Cheatography

MySQL Concepts Cheat Sheet

by Arshdeep via cheatography.com/201979/cs/42813/

Operators in MySQL

Comparison Operators	Logical Operators	Arithmetic Operators
=: Equal to	AND: Returns true if all conditions separated by AND are true	+: Addition
<> or !=: Not equal to	OR: Returns true if any condition separated by OR is true	-: Subtraction
<: Less than	NOT: Reverses the value of the following condition	*: Multiplication
>: Greater than	NULL Operators	/: Division
<=: Less than or equal to	IS NULL: Checks if a value is NULL	%: Modulus (Returns the remainder of a division)
>=: Greater than or equal to	IS NOT NULL: Checks if a value is not NULL	LIKE: Used for pattern matching in strings

String Functions

Function	Explanation	Example
CONCAT()	Concatenates two or more strings.	SELECT CONCAT('Hello ', 'World') AS ConcatenatedString; -- Output: Hello World
SUBSTRING()	Extracts a substring from a string.	SELECT SUBSTRING('MySQL', 2, 3) AS SubstringResult; -- Output: ySQ
UPPER()	Converts a string to uppercase.	SELECT UPPER('mysql') AS UppercaseString; -- Output: MYSQL
LOWER()	Converts a string to lowercase.	SELECT LOWER('MYSQL') AS LowercaseString; -- Output: mysql
LENGTH()	Returns the length of a string.	SELECT LENGTH('MySQL') AS StringLength; -- Output: 5

String Functions (cont)

TRIM()	Removes leading and trailing spaces from a string.	SELECT TRIM(' MySQL ') AS TrimmedString; -- Output: MySQL
REPLACE()	Replaces occurrences of a specified substring within a string.	SELECT REPLACE('Hello World', 'World', 'MySQL') AS ReplacedString; -- Output: Hello MySQL

Date and Time Functions

Function	Explanation	Example
NOW()	Returns the current date and time.	SELECT NOW() AS CurrentDateTime; -- Output: Current date and time in 'YYYY-MM-DD HH:MM:SS' format
CURDATE()	Returns the current date.	SELECT CURDATE() AS CurrentDate; -- Output: Current date in 'YYYY-MM-DD' format
CURTIME()	Returns the current time.	SELECT CURTIME() AS CurrentTime; -- Output: Current time in 'HH:MM:SS' format
YEAR()	Extracts the year from a date.	SELECT YEAR('2024-03-23') AS ExtractedYear; -- Output: 2024
MONTH()	Extracts the month from a date.	SELECT MONTH('2024-03-23') AS ExtractedMonth; -- Output: 3
DAY()	Extracts the day from a date.	SELECT DAY('2024-03-23') AS ExtractedDay; -- Output: 23

Window Functions

Function	Explanation	Example

Window Functions (cont)

ROW_NUMBER()	This function assigns a unique integer to each row within a partition according to the specified order. It starts from 1 for the first row and increments by 1 for each subsequent row.	SELECT name, ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num FROM employees;
RANK()	Similar to ROW_NUMBER(), but RANK() assigns the same rank to rows with equal values and leaves gaps in the sequence for ties.	SELECT name, RANK() OVER (ORDER BY score DESC) AS rank FROM students;
DENSE_RANK()	DENSE_RANK() is similar to RANK(), but it does not leave gaps in the ranking sequence for ties.	SELECT name, DENSE_RANK() OVER (ORDER BY age) AS dense_rank FROM users;
NTILE()	This function divides the result set into a specified number of buckets and assigns a bucket number to each row. It ensures an approximately equal number of rows in each bucket.	SELECT name, salary, NTILE(4) OVER (ORDER BY salary) AS quartile FROM employees;
LEAD() and LAG()	LEAD() and LAG() functions allow you to access data from a subsequent or previous row in the result set, respectively.	SELECT name, salary, LEAD(salary) OVER (ORDER BY salary) AS next_salary, LAG(salary) OVER (ORDER BY salary) AS previous_salary FROM employees;

Joins

Join	Explanation	Syntax	Example
INNER JOIN	Returns records that have matching values in both tables.	SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;	SELECT orders.order_id, customers.customer_name FROM orders INNER JOIN customers ON orders.customer_id = customers.customer_id;
LEFT JOIN (or LEFT OUTER JOIN)	Returns all records from the left table and the matched records from the right table. If there's no match, the result is NULL on the right side.	SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;	SELECT customers.customer_name, orders.order_id FROM customers LEFT JOIN orders ON customers.customer_id = orders.customer_id;
RIGHT JOIN (or RIGHT OUTER JOIN)	Returns all records from the right table and the matched records from the left table. If there's no match, the result is NULL on the left side.	SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;	SELECT orders.order_id, customers.customer_name FROM orders RIGHT JOIN customers ON orders.customer_id = customers.customer_id;

