

advanced mysql queries with examples

Published: September 19, 2026 | Index ID: #-

Introduction

When you first learn MySQL, the focus is usually on simple `SELECT` statements, basic CRUD operations, and straightforward `JOINS`. As your projects grow in complexity, you'll find that the database layer must handle more demanding tasks: reporting, analytics, data transformation, and performance optimization. This is where advanced MySQL queries come into play.

In this article we'll explore a wide range of techniques that go beyond the basics. From subqueries and common table expressions (CTEs) to window functions and recursive queries, you'll see how each concept can be applied in real-world scenarios. By the end, you'll have a solid toolkit for writing efficient, maintainable, and highly expressive SQL statements.

Why Advanced MySQL Queries Matter

Modern applications demand more than just data retrieval. They need:

- Complex aggregations for dashboards and business intelligence.
- Efficient reporting that can handle millions of rows.
- Robust data transformation pipelines.
- Optimized performance to keep latency low.

Advanced queries help you meet these needs without compromising maintainability or scalability. They let you express intent clearly, reduce round-trips to the database, and harness the full power of MySQL's query engine.

Getting Ready: A Quick Recap of MySQL Basics

Before diving into advanced concepts, it's helpful to refresh key fundamentals that form the foundation:

1. Data Types: `INT`, `VARCHAR`, `DATE`, `DECIMAL`, etc.
2. Indexes: `B-tree`, `FULLTEXT`, `HASH`.
3. Joins: `INNER`, `LEFT`, `RIGHT`, `FULL OUTER` (via `UNION`).
4. Aggregations: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.
5. Subqueries: `SELECT` within `SELECT`.

Having these concepts firmly in place will make the advanced patterns easier to understand and apply.

1. Mastering Subqueries

Subqueries are a cornerstone of advanced SQL. They allow you to encapsulate a query within another, creating a powerful tool for filtering, aggregation, and transformation.

1.1. Scalar Subqueries

Return a single value that can be used in a `WHERE` clause or `SELECT` list.

```
SELECT  
    employee_id,
```

```
    first_name,  
    last_name,  
    (SELECT MAX(salary) FROM salaries WHERE employee_id = e.employee_id) AS max_salary  
FROM  
    employees e;
```

1.2. Correlated Subqueries

Depend on the outer query's row, enabling row-by-row evaluation.

```
SELECT  
    department_id,  
    COUNT(*) AS num_employees  
FROM  
    employees e  
WHERE  
    salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id)  
GROUP BY  
    department_id;
```

1.3. Subqueries in the FROM Clause (Derived Tables)

Useful for creating temporary result sets that can be joined or aggregated further.

```
SELECT  
    dept.department_name,  
    avg_salary  
FROM  
    departments dept  
JOIN  
    (SELECT department_id, AVG(salary) AS avg_salary  
    FROM employees  
    GROUP BY department_id) AS dept_avg  
ON dept.department_id = dept_avg.department_id;
```

2. Common Table Expressions (CTEs)

CTEs, introduced in MySQL 8.0, provide a cleaner syntax for complex queries, especially when you need to reference a subquery multiple times.

2.1. Basic CTE

Define a named result set that can be used like a temporary table.

```
WITH dept_avg_salary AS (  
    SELECT department_id, AVG(salary) AS avg_salary
```

```

FROM employees
GROUP BY department_id
)
SELECT
    d.department_name,
    das.avg_salary
FROM
    departments d
JOIN
    dept_avg_salary das
ON d.department_id = das.department_id;

```

2.2. Recursive CTEs

Handle hierarchical data such as organizational charts or category trees.

```

WITH RECURSIVE category_path AS (
    SELECT
        category_id,
        parent_id,
        category_name,
        CAST(category_name AS CHAR(255)) AS full_path
    FROM
        categories
    WHERE
        parent_id IS NULL

    UNION ALL

    SELECT
        c.category_id,
        c.parent_id,
        c.category_name,
        CONCAT(cp.full_path, ' > ', c.category_name) AS full_path
    FROM
        categories c
    INNER JOIN
        category_path cp
    ON c.parent_id = cp.category_id
)
SELECT
    *
FROM
    category_path

```

ORDER BY

full_path;

3. Window Functions

Window functions allow you to perform calculations across a set of rows related to the current row. They're ideal for ranking, running totals, and moving averages.

3.1. ROW_NUMBER, RANK, and DENSE_RANK

Assign a unique sequential integer to rows within a partition.

SELECT

employee_id,

department_id,

salary,

ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank_in_dept

FROM

employees;

3.2. SUM() OVER() for Running Totals

Calculate cumulative sums without needing GROUP BY.

SELECT

order_id,

order_date,

amount,

SUM(amount) OVER (ORDER BY order_date ASC) AS running_total

FROM

orders

ORDER BY

order_date;

3.3. LAG and LEAD Functions

Access data from previous or next rows in the result set.

