# **Introduction to LIKE (SQL Continued)**

* **Purpose**: LIKE lets you perform simple pattern matching on string columns, using wildcard characters instead of exact equality.
* **Why it Came**: Early SQL needed a way to query text when you know only part of the value—e.g. find all users whose email domain is “gmail.com,” or products whose name contains “chair.”

**2. Rules & Syntax**

| **Wildcard** | **Meaning** |
| --- | --- |
| % | Matches any sequence of zero or more characters |
| \_ | Matches exactly one character |

column LIKE pattern [ESCAPE 'escape\_char']

* **Case sensitivity** depends on the column’s collation (e.g. utf8mb4\_general\_ci is case‑insensitive, utf8mb4\_bin is case‑sensitive).
* To search for literal % or \_, use an ESCAPE clause:

WHERE name LIKE '%50\%%' ESCAPE '\'

* Combine with NOT to exclude matches:

WHERE description NOT LIKE 'test\_%'

**3. When & Where to Use**

* **User‑driven searches** in UIs (search-as-you-type): e.g. autocomplete on product names.
* **Data cleansing**: find rows with malformed values (e.g. phone numbers not matching a pattern).
* **Reporting**: categorize or filter text data by prefix/suffix or substring.

**4. Real‑World Examples**

**Example 1: Find all users with Gmail addresses**

SELECT user\_id, email

FROM users

WHERE email LIKE '%@gmail.com';

* **Use case**: Marketing team wants to send a targeted campaign to Gmail users.

**Example 2: List products whose names start with “Ultra”**

SELECT product\_id, name, price

FROM products

WHERE name LIKE 'Ultra%';

* **Use case**: Display all “Ultra” series products in an e‑commerce filter.

**Example 3: Identify phone numbers in non‑standard formats**

-- Expecting (123) 456‑7890 format

SELECT customer\_id, phone

FROM customers

WHERE phone NOT LIKE '(\_\_\_) \_\_\_-\_\_\_\_';

* **Use case**: Data‑quality check to find entries that need normalization.

**Example 4: Search for testimonials containing “excellent” anywhere**

SELECT testimonial\_id, content

FROM reviews

WHERE content LIKE '%excellent%';

* **Use case**: Customer‑success team extracts all reviews praising the product.

**5. Using LIKE Safely in JDBC**

When using user input in a LIKE pattern, always use PreparedStatement to avoid SQL injection and correctly escape wildcards:

String sql =

"SELECT \* FROM products WHERE name LIKE CONCAT('%', ?, '%')";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, searchTerm); // safely binds user input

try (ResultSet rs = ps.executeQuery()) {

while (rs.next()) {

// process matching products

}

}

}

* **Why**: CONCAT('%', ?, '%') ensures the % wildcards are outside the bound parameter, preventing malicious input from breaking your pattern logic.

**BETWEEN, NOT BETWEEN, and IN Operators**

**What & Why They Came**

* **BETWEEN…AND** and **NOT BETWEEN** simplify range checks.
* **IN (…)** and **NOT IN (…)** provide set membership checks.  
  These constructs were introduced early in SQL to make filtering more expressive and readable than chained comparisons.

**Rules & Syntax**

* expr BETWEEN low AND high is equivalent to expr ≥ low AND expr ≤ high.
* expr NOT BETWEEN low AND high negates that range.
* expr IN (val1, val2, …) tests equality against multiple discrete values.
* expr NOT IN (…) excludes those values.
* Watch out: if any member is NULL, IN may yield UNKNOWN; use IS NULL separately.

**When & Where to Use**

* **BETWEEN**: date ranges, numeric thresholds.
* **IN**: matching against a small lookup set (e.g. status codes).