Joins (cont)

FULL JOIN (or FULL OUTER JOIN)	Returns all records when there's a match in either left or right table. If there's no match, the result is NULL on the unmatched side.	SELECT columns FROM table1 FULL JOIN table2 ON table1.column = table2.column;	SELECT customers.customer_name, orders.order_id FROM customers FULL JOIN orders ON customers.customer_id = orders.customer_id;
CROSS JOIN	Returns the Cartesian product of the two tables, i.e., all possible combinations of rows.	SELECT columns FROM table1 CROSS JOIN table2;	SELECT * FROM employees CROSS JOIN departments;
Self-Join	Joins a table with itself, typically used to compare rows within the same table.	SELECT columns FROM table1 alias1 INNER JOIN table1 alias2 ON alias1.column = alias2.column;	SELECT e1.employee_name, e2.manager_name FROM employees e1 INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;

Stored Procedure

Definition	A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. It's like a function in a traditional programming language.
Syntax	CREATE PROCEDURE procedure_name (parameters) BEGIN -- SQL statements END;
Parameters	Stored procedures can accept input parameters, which can be used within the procedure's SQL statements.

Stored Procedure (cont)

Example	CREATE PROCEDURE GetEmployee(IN emp_id INT) BEGIN SELECT * FROM employees WHERE employee_id = emp_id; END;
Calling a Stored Procedure	CALL procedure_name(arguments);
Example	CALL GetEmployee(1001);
Dropping a Stored Procedure	DROP PROCEDURE IF EXISTS procedure_name;
Example	DROP PROCEDURE IF EXISTS GetEmployee;
Variables	Stored procedures can declare and use variables within their code.
Example	CREATE PROCEDURE UpdateSalary(IN emp_id INT, IN salary DECIMAL(10, 2)) BEGIN DECLARE emp_name VARCHAR(50); SELECT employee_name INTO emp_name FROM employees WHERE employee_id = emp_id; UPDATE employees SET employee_salary = salary WHERE employee_id = emp_id; END;
Control Flow	Stored procedures support control flow constructs such as IF, CASE, and LOOP.
Example	CREATE PROCEDURE CheckAge(IN age INT) BEGIN IF age < 18 THEN SELECT 'Minor'; ELSEIF age BETWEEN 18 AND 64 THEN SELECT 'Adult'; ELSE SELECT 'Senior'; END IF; END;
Cursors	Stored procedures can use cursors to process multiple rows returned by a query.

Stored Procedure (cont)

Example	CREATE PROCEDURE DisplayEmployees() BEGIN DECLARE done BOOLEAN DEFAULT FALSE; DECLARE emp_name VARCHAR(50); DECLARE emp_salary DECIMAL(10, 2); DECLARE emp_cursor CURSOR FOR SELECT employee_name, employee_-salary FROM employees; DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE; OPEN emp_cursor; read_loop: LOOP FETCH emp_cursor INTO emp_name, emp_salary; IF done THEN LEAVE read_loop; END IF; -- Process fetched data END LOOP; CLOSE emp_cursor; END;

Indexing

Indexing	Indexing is a way to optimize database performance by quickly locating rows in a table. It allows for faster retrieval of data by creating a sorted reference to the data in a table.
Types	Single Column Index, Composite Index, Unique Index, Primary Key, and Foreign Key
Single Column Index	Index created on a single column.
Composite Index	Index created on multiple columns.
Unique Index	Index where all values must be unique (no duplicate values).
Primary Key	Unique index with the constraint that all values must be unique and not NULL. Typically used to uniquely identify each row in a table.
Foreign Key	Index that references the primary key in another table. Used to establish relationships between tables.

Creating Indexes

Indexing (cont)

Syntax	CREATE [UNIQUE] INDEX index_name ON table_name (column_name);
Example	CREATE INDEX idx_lastname ON employees (last_-name);

Dropping Indexes:

Syntax	DROP INDEX index_name ON table_name;
Example	DROP INDEX idx_lastname ON employees;

Viewing Indexes:

Syntax	SHOW INDEX FROM table_name;
Example	SHOW INDEX FROM employees;

Types of SQL Functions

Scalar Functions: Scalar functions operate on individual rows and return a single result per row. They can be used in SELECT, WHERE, ORDER BY, and other clauses.

