mysql.sock -u root shutdown
```

Ubuntu and Docker

On Ubuntu 24.04, install the server package with `sudo apt install mysql-server`, check it with `sudo systemctl status mysql`, and harden the installation with `sudo mysql_secure_installation`.

For a quick container (using OrbStack or Docker):

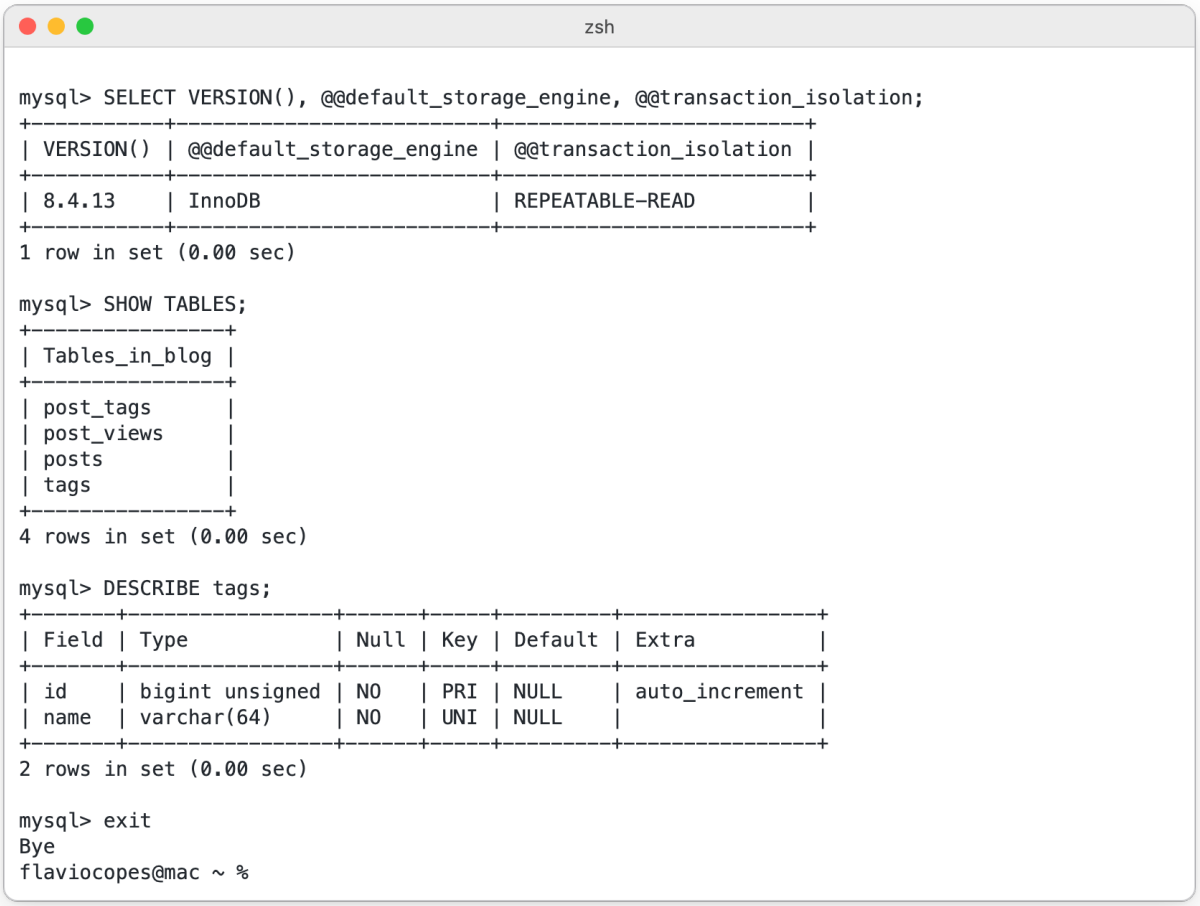
```
docker run --name mysql-dev -e MYSQL_ROOT_PASSWORD=secret -p 3306:3306 -d mysql:8.4
```

Connecting with the `mysql` client

Connect to a local server via Unix socket or TCP:

```
mysql --socket=/tmp/myb/mysql.sock -u root blog
mysql -h 127.0.0.1 -P 3318 -u blog_app -p blog
```

Never pass the password right after `-p` on the command line (`-psecret`), because other users on the machine can see it in `ps` and it stays in your shell history. Either type `-p` with nothing after it (MySQL prompts securely), use `MYSQL_PWD` in a script, or store credentials in `~/.my.cnf` with `chmod 600`.



Everyday commands inside `mysql`

Command	What it does
`SHOW DATABASES;`	list databases
`USE blog;`	switch to the `blog` database
`SHOW TABLES;`	list tables in the current database
`DESCRIBE tags;`	show columns, types, nullability, keys, and extras
`SHOW CREATE TABLE tags\G`	print the exact DDL, charset, collation, and constraints
`SHOW INDEX FROM post_views;`	list all indexes on a table
`SHOW PROCESSLIST;`	see active connections and running queries
`\G` (instead of `;`)	display rows vertically (great for wide tables)
`status` or `\s`	show connection info, server version, socket, and charset
`exit` or `\q`	quit the client

Reading connection errors

When a connection fails, the error number tells you where it stopped. I triggered the three most common ones on my test server:

```
ERROR 2003 (HY000): Can't connect to MySQL server on '127.0.0.1:3319' (61)
ERROR 1045 (28000): Access denied for user 'nobody'@'localhost' (using password: NO)
ERROR 1049 (42000): Unknown database 'nope'
```

- **ERROR 2003**: nothing is listening on that host/port (server stopped, wrong port, or firewall blocking).
- **ERROR 1045**: you reached MySQL, but the username, host pattern, or password didn't match.
- **ERROR 1049**: you logged in, but the database name doesn't exist.

One subtle trap with `mysql -h localhost` vs `mysql -h 127.0.0.1`: on Unix systems, **`localhost` tells the client to use the Unix socket file**, while **`127.0.0.1` forces TCP**. That matters for both connectivity and user host matching.

Users, host patterns, and least privilege

In MySQL, an account is not just a username. It is a **`'user'@'host'` pair**.

`'blog_app'@'localhost'` and `'blog_app'@'127.0.0.1'` and `'blog_app'@'%'` are three completely separate accounts that can have different passwords and different privileges.

Creating least-privilege accounts

Keep `root` strictly for administration and schema migrations. For the blog database, I created two dedicated application accounts:

```
CREATE DATABASE blog CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;

CREATE USER 'blog_app'@'127.0.0.1' IDENTIFIED BY 'app-secret-123';
CREATE USER 'blog_reader'@'127.0.0.1' IDENTIFIED BY 'reader-secret-123';

GRANT SELECT, INSERT, UPDATE, DELETE ON blog.* TO 'blog_app'@'127.0.0.1';
GRANT SELECT ON blog.* TO 'blog_reader'@'127.0.0.1';
```

Inspect the accounts and their authentication plugin in `mysql.user`:

```
SELECT user, host, plugin FROM mysql.user WHERE user LIKE 'blog%';
```

```
+-------------+-----------+-----------------------+
| user        | host      | plugin                |
+-------------+-----------+-----------------------+
| blog_app    | 127.0.0.1 | caching_sha2_password |
| blog_reader | 127.0.0.1 | caching_sha2_password |
+-------------+-----------+-----------------------+
```

MySQL 8.4 uses `caching_sha2_password` by default (the older `mysql_native_password` plugin is disabled by default in 8.4).

Check exact grants with `SHOW GRANTS`:

```
SHOW GRANTS FOR 'blog_app'@'127.0.0.1';
```

```
GRANT USAGE ON *.* TO `blog_app`@`127.0.0.1`
GRANT SELECT, INSERT, UPDATE, DELETE ON `blog`.* TO `blog_app`@`127.0.0.1`
```

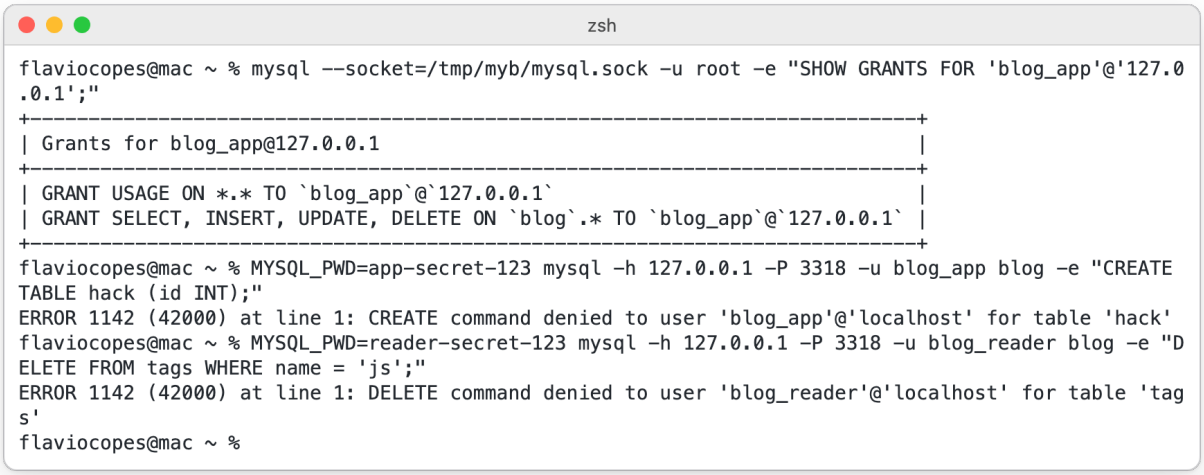
`GRANT USAGE ON *.*` simply means "this account is allowed to log in, with no global privileges".

Verifying permissions in action

Let's test both accounts. `blog_app` can read rows (`2,178 posts, 2,082,191 words`), but cannot alter the schema, and `blog_reader` cannot delete rows:

```
ERROR 1142 (42000) at line 1: CREATE command denied to user 'blog_app'@'localhost' for table 'hack'
ERROR 1142 (42000) at line 1: DELETE command denied to user 'blog_reader'@'localhost' for table 'tags'
```

To remove a privilege later, run `REVOKE DELETE ON blog.* FROM 'blog_app'@'127.0.0.1';`. Unlike PostgreSQL, `GRANT` on `blog.*` applies to future tables in `blog` automatically.



Host pattern	Matches	When to use
`'app'@'localhost'`	Local Unix socket connections only	App running on the same Linux host
`'app'@'127.0.0.1'`	Local IPv4 TCP connections	Local dev or SSH tunnel
`'app'@'10.0.1.%'`	Private VPC subnet	App servers in a private network

Host pattern	Matches	When to use
`'app'@'%'`	Any remote TCP host	Managed DBs with TLS + firewall allowlist

Loading the posts database and choosing column types

Let's load the 2,178 published posts of `flaviocopes.com` into `blog` using Node.js and `mysql2/promise`. Here is the schema:

```
CREATE TABLE posts (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  slug VARCHAR(200) NOT NULL UNIQUE,
  title VARCHAR(300) NOT NULL,
  published_on DATE NOT NULL,
  words INT UNSIGNED NOT NULL,
  body MEDIUMTEXT NOT NULL,
  frontmatter JSON NOT NULL,
  FULLTEXT KEY posts_ft_idx (title, body)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE tags (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(64) NOT NULL UNIQUE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE post_tags (
  post_id BIGINT UNSIGNED NOT NULL,
  tag_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (post_id, tag_id),
  CONSTRAINT fk_pt_post FOREIGN KEY (post_id) REFERENCES posts (id) ON DELETE CASCADE,
  CONSTRAINT fk_pt_tag FOREIGN KEY (tag_id) REFERENCES tags (id) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Here is the Node.js loader script (`load.mjs`) that reads every published Markdown file, parses its YAML frontmatter, creates the tables, and inserts all rows inside a single InnoDB transaction:

```
import mysql from 'mysql2/promise'
import { readdir, readFile } from 'node:fs/promises'
import path from 'node:path'
import { parse } from 'yaml'

const dir = './src/posts'
const today = new Date()
const posts = []

for (const file of (await readdir(dir)).sort()) {
  if (!file.endsWith('.md')) continue
  const source = await readFile(path.join(dir, file), 'utf8')
  const match = source.match(/^---\r?\n([\s\S]*?)\r?\n---\r?\n([\s\S]*)$/)
  if (!match) continue
  const data = parse(match[1]) ?? {}
  const body = match[2]
  const date = new Date(data.date)
  if (data.draft || Number.isNaN(date.getTime()) || date > today) continue
  const tags = [...new Set([data.tag ?? []].flat().map((t) => String(t).toLowerCase()))]
  const words = body.replace(/```[\s\S]*?```/g, '').split(/\s+/).filter(Boolean).length
  posts.push({
    slug: file.replace(/\.md$/, ''),
    title: String(data.title),
    published_on: date.toISOString().slice(0, 10),
```

```
    words,
    body,
    frontmatter: JSON.stringify(data),
    tags,
  })
}

const tagNames = [...new Set(posts.flatMap((p) => p.tags))].sort()
const conn = await mysql.createConnection({
  socketPath: '/tmp/myb/mysql.sock',
  user: 'root',
  database: 'blog',
})

const t0 = performance.now()
await conn.beginTransaction()
const tagIds = new Map()
for (const name of tagNames) {
  const [res] = await conn.execute('INSERT INTO tags (name) VALUES (?)', [name])
  tagIds.set(name, res.insertId)
}
for (const p of posts) {
  const [res] = await conn.execute(
    'INSERT INTO posts (slug, title, published_on, words, body, frontmatter) VALUES (?, ?, ?, ?, ?, ?)
    ↪ ',
    [p.slug, p.title, p.published_on, p.words, p.body, p.frontmatter],
  )
  for (const t of p.tags) {
    await conn.execute('INSERT INTO post_tags (post_id, tag_id) VALUES (?, ?)', [res.insertId, tagIds.
    ↪ get(t)])
  }
}
await conn.commit()
console.log(`Loaded ${posts.length} posts, ${tagNames.length} tags in ${(performance.now() - t0).
    ↪ toFixed(1)} ms`)
await conn.end()
```

Loading all 2,178 posts (over 2 million words), 49 tags, and their join links inside one transaction took **1,037 ms** on my Mac.

Why did wrapping the loop in `beginTransaction()` and `commit()` matter so much? By default, MySQL runs in `autocommit = 1` mode, which means every standalone `INSERT` flushes the InnoDB redo log to disk (`fsync`) before returning. Running 4,400 individual autocommit inserts would flush to disk 4,400 times. Wrapping them in one transaction flushes once at `COMMIT`.

For even larger bulk loads (like the 200,000 `post_views` rows we insert later), batch 1,000 to 2,000 rows per multi-row `INSERT INTO post_views (post_id, viewed_at, country) VALUES (...), (...), ...` statement. That cuts both network round-trips and SQL parsing overhead.

Strict SQL mode in MySQL 8.4

Older versions of MySQL were notorious for silently truncating strings that didn't fit in a `VARCHAR` or turning invalid dates like `'2026-02-30'` into `'0000-00-00'`.

MySQL 8.4 enables strict mode by default (`STRICT_TRANS_TABLES`, `NO_ZERO_DATE`, `ERROR_FOR_DIVISION_BY_ZERO`, and `ONLY_FULL_GROUP_BY`). If a value doesn't fit or a date is invalid, MySQL rejects the statement with an error instead of silently

corrupting your data.

You can check your server's active SQL mode at any time:

```
SELECT @@sql_mode;
```

Leave strict mode on. If you ever inherit a legacy app that fails under `ONLY_FULL_GROUP_BY`, fix the `GROUP BY` query rather than turning off strict mode globally.

Why `utf8mb4` matters (never `utf8` / `utf8mb3`)

In older MySQL versions, `utf8` was an alias for `utf8mb3`, which only supports 3-byte UTF-8 characters and breaks on emoji and many non-Latin symbols. Watch what happens when you insert `'Hello 🚀'` into a `utf8mb3` table vs a `utf8mb4` table:

```
CREATE TEMPORARY TABLE enc_bad (txt VARCHAR(20)) CHARACTER SET utf8mb3;
INSERT INTO enc_bad VALUES ('Hello 🚀');
```

```
ERROR 1366 (HY000) at line 2: Incorrect string value: '\xF0\x9F\x9A\x80' for column 'txt' at row 1
```

Now with `utf8mb4`:

```
CREATE TEMPORARY TABLE enc_good (txt VARCHAR(20)) CHARACTER SET utf8mb4;
INSERT INTO enc_good VALUES ('Hello 🚀');
SELECT txt, CHAR_LENGTH(txt) AS chars, LENGTH(txt) AS bytes FROM enc_good;
```

```
+----------+-------+-------+
| txt      | chars | bytes |
+----------+-------+-------+
| Hello 🚀 |     7 |    10 |
+----------+-------+-------+
```

7 characters, 10 bytes (the rocket emoji takes 4 bytes). Always declare `CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci` on your databases and tables.