**Real‑World Examples**

1. **Filter orders in Q1 2025**

SELECT order\_id, order\_date, total\_amount

FROM orders

WHERE order\_date BETWEEN '2025-01-01' AND '2025-03-31';

String sql =

"SELECT order\_id, order\_date, total\_amount " +

"FROM orders WHERE order\_date BETWEEN ? AND ?";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setDate(1, Date.valueOf("2025-01-01"));

ps.setDate(2, Date.valueOf("2025-03-31"));

try (ResultSet rs = ps.executeQuery()) { … }

}

1. **Exclude discontinued or out‑of‑stock products**

sql

SELECT product\_id, name

FROM products

WHERE status NOT IN ('discontinued', 'out\_of\_stock');

java

String sql =

"SELECT product\_id, name FROM products " +

"WHERE status NOT IN (?, ?)";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, "discontinued");

ps.setString(2, "out\_of\_stock");

…

}

1. **Filter customers by age bracket**

SELECT customer\_id, name, age

FROM customers

WHERE age BETWEEN 18 AND 25;

// Similar to example 1: set two integers for the age bounds

**ORDER BY Clause**

**What & Why It Came**

* **ORDER BY** was added to SQL to let users sort result sets on one or more columns (ascending by default).

**Rules & Syntax**

* Specify column names or expressions.
* Use ASC (default) or DESC.
* Multiple levels: ORDER BY col1 DESC, col2 ASC.

**When & Where to Use**

* Presenting data in a meaningful sequence (e.g. latest records first).
* Pagination (with LIMIT/OFFSET).

**Real‑World Examples**

1. **Show top 10 customers by lifetime spend**

sql

SELECT customer\_id, name, SUM(total\_amount) AS lifetime\_spend

FROM orders

GROUP BY customer\_id

ORDER BY lifetime\_spend DESC

LIMIT 10;

java

String sql =

"SELECT customer\_id, name, SUM(total\_amount) AS lifetime\_spend " +

"FROM orders GROUP BY customer\_id " +

"ORDER BY lifetime\_spend DESC LIMIT 10";

1. **List products alphabetically**

sql

SELECT product\_id, name, price

FROM products

ORDER BY name ASC;

java

String sql = "SELECT product\_id, name, price FROM products ORDER BY name";

**Aggregate Functions: AVG, COUNT, MAX, MIN, SUM**

**What & Why They Came**

* Aggregates provide built‑in summarization (introduced in SQL‑92) for reporting and analytics.

**Functions & Rules**

* AVG(expr), SUM(expr), MIN(expr), MAX(expr), COUNT(expr|\*).
* Non‑null values only (except COUNT(\*)).
* Often paired with GROUP BY.

**When & Where to Use**

* Generating totals, averages, and other summaries for dashboards or reports.

**Real‑World Examples**

1. **Average order value per month**

sql

SELECT DATE\_FORMAT(order\_date, '%Y-%m') AS month,

AVG(total\_amount) AS avg\_order\_value

FROM orders

WHERE status = 'completed'

GROUP BY month;

java

String sql =

"SELECT DATE\_FORMAT(order\_date, '%Y-%m') AS month, " +

"AVG(total\_amount) AS avg\_order\_value " +

"FROM orders WHERE status = ? GROUP BY month";

1. **Count of active users**

sql

SELECT COUNT(\*) AS active\_users

FROM users

WHERE last\_login > NOW() - INTERVAL 30 DAY;

java

String sql =

"SELECT COUNT(\*) AS active\_users " +

"FROM users WHERE last\_login > ?";

ps.setTimestamp(1, Timestamp.valueOf(LocalDateTime.now().minusDays(30)));

1. **Max and Min product prices by category**

sql

SELECT category\_id,

MIN(price) AS lowest\_price,

MAX(price) AS highest\_price

FROM products

GROUP BY category\_id;

java

String sql =

"SELECT category\_id, MIN(price) AS lowest\_price, MAX(price) AS highest\_price " +

"FROM products GROUP BY category\_id";

1. **Total revenue per region**

SELECT region, SUM(total\_amount) AS revenue

FROM orders

GROUP BY region;

String sql =

"SELECT region, SUM(total\_amount) AS revenue " +

"FROM orders GROUP BY region";

**GROUP BY Statement**

**What & Why It Came**

* **GROUP BY** groups rows by one or more columns to allow aggregate functions to calculate per‑group metrics.