Aggregate Functions: Aggregate functions operate on sets of rows and return a single result that summarizes the entire set. They are commonly used with the GROUP BY clause.

Window Functions: Window functions perform calculations across a set of rows related to the current row, without collapsing the result set into a single row. They are used with the OVER() clause.

Control Flow Functions: Control flow functions allow conditional execution of logic within SQL statements. They are often used to implement branching or conditional behavior.

User-Defined Functions (UDFs): User-defined functions are custom functions created by users to perform specific tasks that are not provided by built-in functions. They can be written in languages like SQL, C, or C++ and loaded into MySQL.

Numeric Functions

Function	Explanation	Example
ABS()	Returns the absolute value of a number.	SELECT ABS(-10) AS AbsoluteValue; -- Output: 10
ROUND()	Rounds a number to a specified number of decimal places.	SELECT ROUND(3.14159, 2) AS RoundedNumber; -- Output: 3.14
CEIL()	Returns the smallest integer greater than or equal to a number.	SELECT CEIL(3.2) AS CeilingValue; -- Output: 4

Numeric Functions (cont)

FLOOR()	Returns the largest integer less than or equal to a number.	SELECT FLOOR(3.8) AS FloorValue; -- Output: 3
MOD()	Returns the remainder of a division operation.	SELECT MOD(10, 3) AS ModulusValue; -- Output: 1

Aggregate Functions

Function	Explanation	Example
COUNT()	The COUNT() function returns the number of rows that match a specified condition.	SELECT COUNT(*) AS total_customers FROM customers;
SUM()	The SUM() function calculates the sum of values in a column.	SELECT SUM(quantity) AS total_quantity FROM orders;
AVG()	The AVG() function calculates the average of values in a column.	SELECT AVG(price) AS average_price FROM products;
MAX()	The MAX() function returns the maximum value in a column.	SELECT MAX(salary) AS max_salary FROM employees;
MIN()	The MIN() function returns the minimum value in a column.	SELECT MIN(age) AS min_age FROM users;
GROUP_CONCAT()	The GROUP_CONCAT() function concatenates the values of a column into a single string.	SELECT GROUP_CONCAT(product_name) AS product_list FROM products;
STD()	The STD() function calculates the standard deviation of values in a column.	SELECT STD(sales) AS sales_std_deviation FROM monthly_sales;
VARIANCE()	The VARIANCE() function calculates the variance of values in a column.	SELECT VARIANCE(height) AS height_variance FROM students;

Control Flow Functions

Function	Explanation	Syntax	Example
CASE Statement	The CASE statement evaluates a list of conditions and returns one of multiple possible result expressions. It's similar to a switch or if-else statement in other programming languages.	CASE WHEN condition1 THEN result1 WHEN condition2 THEN result2 ... ELSE default_result END	SELECT CASE WHEN age < 18 THEN 'Minor' WHEN age BETWEEN 18 AND 64 THEN 'Adult' ELSE 'Senior' END AS age_group FROM persons;
IF() Function	The IF() function returns one value if a condition is TRUE and another value if the condition is FALSE.	IF(condition, value_if_true, value_if_false)	SELECT IF(score >= 60, 'Pass', 'Fail') AS result FROM students;
COALESCE() Function	The COALESCE() function returns the first non-NULL value in a list of expressions.	COALESCE(value1, value2, ...)	SELECT COALESCE(first_name, 'Anonymous') AS display_name FROM users;
NULLIF() Function	The NULLIF() function returns NULL if the two specified expressions are equal; otherwise, it returns the first expression.	NULLIF(expression1, expression2)	SELECT NULLIF(dividend, 0) AS result FROM calculations;

Subqueries

Subquery	Example
A subquery, also known as a nested query or inner query, is a query nested within another SQL statement. It allows you to use the result of one query as a part of another query.	SELECT column_name FROM table_name WHERE column_name OPERATOR (SELECT column_name FROM table_name WHERE condition);
Single-Row Subquery: Returns only one row of results.	SELECT name FROM employees WHERE employee_id = (SELECT manager_id FROM departments WHERE department_id = 100);
Multiple-Row Subquery: Returns multiple rows of results.	SELECT product_name FROM products WHERE category_id IN (SELECT category_id FROM categories WHERE category_name = 'Electronics');
Inline View Subquery: Creates a temporary table within a query.	SELECT * FROM (SELECT employee_id, first_name, last_name FROM employees) AS emp_info WHERE emp_info.employee_id > 100;
Correlated Subquery: References one or more columns in the outer query.	SELECT product_name FROM products p WHERE p.unit_price > (SELECT AVG(unit_price) FROM products WHERE category_id = p.category_id);