The inline `REFERENCES` trap

This is one of MySQL's most infamous gotchas. In PostgreSQL and SQLite, you can write a foreign key right next to a column: `pid INT REFERENCES fk_trap_parent(id)`.

MySQL parses that syntax without an error, and **silently ignores it**:

```
CREATE TABLE fk_trap_parent (id INT PRIMARY KEY);
CREATE TABLE fk_trap_child (pid INT REFERENCES fk_trap_parent(id));
INSERT INTO fk_trap_child VALUES (999);
SELECT * FROM fk_trap_child;
SHOW CREATE TABLE fk_trap_child\G
```

```
+------+
| pid  |
+------+
|  999 |
+------+
       Table: fk_trap_child
Create Table: CREATE TABLE `fk_trap_child` (
  `pid` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

`999` went right in, and `SHOW CREATE TABLE` proves the constraint was never created. Always write foreign keys as table-level `CONSTRAINT name FOREIGN KEY (col) REFERENCES other (id)`, like we did in `post_tags`.

Choosing column types in MySQL

Need	Use	Avoid / watch out
Primary key ID	`BIGINT UNSIGNED AUTO_INCREMENT`	`INT` (caps at 2.1B signed)
Short strings, indexed	`VARCHAR(n)`	`TEXT` (needs prefix length to index)
Long text (posts, JSON)	`TEXT` (64 KB) or `MEDIUMTEXT` (16 MB)	`VARCHAR` over a few KB
Money	`DECIMAL(12, 2)` or `BIGINT` cents	`FLOAT` / `DOUBLE`
Calendar date	`DATE`	`VARCHAR`
Event timestamp	`DATETIME(6)` (stored in UTC) or `TIMESTAMP`	`TIMESTAMP` caps at year 2038
Boolean	`TINYINT(1)` / `BOOLEAN`	Stored as `0` or `1`
Structured metadata	`JSON`	Validated binary JSON, like `jsonb`

Notice `MEDIUMTEXT` for `posts.body`: MySQL's `TEXT` tops out at 65,535 bytes, and my longest blog posts (19,000+ words) exceed 64 KB! `MEDIUMTEXT` handles up to 16 MB cleanly.

JSON, `FULLTEXT` search, and upserts

MySQL 8.4 stores `JSON` columns in an optimized binary format, includes `FULLTEXT` indexes inside InnoDB, and supports atomic upserts.

Querying `JSON` columns

Just like PostgreSQL and SQLite, MySQL supports `->` (returns JSON) and `->>` (returns unquoted text):

```
SELECT JSON_TYPE(frontmatter -> '$.tag') AS tag_type, COUNT(*) AS posts
FROM posts GROUP BY 1 ORDER BY 2 DESC;
```

```
+----------+-------+
| tag_type | posts |
+----------+-------+
| STRING   |  2071 |
| ARRAY    |   107 |
+----------+-------+
```

Finding the most recently updated posts with `JSON_CONTAINS_PATH`:

```
SELECT slug, frontmatter ->> '$.updated' AS updated
FROM posts
WHERE JSON_CONTAINS_PATH(frontmatter, 'one', '$.updated')
ORDER BY updated DESC LIMIT 4;
```

```
+--------------------------+------------+
| slug                     | updated    |
+--------------------------+------------+
| lm-studio-bionic         | 2026-11-28 |
| grok-build               | 2026-11-28 |
| coding-agents-from-phone | 2026-11-28 |
| claude-duet              | 2026-11-28 |
+--------------------------+------------+
```

Indexing JSON fields with generated columns

Querying a JSON column with `frontmatter->>'$.tag'` is convenient, but MySQL cannot use a regular B-tree index on a JSON expression directly unless you expose it via a **generated column** (or a functional index).

Let's measure the difference on our 2,178 posts. First, inspect the keys inside `frontmatter`:

```
SELECT slug, JSON_KEYS(frontmatter) AS keys_json
FROM posts
LIMIT 3;
```

```
+---------------------------------------+-------------------------------------------------------------
| slug                                  | keys_json
    ↪       |
+---------------------------------------+-------------------------------------------------------------
| things-to-avoid-in-javascript-bad-parts | ["tag", "date", "title", "updated", "description", "
    ↪ tags_weight"] |
| introduction-to-yeoman                | ["tag", "date", "title", "updated", "description"]
    ↪       |
| html5-useful-tricks                   | ["tag", "date", "title", "noindex", "description"]
    ↪       |
+---------------------------------------+-------------------------------------------------------------
```

Now run `EXPLAIN ANALYZE` on a JSON filter before adding any index:

```
EXPLAIN ANALYZE
SELECT slug, title FROM posts
WHERE frontmatter->>'$.tag' = 'javascript'\G
```

```
EXPLAIN: -> Filter: (json_unquote(json_extract(posts.frontmatter,'$.tag')) = 'javascript')
           (cost=1205 rows=1603) (actual time=1.18..1.18 rows=0 loops=1)
    -> Table scan on posts  (cost=1205 rows=1603) (actual time=0.0127..0.92 rows=2178 loops=1)
```

MySQL had to scan all 2,178 rows and extract the JSON field from every single row, taking **1.18 ms**.

Now add a `VIRTUAL` generated column `primary_tag` and index it:

```
ALTER TABLE posts
  ADD COLUMN primary_tag VARCHAR(64)
    GENERATED ALWAYS AS (frontmatter->>'$.tag') VIRTUAL,
  ADD INDEX posts_primary_tag_idx (primary_tag);
```

Run `EXPLAIN ANALYZE` again:

```
EXPLAIN ANALYZE
SELECT slug, title FROM posts
WHERE primary_tag = 'javascript'\G
```

```
EXPLAIN: -> Index lookup on posts using posts_primary_tag_idx (primary_tag='javascript')
           (cost=0.825 rows=1) (actual time=0.00704..0.00704 rows=0 loops=1)
```

Down from **1.18 ms** to **0.00704 ms** (7 microseconds, **167× faster**). And we can group by `primary_tag` directly:

```
SELECT primary_tag, COUNT(*) AS n
FROM posts
WHERE primary_tag IS NOT NULL
GROUP BY primary_tag
ORDER BY n DESC
LIMIT 6;
```

```
+-------------+-----+
| primary_tag | n   |
+-------------+-----+
| js          | 342 |
| ai          | 204 |
| node        | 149 |
| platform    |  94 |
| react       |  88 |
| cli         |  86 |
+-------------+-----+
```

MySQL offers two kinds of generated columns:

Type	Stored on disk in table?	Can be indexed?	When to use
`VIRTUAL` (default)	No (computed on read, stored only in secondary index)	Yes	Almost always, especially for JSON extraction
`STORED`	Yes (written on `INSERT` / `UPDATE`)	Yes	Expensive expressions read frequently without an index, or foreign key targets

Start with `VIRTUAL`. It takes zero extra space in the main table and still lets InnoDB build a full secondary B-tree index on the extracted value.

Built-in `FULLTEXT` search

Because we added `FULLTEXT KEY posts_ft_idx (title, body)` on `posts`, InnoDB maintains an inverted full-text index automatically. Query it with `MATCH(...) AGAINST(...)`:

```
SELECT slug,
       ROUND(MATCH(title, body) AGAINST ('mysql innodb' IN NATURAL LANGUAGE MODE), 2) AS score
FROM posts
WHERE MATCH(title, body) AGAINST ('mysql innodb' IN NATURAL LANGUAGE MODE)
ORDER BY score DESC LIMIT 4;
```

```
+-------------------------+--------+
| slug                    | score  |
+-------------------------+--------+
| full-ubuntu-disk-codex | 103.39 |
| mysql-how-to-install   |  83.16 |
| node-mysql             |  82.78 |
| linux-server-commands  |   47.2 |
+-------------------------+--------+
```

For boolean operators (`+mysql -postgres "connection pool"`), switch `IN NATURAL LANGUAGE MODE` to `IN BOOLEAN MODE`.

Upserts with `ON DUPLICATE KEY UPDATE`

MySQL's atomic upsert uses `ON DUPLICATE KEY UPDATE`. In MySQL 8.0.19+ and 8.4, use a row alias (`AS new_row`) instead of the deprecated `VALUES()` function:

```
CREATE TABLE page_views (
  slug VARCHAR(200) PRIMARY KEY,
  views BIGINT UNSIGNED NOT NULL DEFAULT 0
);

INSERT INTO page_views (slug, views) VALUES ('node-mysql', 1) AS new_row
  ON DUPLICATE KEY UPDATE views = page_views.views + new_row.views;
```