**Rules & Syntax**

* All non‑aggregated columns in SELECT must appear in GROUP BY.
* MySQL’s default ONLY\_FULL\_GROUP\_BY SQL mode enforces strict compliance.

**When & Where to Use**

* Reporting by category, time bucket, or other dimensions.

**Real‑World Examples**

1. **Orders per sales channel**

SELECT sales\_channel, COUNT(\*) AS orders\_count

FROM orders

GROUP BY sales\_channel;

1. **Monthly new user signups**

SELECT DATE\_FORMAT(signup\_date, '%Y-%m') AS month,

COUNT(\*) AS new\_users

FROM users

GROUP BY month;

**HAVING Clause**

**What & Why It Came**

* **HAVING** filters groups after aggregation (introduced because WHERE can’t use aggregates).

**Rules & Syntax**

* Placed after GROUP BY but before ORDER BY.
* Conditions can reference aggregate functions.

**When & Where to Use**

* Filtering aggregated results (e.g. only categories with ≥100 sales).

**Real‑World Examples**

1. **Categories with high average price**

SELECT category\_id, AVG(price) AS avg\_price

FROM products

GROUP BY category\_id

HAVING AVG(price) > 50;

1. **Months with more than 1,000 orders**

SELECT DATE\_FORMAT(order\_date, '%Y-%m') AS month,

COUNT(\*) AS order\_count

FROM orders

GROUP BY month

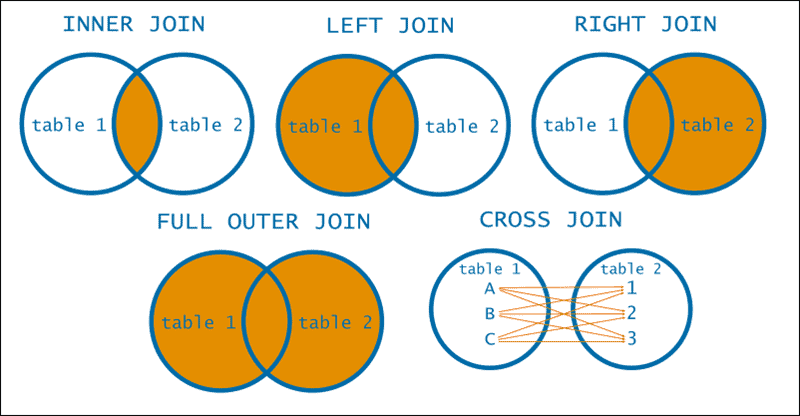
HAVING order\_count > 1000;

**JOINS and Types of Joins**

**What & Why They Came**

* **JOINS** let you combine rows from two or more tables based on related columns—central to the relational model.
* SQL‑86 introduced the explicit JOIN … ON syntax, replacing older comma joins with WHERE clauses.

| **Join Type** | **Description** |
| --- | --- |
| **INNER JOIN** | Returns rows with matching keys in both tables. |
| **LEFT JOIN** | All rows from left table; matching or NULLs on right. |
| **RIGHT JOIN** | All rows from right table; matching or NULLs on left. |
| **FULL JOIN** | (Not supported directly in MySQL) union of left & right joins. |
| **CROSS JOIN** | Cartesian product (all combinations). |
|  |  |



**Rules & Syntax**

* Always specify join condition in ON to avoid accidental Cartesian products (except for explicit CROSS JOIN).
* MySQL supports STRAIGHT\_JOIN to force table order.

**When & Where to Use**

* **INNER JOIN** for related data (e.g. orders → customers).
* **LEFT JOIN** to include parent rows even if no child exists (e.g. products with zero sales).
* **CROSS JOIN** rarely, for combinatorial scenarios.