Common Table Expressions (CTE)

Explanation	Common Table Expressions (CTEs) provide a way to define temporary result sets that can be referenced within a single SELECT, INSERT, UPDATE, or DELETE statement. They enhance the readability and maintainability of complex queries.
Syntax	WITH cte_name (column1, column2, ...) AS (-- CTE query SELECT ... FROM ... WHERE ...) -- Main query using the CTE SELECT ... FROM cte_name;

Common Table Expressions (CTE) (cont)

Example	-- Define a CTE to get the top 5 customers with the highest total orders WITH top_customers AS (SELECT customer_id, SUM(order_total) AS total_spent FROM orders GROUP BY customer_id ORDER BY total_spent DESC LIMIT 5) -- Use the CTE to get detailed information about the top customers SELECT c.customer_id, c.customer_name, tc.total_spent FROM customers c JOIN top_customers tc ON c.customer_id = tc.customer_id;

Views

Explanation	Views in MySQL are virtual tables created by executing a SELECT query and are stored in the database. They allow users to simplify complex queries, restrict access to certain columns, and provide a layer of abstraction over the underlying tables.
Syntax to Create Views	CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;
Example to Create Views	CREATE VIEW customer_contacts AS SELECT customer_id, first_name, last_name, email FROM customers WHERE subscription_status = 'active';
Syntax to Drop Views	DROP VIEW view_name;
Example to Drop Views	DROP VIEW customer_contacts;
Syntax to Update View	CREATE OR REPLACE VIEW view_name AS SELECT new_column1, new_column2, ... FROM new_table WHERE new_condition;
Example to Update View	CREATE OR REPLACE VIEW active_customers AS SELECT customer_id, first_name, last_name, email FROM customers WHERE subscription_status = 'active';

Views (cont)

Syntax to Retrieve Data	SELECT * FROM view_name;
Example to Retrieve Data	SELECT * FROM customer_contacts;

Trigger

Introduction	A trigger is a database object that automatically performs an action in response to certain events on a particular table.
Syntax	CREATE TRIGGER trigger_name {BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON table_name FOR EACH ROW trigger_body
trigger_name	Name of the trigger.
BEFORE | AFTER	Specifies when the trigger should be fired, before or after the event.
INSERT | UPDATE | DELETE	Specifies the event that triggers the action.
table_name	Name of the table on which the trigger operates.
FOR EACH ROW	Indicates that the trigger will be fired for each row affected by the triggering event.
trigger_body	Actions to be performed when the trigger is fired.
Example	CREATE TRIGGER audit_trigger AFTER INSERT ON employees FOR EACH ROW BEGIN INSERT INTO audit_log (event_type, event_time, user_id) VALUES ('INSERT', NOW(), NEW.id); END;
BEFORE Triggers	Fired before the triggering action occurs. Can be used to modify data before it is inserted, updated, or deleted.
AFTER Triggers	Fired after the triggering action occurs. Can be used for logging, auditing, or other post-action tasks.

Trigger (cont)

Accessing Data	Use NEW.column_name to access the new value of a column in an INSERT or UPDATE trigger. Use OLD.column_name to access the old value of a column in an UPDATE or DELETE trigger.
Dropping a Trigger	DROP TRIGGER [IF EXISTS] trigger_name;

Performance Optimization

Indexing:	
Use Indexes	Indexes help in speeding up the data retrieval process by creating efficient lookup paths.
Choose the Right Columns	Identify columns frequently used in WHERE, JOIN, and ORDER BY clauses for indexing.
Avoid Overindexing	Unnecessary indexes can slow down write operations and consume disk space.
Regularly Analyze and Optimize Indexes	Monitor index usage and performance regularly. Use tools like EXPLAIN to analyze query execution plans.
Query Optimization:	
Optimize Queries	Write efficient queries by avoiding unnecessary joins, using appropriate WHERE clauses, and minimizing data retrieval.
Use LIMIT	When fetching a large dataset, limit the number of rows returned to reduce the workload on the server.
Avoid SELECT	Explicitly specify only the required columns in SELECT statements to reduce data transfer overhead.