```
INSERT INTO page_views (slug, views) VALUES ('node-mysql', 4) AS new_row
  ON DUPLICATE KEY UPDATE views = page_views.views + new_row.views;

SELECT * FROM page_views;
```

```
+------------+-------+
| slug       | views |
+------------+-------+
| node-mysql |     5 |
+------------+-------+
```

CTEs and window functions in MySQL 8.4

MySQL 8.0 finally added Common Table Expressions (`WITH`) and SQL window functions (`OVER (...)`), and in MySQL 8.4 they are fast and mature. They let you replace tangled subqueries and client-side JavaScript loops with a single readable SQL query.

Ranking and running totals with `WITH` and `OVER`

Suppose we want the top 8 tags on the blog, their rank, and a cumulative running total of post assignments. Using a CTE to count posts per tag first makes the window functions easy to read:

```
WITH tag_counts AS (
  SELECT t.name, COUNT(pt.post_id) AS post_count
  FROM tags t
  JOIN post_tags pt ON pt.tag_id = t.id
  GROUP BY t.id, t.name
)
SELECT
  name,
  post_count,
  RANK() OVER (ORDER BY post_count DESC) AS rnk,
  SUM(post_count) OVER (ORDER BY post_count DESC, name ASC) AS running_total
FROM tag_counts
ORDER BY post_count DESC, name ASC
LIMIT 8;
```

```
+----------+------------+-----+---------------+
| name     | post_count | rnk | running_total |
+----------+------------+-----+---------------+
| js       |        349 |   1 |           349 |
| ai       |        211 |   2 |           560 |
| node     |        151 |   3 |           711 |
| platform |        100 |   4 |           811 |
| react    |         99 |   5 |           910 |
| css      |         87 |   6 |           997 |
| cli      |         86 |   7 |          1083 |
```

```
| python   |         70 |   8 |          1153 |
+----------+------------+-----+---------------+
```

Notice the deterministic tie-breaker (`ORDER BY post_count DESC, name ASC`) inside `SUM(...) OVER (...)`. Without a tie-breaker, two tags with the same `post_count` would be treated as peers and both receive the same running sum instead of stepping cleanly row by row.

Generating date series with recursive CTEs

Unlike PostgreSQL's `generate_series()`, MySQL does not have a built-in set-returning function for dates. Whenever you build a daily analytics chart and want days with zero events to still appear as `0`, use a **recursive CTE**:

```
WITH RECURSIVE days AS (
  SELECT DATE('2026-11-22') AS day
  UNION ALL
  SELECT day + INTERVAL 1 DAY FROM days WHERE day < '2026-11-28'
)
SELECT
  d.day,
  COUNT(v.id) AS views
FROM days d
LEFT JOIN post_views v
  ON v.viewed_at >= d.day AND v.viewed_at < d.day + INTERVAL 1 DAY
GROUP BY d.day
ORDER BY d.day;
```

```
+------------+-------+
| day        | views |
+------------+-------+
| 2026-11-22 |   650 |
| 2026-11-23 |   649 |
| 2026-11-24 |   649 |
| 2026-11-25 |   649 |
| 2026-11-26 |   649 |
| 2026-11-27 |   650 |
| 2026-11-28 |   649 |
+------------+-------+
```

Notice two important details in that join:

- We wrote `v.viewed_at >= d.day AND v.viewed_at < d.day + INTERVAL 1 DAY` instead of `DATE(v.viewed_at) = d.day`. Calling `DATE()` on the column prevents MySQL from using an index on `viewed_at`.
- MySQL caps recursive CTEs at `cte_max_recursion_depth` (1,000 iterations by default) so an accidental infinite loop fails fast instead of hanging the server.

InnoDB transactions, locking, and implicit DDL commits

InnoDB supports full ACID transactions with MVCC snapshots (`START TRANSACTION`, `COMMIT`, `ROLLBACK`). Two behaviors catch people coming from PostgreSQL or SQLite by surprise.

Implicit commits on DDL

In PostgreSQL and SQLite, `CREATE TABLE` and `ALTER TABLE` can be rolled back inside a transaction. **In MySQL, any DDL statement silently commits your open transaction first.**

Watch what happens when I insert a row, run `CREATE TABLE`, and then `ROLLBACK`:

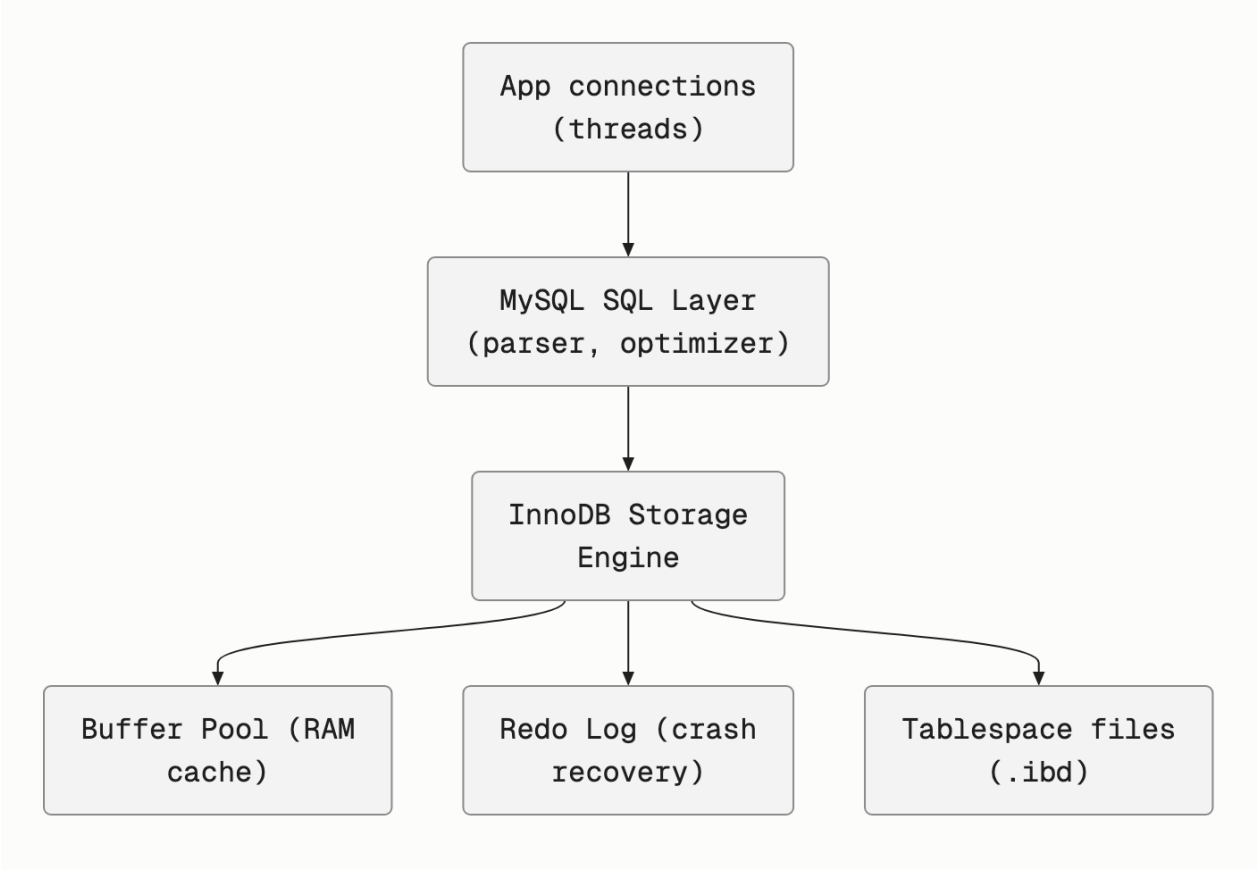
```
START TRANSACTION;
INSERT INTO page_views (slug, views) VALUES ('implicit-test', 10);
CREATE TABLE ddl_commits (id INT PRIMARY KEY);
ROLLBACK;
SELECT * FROM page_views WHERE slug = 'implicit-test';
```

```
+---------------+-------+
| slug          | views |
+---------------+-------+
| implicit-test |    10 |
+---------------+-------+
```

`'implicit-test'` survived the `ROLLBACK` because `CREATE TABLE` had already committed the transaction! Keep DDL statements out of application transactions, and design schema migrations knowing each `ALTER TABLE` commits immediately.

`REPEATABLE READ` and InnoDB locking

MySQL's default isolation level is `REPEATABLE READ`: every `SELECT` inside a transaction sees the snapshot from its first read. When a transaction updates rows, InnoDB acquires row-level locks (and gap locks on ranges to prevent phantom rows).



You can inspect every active InnoDB lock in real time using `performance_schema.data_locks`. Let's watch it happen with two concurrent sessions.

In Session A, start a transaction and update the `'js'` tag without committing yet:

```
START TRANSACTION;
UPDATE tags SET name = 'js' WHERE name = 'js';
```

While Session A is still open, query `performance_schema.data_locks` from a second terminal:

```
SELECT OBJECT_NAME, INDEX_NAME, LOCK_TYPE, LOCK_MODE, LOCK_STATUS, LOCK_DATA
FROM performance_schema.data_locks
WHERE OBJECT_SCHEMA = 'blog';
```

```
+-------------+------------+-----------+---------------+-------------+-----------+
| OBJECT_NAME | INDEX_NAME | LOCK_TYPE | LOCK_MODE     | LOCK_STATUS | LOCK_DATA |
+-------------+------------+-----------+---------------+-------------+-----------+
| tags        | NULL       | TABLE     | IX            | GRANTED     | NULL      |
| tags        | name       | RECORD    | X,REC_NOT_GAP | GRANTED     | 'js', 24  |
| tags        | PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 24        |
+-------------+------------+-----------+---------------+-------------+-----------+
```

Look closely at those three rows:

- `TABLE` with `IX` (intention exclusive): tells InnoDB that a transaction holds a row lock inside `tags`.

- `RECORD` on the unique `name` index (`'js', 24`) with `X,REC_NOT_GAP`: an exclusive record lock on the exact matching entry (no gap lock needed because `name` is `UNIQUE`).
- `RECORD` on `PRIMARY` (`24`): the exclusive lock on the clustered primary key row itself.

Now, while Session A still holds that lock, Session B lowers its lock timeout to 2 seconds and tries to update the same row:

```
SET SESSION innodb_lock_wait_timeout = 2;
START TRANSACTION;
UPDATE tags SET name = 'javascript-new' WHERE name = 'js';
```

```
ERROR 1205 (HY000) at line 4: Lock wait timeout exceeded; try restarting transaction
```

By default, `innodb_lock_wait_timeout` is **50 seconds**, which means a stuck transaction can freeze incoming web requests for nearly a minute before throwing `ERROR 1205`.

Deadlocks in InnoDB

What if Session 1 locks row A and wants row B, while Session 2 locks row B and wants row A? Neither session can make progress.

InnoDB runs an automatic deadlock detector. The moment it sees a cycle, it picks the smaller transaction as the victim, rolls it back immediately, and returns `ERROR 1213` so the other session can commit.

I reproduced a real deadlock on `blog` with two concurrent sessions updating two rows (`dl_a` and `dl_b`) in opposite order:

- **Session 1**: `UPDATE tags ... WHERE name = 'dl_a'`, wait, then `UPDATE tags ... WHERE name = 'dl_b'`.
- **Session 2**: `UPDATE tags ... WHERE name = 'dl_b'`, wait, then `UPDATE tags ... WHERE name = 'dl_a'`.

Session 2 committed cleanly, and Session 1 immediately received:

```
ERROR 1213 (40001) at line 5: Deadlock found when trying to get lock; try restarting transaction
```

Three habits eliminate almost all lock contention and deadlocks in real apps:

1. **Keep transactions short.** Never hold a transaction open while calling an external HTTP API or sending an email.
2. **Update rows in a consistent order.** If two endpoints both update a parent row and a child row, or two accounts in a transfer, always lock them in ascending primary key order (`ORDER BY id`).
3. **Retry `ER_LOCK_DEADLOCK` (`1213`)** in application code with a short jittered backoff (2–3 retries) when running high-concurrency writes.

Isolation levels compared

Level	Dirty reads?	Non-repeatable reads?	Gap locks in InnoDB?	Notes
`READ UNCOMMITTED`	Yes	Yes	Minimal	Rarely used; reads uncommitted changes
`READ COMMITTED`	No	Yes	Mostly disabled	Each statement gets a fresh snapshot; popular for high-concurrency OLTP
`REPEATABLE READ`	No	No	Yes	MySQL default; consistent snapshot for the whole transaction
`SERIALIZABLE`	No	No	Heavy (shared locks on reads)	Strictest; converts plain `SELECT` into `SELECT ... FOR SHARE`

If your workload hits frequent gap-lock deadlocks on range updates and you don't need transaction-wide snapshot reads, switching the session or server to `READ COMMITTED` is a well-trodden MySQL tuning move.

Indexes, `EXPLAIN`, and `EXPLAIN ANALYZE`

In InnoDB, every table is physically organized as a **clustered index** on its `PRIMARY KEY`. That means:

- Lookups by primary key are single B-tree lookups.
- Every secondary index stores the primary key value in its leaf nodes (which is another reason to keep primary keys compact, like `BIGINT UNSIGNED`, not huge random strings).

To measure real index impact, I inserted **200,000 rows** into `post_views` in 480 ms.

Before and after a composite covering index

First, run `EXPLAIN` and `EXPLAIN ANALYZE` with no index on `(post_id, viewed_at)`:

```
EXPLAIN SELECT COUNT(*) FROM post_views
WHERE post_id = 732 AND viewed_at >= '2026-11-01';
```

```
+----+-------------+------------+------+---------------+------+---------+------+--------+----------+---
| id | select_type | table      | type | possible_keys | key  | key_len | ref  | rows   | filtered |
    ↪ Extra       |
+----+-------------+------------+------+---------------+------+---------+------+--------+----------+---
|  1 | SIMPLE      | post_views | ALL  | NULL          | NULL | NULL    | NULL | 200000 |     3.33 |
    ↪ Using where |
+----+-------------+------------+------+---------------+------+---------+------+--------+----------+---
```

`type: ALL` means a full table scan of all 200,000 rows. `EXPLAIN ANALYZE` runs the query and measures real time:

```
-> Aggregate: count(0) (actual time=19.5..19.5 rows=1 loops=1)
    -> Filter: ((post_views.post_id = 732) and (post_views.viewed_at >= TIMESTAMP'2026-11-01 00:00:00')
    ↪ ) (actual time=2.22..19.5 rows=8 loops=1)
        -> Table scan on post_views (actual time=0.323..15.9 rows=200000 loops=1)
```

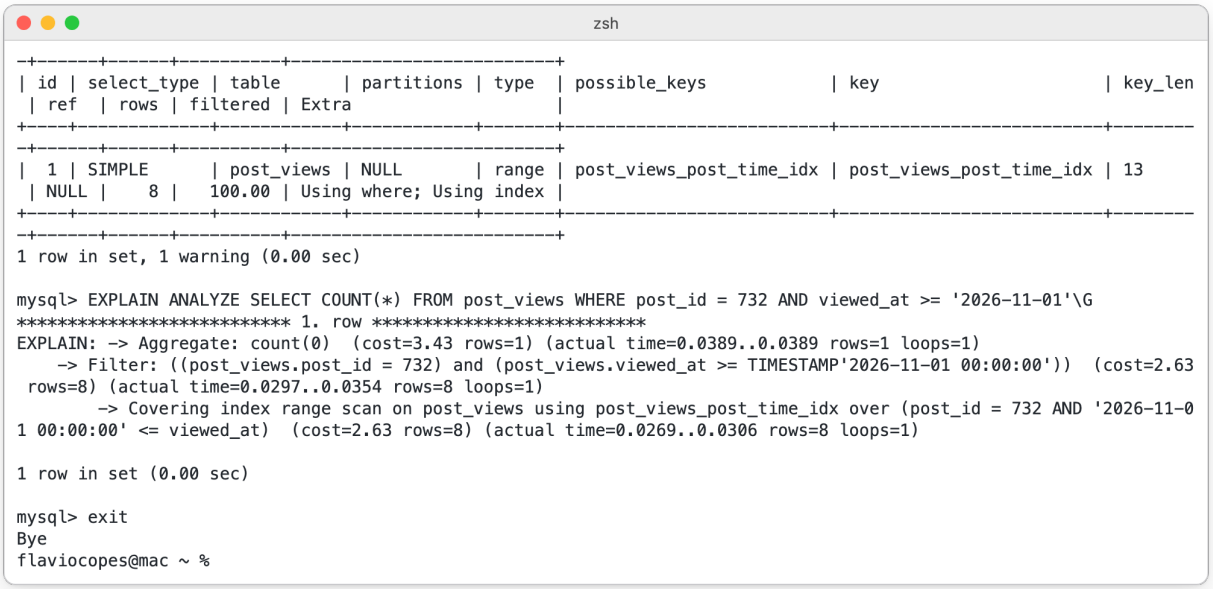
19.5 milliseconds to find 8 rows. Now add a composite index on `(post_id, viewed_at)`:

```
CREATE INDEX post_views_post_time_idx ON post_views (post_id, viewed_at);
EXPLAIN SELECT COUNT(*) FROM post_views
WHERE post_id = 732 AND viewed_at >= '2026-11-01';
```

```
+----+-------------+------------+-------+--------------------------+--------------------------+--------
| id | select_type | table      | type  | possible_keys            | key                      | key_len
    ↪  | ref  | rows | filtered | Extra                    |
+----+-------------+------------+-------+--------------------------+--------------------------+--------
|  1 | SIMPLE      | post_views | range | post_views_post_time_idx | post_views_post_time_idx | 13
    ↪  | NULL |    8 |   100.00 | Using where; Using index |
+----+-------------+------------+-------+--------------------------+--------------------------+--------
```

```
-> Aggregate: count(0) (cost=3.43 rows=1) (actual time=0.00646..0.0065 rows=1 loops=1)
    -> Covering index range scan on post_views using post_views_post_time_idx over (post_id = 732 AND
    ↪ '2026-11-01 00:00:00' <= viewed_at) (actual time=0.00342..0.00396 rows=8 loops=1)
```

From **19.5 ms** down to **0.0065 ms** (6.5 microseconds, **3,000× faster!**). `Extra: Using index` in MySQL means a covering index scan: InnoDB answered the query from the index alone without reading the table rows.



Reading `EXPLAIN` access types (best to worst)

`type`	Meaning
`const`	Primary key or unique index lookup returning at most 1 row

type	Meaning
eq_ref	Primary key or unique lookup inside a join
ref	Non-unique index lookup
range	Index range scan (>, BETWEEN, IN)
index	Full index scan (reads entire index)
ALL	Full table scan (reads every row)

Composite indexes and the leftmost prefix rule

Our index `post_views_post_time_idx (post_id, viewed_at)` worked because our `WHERE` clause filtered on `post_id = 732` (equality) first and `viewed_at >= '2026-11-01'` (range) second.

MySQL B-tree indexes follow the **leftmost prefix rule**:

- An index on `(post_id, viewed_at)` accelerates `WHERE post_id = ?` and `WHERE post_id = ? AND viewed_at >= ?`.
- It does **not** accelerate `WHERE viewed_at >= ?` alone, because `viewed_at` is sorted inside each `post_id`, not globally across the index.
- Put **equality columns first** (=), then **range columns** (>, <, `BETWEEN`) or `ORDER BY` columns last.

Common reasons MySQL ignores an index

Even when an index exists, four common patterns force InnoDB back to a full table scan:

1. **Wrapping an indexed column in a function:** `WHERE YEAR(published_on) = 2026` cannot use an index on `published_on`. Rewrite it as a range: `WHERE published_on >= '2026-01-01' AND published_on < '2027-01-01'`.
2. **Leading wildcards in `LIKE`:** `WHERE title LIKE 'mysql%'` uses a B-tree index, while `WHERE title LIKE '%mysql%'` cannot. For substring or word search, use a `FULLTEXT` index instead.
3. **Implicit type conversion:** Comparing a `VARCHAR` column (`slug`) to an integer (`WHERE slug = 123`) forces MySQL to cast every row's string to a number, skipping the index.
4. **Low selectivity:** If a query matches 30–40% of the table (for example, `WHERE is_published = 1` when 98% of rows are published), InnoDB's optimizer intentionally chooses a sequential table scan because random index lookups would be slower.

Using MySQL from Node.js (`mysql2`)

For Node.js, use `mysql2/promise`. It supports `caching_sha2_password` natively, prepared statements, and bounded connection pools.

```
npm install mysql2
```

Bounded pools, `dateStrings`, and parameterized queries

Create one pool at startup and share it across your app. Never open a new connection per HTTP request:

```
import http from 'node:http'
import mysql from 'mysql2/promise'

const pool = mysql.createPool({
  host: '127.0.0.1',
  port: 3318,
  user: 'blog_app',
  password: process.env.MYSQL_PASSWORD,
  database: 'blog',
  connectionLimit: 10,
  waitForConnections: true,
  queueLimit: 50,
  dateStrings: true,
})
```

Three options worth highlighting:

- `connectionLimit: 10` caps open connections from this process so traffic spikes can't exhaust MySQL's `max_connections` (151 by default).
- `queueLimit: 50` rejects excess waiting requests fast instead of letting memory grow forever during an outage.
- `dateStrings: true` returns `DATE` and `DATETIME` columns as strings (`'2026-10-12'`) instead of JavaScript `Date` objects shifted by local timezone.

Always use `pool.execute(sql, params)` with `?` placeholders so user input is bound as data, never parsed as SQL:

```
const input = "x' OR '1'='1"
const [bad] = await pool.query(`SELECT COUNT(*) AS n FROM posts WHERE slug = '${input}'`)
const [good] = await pool.execute('SELECT COUNT(*) AS n FROM posts WHERE slug = ?', [input])
console.log({ interpolated: bad[0].n, parameterized: good[0].n })
```

```
{ interpolated: 2178, parameterized: 0 }
```

A working HTTP search API with `mysql2`

Here is our full-text search endpoint over the 2,178 posts, with graceful `SIGTERM` shutdown:

```
const server = http.createServer(async (req, res) => {
  const url = new URL(req.url, 'http://localhost')
  try {
    if (req.method === 'GET' && url.pathname === '/search') {
      const q = (url.searchParams.get('q') ?? '').trim()
      const limit = Math.min(Number(url.searchParams.get('limit')) || 3, 50)
      if (!q) {
        res.writeHead(400, { 'content-type': 'application/json' })
        return res.end(JSON.stringify({ error: 'missing q' }) + '\n')
      }
```

```
      const [rows] = await pool.execute(
        `SELECT slug, title, published_on,
                ROUND(MATCH(title, body) AGAINST (? IN NATURAL LANGUAGE MODE), 2) AS score
         FROM posts
         WHERE MATCH(title, body) AGAINST (? IN NATURAL LANGUAGE MODE)
         ORDER BY score DESC
         LIMIT ?`,
        [q, q, String(limit)],
      )
      res.writeHead(200, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ q, count: rows.length, rows }, null, 2) + '\n')
    }
    res.writeHead(404, { 'content-type': 'application/json' })
    res.end(JSON.stringify({ error: 'not found' }) + '\n')
  } catch (err) {
    res.writeHead(500, { 'content-type': 'application/json' })
    res.end(JSON.stringify({ error: err.message }) + '\n')
  }
})

server.listen(3030, () => console.log('listening on http://localhost:3030'))

process.on('SIGTERM', () => {
  server.close(async () => {
    await pool.end()
    console.log('pool closed')
    process.exit(0)
  })
})
```

Testing it with `curl`:

```
curl -s 'http://localhost:3030/search?q=mysql+innodb&limit=2'
```

```
{
  "q": "mysql innodb",
  "count": 2,
  "rows": [
    {
      "slug": "full-ubuntu-disk-codex",
      "title": "How I fixed a full Ubuntu server disk with Codex",
      "published_on": "2026-10-12",
      "score": 103.39
    },
    {
      "slug": "mysql-how-to-install",
      "title": "How to install MySQL on macOS",
      "published_on": "2020-01-05",
      "score": 83.16
    }
  ]
}
```

`pool.execute()` vs `pool.query()`

`mysql2` gives you two methods that both accept `?` parameters safely, with one subtle difference:

- **`pool.execute(sql, params)`** uses MySQL's binary prepared statement protocol and caches the prepared statement per connection. It is fast for repeated queries and strict about parameter types (for `LIMIT ?`, pass `String(limit)` in `execute` or use `query`).
- **`pool.query(sql, params)`** interpolates and escapes parameters client-side before sending text SQL. Use it for dynamic `IN (?)` arrays (`pool.query('SELECT * FROM tags WHERE name IN (?)', [['js', 'node']])`) or DDL statements that MySQL doesn't allow in prepared statements.

Never build SQL strings with template literals (`SELECT ... WHERE slug = '${input}'`). Always pass values through the second argument array.

Multi-statement transactions and integration tests in `mysql2`

When a flow needs multiple statements to succeed or fail together, borrow one dedicated connection from the pool with `pool.getConnection()`, run `beginTransaction()`, and **always** call `conn.release()` in `finally`. Forgetting `conn.release()` leaks a connection out of the pool on every error until the pool stalls.

Here is a real `createPostWithTag` function and a `node:test` integration test (`tx.test.mjs`) that verifies both an atomic commit and a full rollback when a duplicate slug triggers `ER_DUP_ENTRY`:

```
import { describe, it, after } from 'node:test'
import assert from 'node:assert/strict'
import mysql from 'mysql2/promise'

const pool = mysql.createPool({
  host: '127.0.0.1',
  port: 3318,
  user: 'blog_app',
  password: 'app-secret-123',
  database: 'blog',
  connectionLimit: 5,
  dateStrings: true,
})

async function createPostWithTag(slug, title, tagName) {
  const conn = await pool.getConnection()
  try {
    await conn.beginTransaction()
    const [res] = await conn.execute(
      "INSERT INTO posts (slug, title, published_on, words, body, frontmatter) VALUES (?, ?, CURDATE(),
    ↪  12, 'Body text', JSON_OBJECT('tag', ?))",
      [slug, title, tagName],
    )
    await conn.execute(
      'INSERT INTO tags (name) VALUES (?) ON DUPLICATE KEY UPDATE id = LAST_INSERT_ID(id)',
      [tagName],
    )
    const [[tagRow]] = await conn.execute('SELECT LAST_INSERT_ID() AS id')
    await conn.execute(
      'INSERT INTO post_tags (post_id, tag_id) VALUES (?, ?)',
      [res.insertId, tagRow.id],
    )
    await conn.commit()
    return res.insertId
  } catch (err) {
    await conn.rollback()
    throw err
```

```
  } finally {
    conn.release()
  }
}

describe('mysql2 transactions', () => {
  after(async () => {
    await pool.execute("DELETE FROM posts WHERE slug LIKE 'test-tx-%'")
    await pool.end()
  })

  it('commits post and tag link atomically', async () => {
    const id = await createPostWithTag('test-tx-ok', 'Tx Test', 'mysql')
    const [rows] = await pool.execute(
      'SELECT p.slug, t.name FROM posts p JOIN post_tags pt ON pt.post_id = p.id JOIN tags t ON t.id =
    ↪ pt.tag_id WHERE p.id = ?',
      [id],
    )
    assert.equal(rows.length, 1)
    assert.equal(rows[0].slug, 'test-tx-ok')
  })

  it('rolls back on duplicate slug error', async () => {
    await assert.rejects(
      () => createPostWithTag('test-tx-ok', 'Duplicate', 'brand-new-rollback-tag'),
      { code: 'ER_DUP_ENTRY' },
    )
    const [tags] = await pool.execute(
      "SELECT id FROM tags WHERE name = 'brand-new-rollback-tag'",
    )
    assert.equal(tags.length, 0)
  })
})
```

Running it with Node.js 22's built-in test runner:

```
node --test tx.test.mjs
```

```
# tests 2
# suites 1
# pass 2
# fail 0
# duration_ms 99.306125
```

Notice the `ON DUPLICATE KEY UPDATE id = LAST_INSERT_ID(id)` idiom on `tags`. If the tag already exists, MySQL still populates `LAST_INSERT_ID()` with the existing tag's ID so the next statement can link it cleanly without an extra `SELECT`.

Observability, storage internals, and server tuning

Once a MySQL database is running in production, you need to know three things: how big your tables and indexes are, which queries spend the most time, and which `my.cnf` settings actually move the needle.

Inspecting table and index sizes

Query `information_schema.TABLES` to see how much disk space each table and its indexes occupy:

```
SELECT
  TABLE_NAME,
  ENGINE,
  TABLE_ROWS,
  ROUND(DATA_LENGTH / 1024 / 1024, 2) AS data_mb,
  ROUND(INDEX_LENGTH / 1024 / 1024, 2) AS index_mb
FROM information_schema.TABLES
WHERE TABLE_SCHEMA = 'blog'
ORDER BY DATA_LENGTH DESC;
```

```
+------------+--------+------------+---------+----------+
| TABLE_NAME | ENGINE | TABLE_ROWS | data_mb | index_mb |
+------------+--------+------------+---------+----------+
| posts      | InnoDB |       1653 |   22.52 |     0.28 |
| post_views | InnoDB |     199920 |    9.52 |     6.52 |
| post_tags  | InnoDB |       2224 |    0.11 |     0.11 |
| tags       | InnoDB |         49 |    0.02 |     0.02 |
+------------+--------+------------+---------+----------+
```

Look at `TABLE_ROWS` for `posts`: it says `1653`, even though we know `SELECT COUNT(*) FROM posts` is 2178! That is normal for InnoDB: `information_schema.TABLES.TABLE_ROWS` is a fast statistical estimate sampled from index pages, never an exact count. Run `ANALYZE TABLE posts;` if statistics drift far enough to confuse the query optimizer.

Also notice `post_views`: 9.52 MB of data and 6.52 MB of index. Secondary indexes are not free. Every extra index speeds up reads while adding disk space, buffer pool pressure, and write overhead on every `INSERT` and `UPDATE`.

Finding slow queries with `performance_schema`

MySQL 8.4 enables `performance_schema` by default and aggregates every normalized query digest automatically. To find the queries that consumed the most total time in `blog`:

```
SELECT
  LEFT(DIGEST_TEXT, 65) AS query,
  COUNT_STAR AS calls,
  ROUND(AVG_TIMER_WAIT / 1e9, 2) AS avg_ms,
  SUM_ROWS_EXAMINED AS rows_examined
FROM performance_schema.events_statements_summary_by_digest
WHERE SCHEMA_NAME = 'blog'
ORDER BY SUM_TIMER_WAIT DESC
LIMIT 6;
```

```
+-------------------------------------------------------------------+-------+---------+---------------+
| query                                                             | calls | avg_ms  | rows_examined |
+-------------------------------------------------------------------+-------+---------+---------------+
| SELECT `SLEEP` (?)                                                |     3 | 1836.99 |             3 |
| UPDATE `tags` SET NAME = ? WHERE NAME = ?                         |     6 |  434.49 |             4 |
| COMMIT                                                            |     3 |  259.43 |             0 |
| INSERT INTO `post_views` ( `post_id` , `viewed_at` , `country` )  |   100 |    4.28 |             0 |
```

```
| WITH RECURSIVE `days` AS ( SELECT DATE (?) AS SQL_TSI_DAY UNION A |     1 |   97.08 |        200014 |
| CREATE INDEX `post_views_post_time_idx` ON `post_views` ( `post_i |     1 |   87.93 |             0 |
+-------------------------------------------------------------------+-------+---------+---------------+
```

Every single number tells a story: the 100 batched inserts into `post_views` averaged 4.28 ms each, and the recursive CTE calendar query examined 200,014 rows in 97.08 ms because `post_views` has no standalone index on `viewed_at`.

Checking defaults and tuning `my.cnf`

Before changing any server setting, check what MySQL 8.4 ships with out of the box:

```
SHOW VARIABLES WHERE Variable_name IN (
  'innodb_buffer_pool_size',
  'innodb_redo_log_capacity',
  'max_connections',
  'slow_query_log',
  'long_query_time',
  'log_bin',
  'binlog_format',
  'innodb_lock_wait_timeout'
);
```

```
+---------------------------+-----------+
| Variable_name             | Value     |
+---------------------------+-----------+
| binlog_format             | ROW       |
| innodb_buffer_pool_size   | 134217728 |
| innodb_lock_wait_timeout  | 50        |
| innodb_redo_log_capacity  | 104857600 |
| log_bin                   | ON        |
| long_query_time           | 10.000000 |
| max_connections           | 151       |
| slow_query_log            | OFF       |
+---------------------------+-----------+
```

Out of the box, MySQL gives InnoDB only **128 MB** of buffer pool (`134217728` bytes), leaves `slow_query_log` turned `OFF`, and sets `long_query_time` to 10 seconds.

On a dedicated 4 GB or 8 GB Linux server, a sensible starting `[mysqld]` block in `/etc/mysql/mysql.conf.d/mysqld.cnf` looks like this:

```
[mysqld]
# Dedicated DB server: 60-70% of RAM (e.g. 2G on a 4 GB box, 5G on an 8 GB box)
innodb_buffer_pool_size = 2G
innodb_redo_log_capacity = 512M
max_connections = 200