**Real‑World Examples**

1. **INNER JOIN: Orders with customer names**

SELECT o.order\_id, c.name, o.total\_amount

FROM orders o

INNER JOIN customers c

ON o.customer\_id = c.customer\_id;

1. **LEFT JOIN: All products and any reviews**

SELECT p.product\_id, p.name, r.rating

FROM products p

LEFT JOIN reviews r

ON p.product\_id = r.product\_id;

1. **RIGHT JOIN: All reviews and their products**

SELECT r.review\_id, p.name, r.comment

FROM products p

RIGHT JOIN reviews r

ON p.product\_id = r.product\_id;

1. **CROSS JOIN: All size‑color combinations**

SELECT s.size, c.color

FROM sizes s

CROSS JOIN colors c;

**Putting It All Together: A Complex JDBC Example**

String sql =

"SELECT c.country, COUNT(o.order\_id) AS orders\_count, " +

" SUM(o.total\_amount) AS revenue " +

"FROM customers c " +

"LEFT JOIN orders o ON c.customer\_id = o.customer\_id " +

"WHERE o.order\_date BETWEEN ? AND ? " +

"GROUP BY c.country " +

"HAVING revenue > ? " +

"ORDER BY revenue DESC";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setDate(1, Date.valueOf("2025-01-01"));

ps.setDate(2, Date.valueOf("2025-03-31"));

ps.setBigDecimal(3, new BigDecimal("100000.00"));

try (ResultSet rs = ps.executeQuery()) {

while (rs.next()) {

System.out.printf(

"Country: %s, Orders: %d, Revenue: %.2f%n",

rs.getString("country"),

rs.getInt("orders\_count"),

rs.getBigDecimal("revenue")

);