SELECT

employee_id,

salary,

LAG(salary, 1) OVER (ORDER BY hire_date) AS prev_salary,

LEAD(salary, 1) OVER (ORDER BY hire_date) AS next_salary

FROM

employees

ORDER BY

```
hire_date;
```

4. Advanced JOIN Techniques

JOINS are the backbone of relational queries. Advanced patterns help you retrieve complex relationships efficiently.

4.1. Self-JOIN for Hierarchical Data

Link a table to itself to find parent-child relationships.

```
SELECT
    c1.category_name AS child,
    c2.category_name AS parent
FROM
    categories c1
LEFT JOIN
    categories c2
ON c1.parent_id = c2.category_id;
```

4.2. Full Outer Join via UNION

MySQL lacks a native FULL OUTER JOIN. Combine LEFT and RIGHT joins with UNION to simulate it.

```
SELECT
    a.*, b.*
FROM
    table_a a
LEFT JOIN
    table_b b
ON a.id = b.a_id
```

```
UNION ALL
```

```
SELECT
    a.*, b.*
FROM
    table_a a
RIGHT JOIN
    table_b b
ON a.id = b.a_id
WHERE a.id IS NULL;
```

4.3. CROSS JOIN for Cartesian Products

Use sparingly; it's handy for generating combinations.

```
SELECT
    e.employee_id,
    d.department_id
FROM
    employees e
CROSS JOIN
    departments d;
```

5. Pivoting Data with CASE Statements

Pivoting transforms rows into columns, enabling easier reporting.

```
SELECT
    employee_id,
    SUM(CASE WHEN month = 1 THEN sales ELSE 0 END) AS sales_jan,
    SUM(CASE WHEN month = 2 THEN sales ELSE 0 END) AS sales_feb,
    SUM(CASE WHEN month = 3 THEN sales ELSE 0 END) AS sales_mar
FROM
    monthly_sales
GROUP BY
    employee_id;
```

6. Optimizing Query Performance

Advanced queries can be powerful, but they can also be heavy. Here are key strategies to keep them efficient.

6.1. Indexing Strategies

- Use composite indexes that match the query's WHERE and ORDER BY clauses.
- Avoid over-indexing; each index adds write overhead.
- Leverage covering indexes that include all columns needed for a query.

6.2. EXPLAIN and ANALYZE

Run EXPLAIN ANALYZE to understand the execution plan and spot bottlenecks.

```
EXPLAIN ANALYZE
SELECT
    *
FROM
    orders
WHERE
    order_date BETWEEN '2023-01-01' AND '2023-01-31';
```

6.3. Query Refactoring

- Replace correlated subqueries with JOINS when possible.

- Use WITH clauses to avoid repeating complex subqueries.
- Break large queries into smaller steps and materialize intermediate results in temporary tables if necessary.

6.4. Partitioning Large Tables

Partition tables by date or range to limit the data scanned.

ALTER TABLE orders

```
PARTITION BY RANGE (YEAR(order_date)) (
  PARTITION p2021 VALUES LESS THAN (2022),
  PARTITION p2022 VALUES LESS THAN (2023),
  PARTITION p2023 VALUES LESS THAN (2024)
);
```

7. Practical Use Cases

Let's look at how advanced MySQL queries solve real business problems.

7.1. Sales Performance Dashboard

Compute monthly revenue, year-over-year growth, and top-performing products.

```
WITH monthly_rev AS (
  SELECT
    DATE_FORMAT(order_date, '%Y-%m') AS month,
    SUM(amount) AS total_amount
  FROM
    orders
  GROUP BY
    month
),
yoy_growth AS (
  SELECT
    month,
    total_amount,
    LAG(total_amount) OVER (ORDER BY month) AS prev_month_amount,
    ((total_amount - LAG(total_amount) OVER (ORDER BY month)) / LAG(total_amount) OVER (ORDER BY month)) * 100 AS growth_pct
  FROM
    monthly_rev
)
SELECT
  month,
  total_amount,
  growth_pct
FROM
  yoy_growth
ORDER BY
```

month;

7.2. Employee Hierarchy and Salary Analysis

Baca Juga (Artikel Terkait)

- [advanced magic besm d20 supplement](http://www.aussizzlegal.com/advanced-magic-besm-d20-supplement.pdf)

URL: <http://www.aussizzlegal.com/advanced-magic-besm-d20-supplement.pdf>

- [advanced management accounting by kaplan](http://www.aussizzlegal.com/advanced-management-accounting-by-kaplan.pdf)

URL: <http://www.aussizzlegal.com/advanced-management-accounting-by-kaplan.pdf>

- [advanced macroeconomics solutions](http://www.aussizzlegal.com/advanced-macroeconomics-solutions.pdf)

URL: <http://www.aussizzlegal.com/advanced-macroeconomics-solutions.pdf>

- [advanced macroeconomics solutions 4th edition](http://www.aussizzlegal.com/advanced-macroeconomics-solutions-4th-edition.pdf)

URL: <http://www.aussizzlegal.com/advanced-macroeconomics-solutions-4th-edition.pdf>

Tags: [#advanced](#) [#mysql](#) [#queries](#) [#with](#) [#examples](#)