# Turn on slow query logging for anything over 500 ms
slow_query_log = ON
```

```
slow_query_log_file = /var/log/mysql/mysql-slow.log
long_query_time = 0.5

# Keep binary logs from filling the disk (7 days in seconds)
binlog_expire_logs_seconds = 604800
```

That last line (`binlog_expire_logs_seconds`) is a lifesaver. Notice `log_bin = ON` and `binlog_format = ROW` are enabled by default in MySQL 8.4 (the default retention is 30 days). On write-heavy servers, unpruned binary logs are one of the most common reasons a VPS runs out of disk space.

Backups, restores, and migrations

Logical backups with `mysqldump`

For InnoDB tables, always pass `--single-transaction` so `mysqldump` takes a consistent snapshot without locking your application's writes:

```
mysqldump -u root --single-transaction --routines --triggers blog > backups/blog.sql
```

On my Mac, dumping all 2,178 posts and 200,000 `post_views` rows wrote a **24 MB** SQL file in **0.17 seconds**.

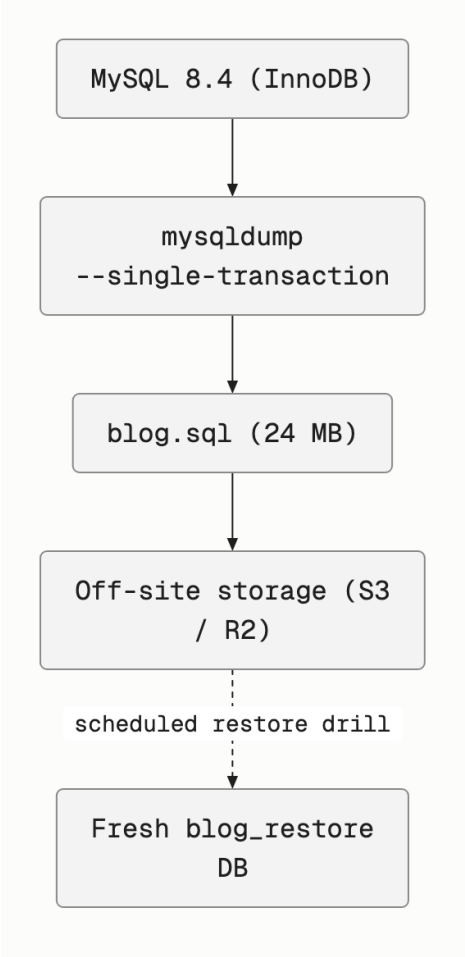
Restoring and verifying

A backup isn't real until you restore it into a clean database and check the counts:

```
mysql -u root -e "CREATE DATABASE blog_restore CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;"
mysql -u root blog_restore < backups/blog.sql
mysql -u root -t blog_restore -e "
  SELECT (SELECT COUNT(*) FROM posts) AS posts,
         (SELECT COUNT(*) FROM post_views) AS views,
         (SELECT COUNT(*) FROM tags) AS tags;
"
```

```
+-------+--------+------+
| posts | views  | tags |
+-------+--------+------+
|  2178 | 200000 |   49 |
+-------+--------+------+
```

Restoring the 24 MB dump took **1.93 seconds**, and every table count matched. For large production databases (tens or hundreds of gigabytes), use parallel dumps with `mysqlsh` (`util.dumpInstance()`) or physical backups with Percona XtraBackup.



Binary logs and point-in-time recovery (PITR)

A nightly `mysqldump` lets you recover to midnight, which means you could still lose hours of writes if a mistake happens at 3 PM.

MySQL solves this with the **binary log** (`log_bin = ON`, enabled by default in MySQL 8.4). Every committed write is recorded sequentially in the binary log files (`binlog.000001`, `binlog.000002`, and so on).

When you take a nightly backup with `--single-transaction --source-data=2`, `mysqldump` writes the exact binary log file and position as a comment at the top of `blog.sql`. To recover to 14:59:00 right before an accidental `DELETE`:

1. Restore the nightly `blog.sql` dump into a fresh database.
2. Replay the archived binary logs from the recorded position up to `--stop-datetime="2026-11-28 14:59:00"` using `mysqlbinlog`:

```
mysqlbinlog --start-position=157 --stop-datetime="2026-11-28 14:59:00" \
  /var/lib/mysql/binlog.000014 | mysql -u root blog_restore
```

Binary logs are also how MySQL **replication** works: a replica connects to the primary, streams row-level binlog events (`binlog_format = ROW`), and applies them continuously so you can offload read-heavy analytics queries or fail over if the primary host goes down.

Safe schema migrations in MySQL

Remember that `ALTER TABLE` commits immediately in MySQL. Before running an `ALTER TABLE` on a large production table, check which algorithm InnoDB will use:

Algorithm	How InnoDB executes it	Blocks concurrent writes?	Typical examples
`INSTANT`	Metadata-only change in data dictionary (milliseconds)	No	Adding or dropping a nullable column, renaming a column, adding a `VIRTUAL` generated column
`INPLACE`	Rebuilds indexes or table in place while logging concurrent DML	Briefly at start/end	Adding or dropping a secondary index, changing `VARCHAR` length within same byte-length range
`COPY`	Copies every row into a new temporary table	Yes	Changing a column data type (`INT` to `BIGINT`), changing primary key, converting character set

You can force MySQL to fail fast instead of locking a busy table by adding `ALGORITHM=INSTANT` or `ALGORITHM=INPLACE, LOCK=NONE`:

```
ALTER TABLE posts
  ADD COLUMN summary VARCHAR(500) NULL,
  ALGORITHM=INSTANT;
```

If InnoDB cannot perform the operation with `INSTANT`, the statement aborts immediately with an error instead of stalling production traffic for ten minutes.

When writing migrations:

1. Keep each migration file small, numbered (`001_init.sql`, `002_add_views.sql`), and tracked in a `schema_migrations` table.
2. Run migrations with a dedicated migration account that holds `ALTER`, `CREATE`, and `DROP`, while keeping the app account limited to `SELECT, INSERT, UPDATE, DELETE`.
3. Test destructive changes on a restored copy of production data first. For multi-gigabyte tables where `ALTER TABLE` requires `COPY`, use online schema migration tools like `gh-ost` or `pt-online-schema-change`.

Production readiness checklist

Before putting a MySQL 8.4 database into production, run through this short list:

- **Charset and collation:** Verify databases and tables use `utf8mb4` and `utf8mb4_0900_ai_ci` so emoji and international text never trigger `ERROR 1366`.
- **Accounts and network:** Bind `mysqld` to a private interface or localhost, require TLS for remote connections, and grant the app user only `SELECT, INSERT, UPDATE, DELETE` on its own database.
- **Connection pooling:** Use one bounded pool per app process (`connectionLimit: 10`, `queueLimit: 50`) so traffic spikes stay well below `max_connections`.
- **Buffer pool and logs:** Size `innodb_buffer_pool_size` to 60–70% of RAM on a dedicated DB box, turn on `slow_query_log` with `long_query_time = 0.5`, and set `binlog_expire_logs_seconds = 604800` so binary logs never fill the disk.
- **Query hygiene:** Parameterize every query, run `EXPLAIN ANALYZE` on hot endpoints before launch, and keep transactions short with retry handling for `ER_LOCK_DEADLOCK` (`1213`).
- **Backups:** Schedule `--single-transaction` dumps to off-site storage (R2 or S3) and run automated restore drills into a temporary database to verify row counts.

MySQL vs PostgreSQL vs MariaDB

People often ask whether to pick MySQL, MariaDB, or PostgreSQL for a new project. Here is how they compare in practice:

Dimension	MySQL 8.4 LTS	MariaDB	PostgreSQL 18
Primary license	GPLv2 (+ Oracle commercial)	GPLv2 (community fork)	Permissive PostgreSQL License
Default isolation	`REPEATABLE READ`	`REPEATABLE READ`	`READ COMMITTED`
Transactional DDL	No (DDL commits implicitly)	No	Yes
Inline `REFERENCES`	Silently ignored	Silently ignored	Enforced
Default string comparison	Case-insensitive (`utf8mb4_0900_ai_ci`)	Case-insensitive	Case-sensitive
Architecture	Threaded (`mysqld`), pluggable engines	Threaded, InnoDB + Aria/ColumnStore	Multi-process

Dimension	MySQL 8.4 LTS	MariaDB	PostgreSQL 18
Ecosystem & hosting	Everywhere (RDS, PlanetScale, DigitalOcean, WordPress/Laravel)	Linux distros, RDS, SkySQL	Everywhere (RDS, Supabase, Neon, Railway)

Choose **MySQL 8.4** when your stack or team already runs on it (WordPress, Laravel, existing MySQL apps, or MySQL-specialized hosting), or when you want a rock-solid threaded InnoDB server with huge operational tooling. Choose **PostgreSQL** when starting greenfield and you want transactional DDL, richer data types, and extensions like `pgvector` or PostGIS.