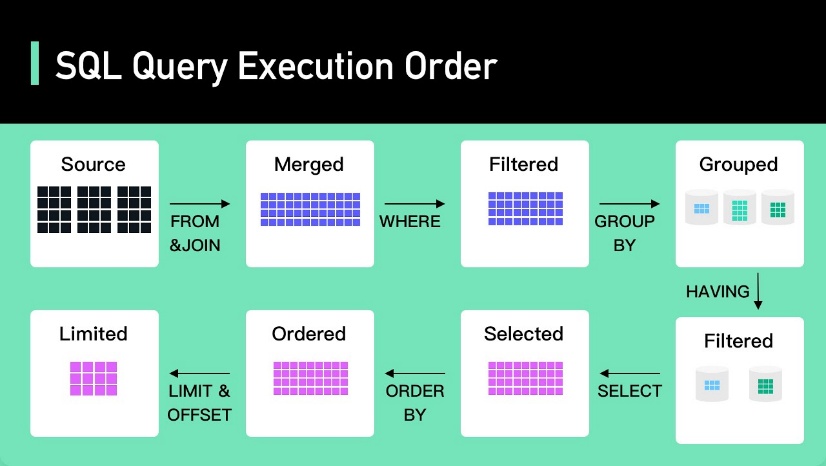
}

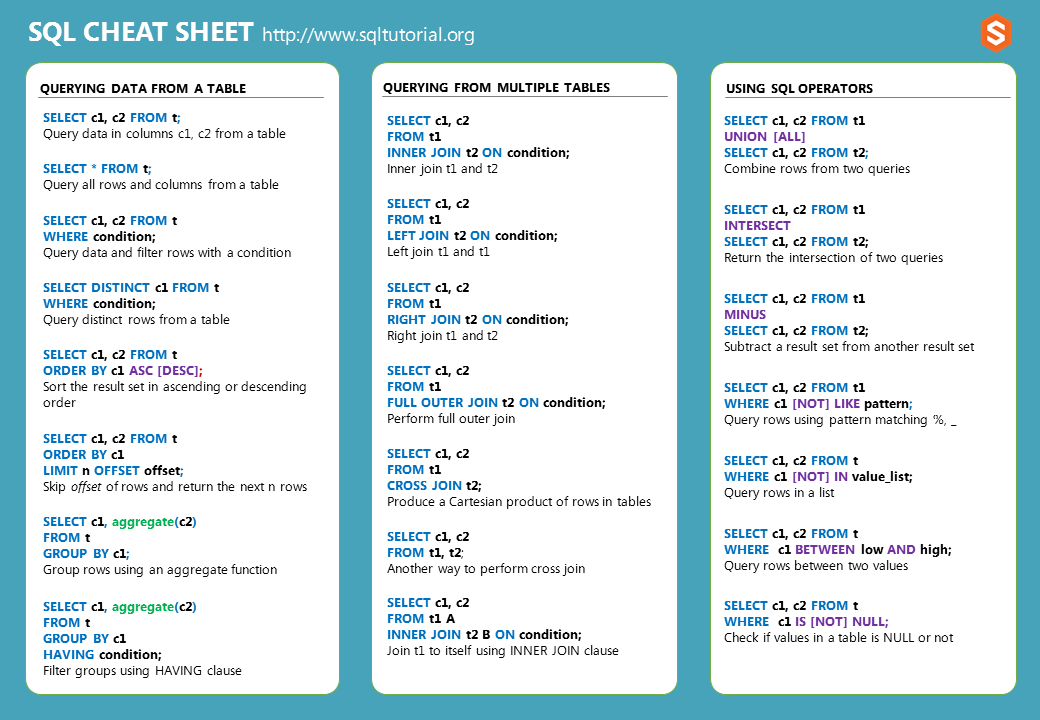
}

}

This snippet demonstrates:

* **BETWEEN** in WHERE
* **LEFT JOIN** between customers and orders
* **GROUP BY** on country
* **HAVING** to filter by aggregated revenue
* **ORDER BY** to sort results





*https://www.sqltutorial.org/sql-cheat-sheet/*

# **Why MongoDB?**

**What & Why It Came**

* **Origins**: Introduced in 2009 by 10gen (now MongoDB, Inc.) to address the needs of modern, agile applications requiring rapid evolution of data models and horizontal scalability.
* **Motivation**: Traditional RDBMSes force a rigid schema up front; every schema change (adding a column, changing a type) can be costly in downtime and migrations. MongoDB’s flexible, document‑oriented model lets you evolve your schema on the fly.

**Core Rules & Properties**

1. **Schema‑less**: Documents in a collection can have different fields, types, and even nested structures.
2. **BSON Storage**: Under the hood, MongoDB stores documents in BSON (binary JSON), supporting rich types (dates, binary data, arrays, embedded documents).
3. **Indexing**: You can create indexes on any field, including nested fields and array values.
4. **Replication & Sharding**: Built‑in replica sets for high availability; automatic sharding for horizontal scaling.

**When & Where to Use**

* **Rapid prototyping**: Startups building MVPs where requirements shift frequently.
* **Hierarchical or nested data**: e.g. user profiles with variable attributes, product catalogs with variant options.
* **High‑write or high‑throughput workloads**: Logging, event streams, IoT data ingestion.
* **Geo‑distributed applications**: Sharded clusters across regions for low‑latency reads.

**Real‑World Examples**

1. **E‑commerce product catalog**
   * Products have variable attributes (size, color, specs). Instead of dozens of nullable columns, each product document carries only relevant fields and an array of variants.
2. **Event logging platform**
   * Each event (click, pageview, transaction) has different metadata. Storing as documents avoids the need for an unwieldy “wide” table and lets you index only the fields you query most.

**2. What Is a Document?**

**What & Why It Came**

* **Definition**: In MongoDB, a **document** is a single record stored in BSON format. It’s the atomic unit of data, analogous to a “row” in a relational table but far richer—able to embed nested sub‑documents and arrays.
* **Motivation**: JSON‑like documents map naturally to objects in application code (e.g., JavaScript objects), reducing impedance mismatch between in‑memory objects and their persisted form.

**Core Rules & Properties**

1. **Key‑Value Pairs**: A document is a set of field–value pairs ({ name: "Alice", age: 30 }).
2. **Nested Structures**: Fields can themselves be documents or arrays ({ address: { city: "Mumbai", zip: "400001" } }).
3. **Size Limit**: Each document is capped at 16 MB, encouraging you to keep individual documents reasonably small.
4. **Immutable \_id**: Every document has a unique \_id field (auto‑generated ObjectId unless you supply your own).

**When & Where to Use**

* **Representing aggregates**: e.g., an order document embedding its line items array.
* **One‑to‑many relationships** that rarely change in size (e.g., a blog post with comments embedded up to a sensible limit).

**Real‑World Examples**

1. **Order with line items**

{

"\_id": ObjectId("…"),

"customer\_id": 12345,

"date": ISODate("2025-04-22T10:30:00Z"),

"items": [

{ "sku": "ABC123", "qty": 2, "price": 199.99 },

{ "sku": "XYZ789", "qty": 1, "price": 349.50 }

],

"status": "shipped"

}

1. **User profile with dynamic attributes**

{

"\_id": ObjectId("…"),

"username": "ravipatel",

"email": "ravi@example.com",

"preferences": {

"newsletter": true,

"theme": "dark"

},

"social": [ "@twitter\_handle", "@github\_handle" ]

}

**3. What Is a Collection?**

**What & Why It Came**

* **Definition**: A **collection** is a grouping of MongoDB documents, akin to a table in RDBMS, but without a fixed schema.
* **Motivation**: Collections provide a logical namespace and indexing scope, while still enabling each document to differ in structure.

**Core Rules & Properties**

1. **Schema‑less**: No enforced schema; you can insert any document shape.
2. **Indexes**: You define indexes at the collection level for any field or sub‑field.
3. **Capped & TTL Collections**:
   * **Capped** collections fix size/number of documents (useful for logs).
   * **TTL** indexes let documents expire automatically.
4. **Namespace**: Collections live within databases; names must be unique per database.

**When & Where to Use**

* **Separate concerns**: e.g., have collections for users, orders, products, logs.
* **High‑throughput logging**: use a capped collection to store only the most recent N entries.
* **Time‑series data**: use TTL collections to auto‑remove old entries.

**Real‑World Examples**

1. **sessions capped collection**

db.createCollection("sessions", { capped: true, size: 10485760 }); // 10 MB

* + Stores only the latest session documents for quick lookups and automatic eviction.

1. **events TTL collection**

db.events.createIndex({ createdAt: 1 }, { expireAfterSeconds: 2592000 }); // 30 days

* + Automatically purges event documents older than 30 days, ideal for audit logs.

**4. MongoDB vs MySQL**

| **Aspect** | **MongoDB** | **MySQL** |
| --- | --- | --- |
| **Data Model** | Schema‑less BSON documents | Fixed schema relational tables |
| **Query Language** | JSON‑style queries (db.collection.find({ … })) | SQL (SELECT … FROM … WHERE …) |
| **Joins** | Limited ($lookup pipeline stage) | Native, optimized JOINs on indexed foreign keys |
| **Transactions** | Multi‑document ACID transactions (since v4.0) | ACID transactions by default (InnoDB engine) |
| **Scaling** | Horizontal sharding built‑in | Primarily vertical; external sharding solutions |
| **Use Cases** | Flexible, nested data; polyglot apps; high‑ingest rate | Structured, relational data; complex reporting |

**When & Where to Choose**

* **Choose MongoDB if**
  + Your data won’t fit neatly into tables or needs frequent schema changes.
  + You need to store and query rich, nested JSON documents.
  + You require out‑of‑the‑box horizontal scaling and geo‑distribution.
* **Choose MySQL if**
  + You have well‑defined schemas with strong relational integrity.
  + You need complex transactions, joins, and mature tooling for reporting.
  + Your workload is OLTP‑heavy with strict consistency guarantees.

**Real‑World Comparison Examples**

1. **Content Management System (CMS)**
   * **MongoDB**: Each article can have a variable set of metadata (tags, embedded media), and new fields can be added on the fly without migrations.
   * **MySQL**: Requires ALTER TABLE to add new metadata columns, and you may need join tables for tags and media, adding complexity.
2. **Financial Ledger**
   * **MySQL**: Preferred for its strict ACID compliance, row‑level locking, and complex joins across accounts, transactions, and audit tables.
   * **MongoDB**: Possible with transactions but less mature for cross‑collection reporting; better suited for storing JWT‑style audit events alongside ledger entries if you need a hybrid approach.