|  |  |  |
| --- | --- | --- |
| Commands | Syntax | Applications |
| Important SQL Clauses (Data Manipulation Language) | | |
| SELECT statement |  |  |
| * The SELECT statement is used to retrieve data from one or more tables in a database. * It is one of the most fundamental and widely used SQL commands, allowing users to specify the columns and conditions for extracting relevant data. | SELECT column1, column2, ...  FROM table\_name  WHERE condition (optional)  GROUP BY column (optional)  HAVING condition (optional)  ORDER BY column ASC|DESC (optional)  LIMIT number (optional); | SELECT firstname, lastname, age  FROM students  WHERE age > 18  ORDER BY lastname ASC  LIMIT 5; |
| Select all data from the table | SECECT \* FROM table\_name |  |
| * The SELECT DISTINCT statement is used to retrieve unique values from one or more columns in a table, eliminating duplicate records from the result set. | SELECT DISTINCT column1, column2, ...  FROM table\_name  WHERE condition (optional); | SELECT DISTINCT firstname,  FROM students; |
|  |  |  |
| FROM clause |  |  |
| * The FROM clause can reference one or more tables, or even subqueries. | SELECTcolumn1, column2, ...  FROM table\_name; | SELECT firstname, lastname  FROM students; |
| * When retrieving data from multiple tables, the FROM clause can be extended with JOIN operations to specify how the tables are related.   Tables are: orders, customers | SELECT orders.order\_id, customers.customer\_name  FROM orders  JOIN customers ON orders.customer\_id = customers.customer\_id; | * In this query, data is retrieved from both the orders and customers tables. The **JOIN** keyword specifies how the two tables are related by using the customer\_id column. |
| * The FROM clause can also include subqueries (a query within another query) | SELECT avg\_salary  FROM (SELECT department, AVG(salary) AS avg\_salary  FROM employees  GROUP BY department) AS department\_avg; | * Here, the subquery computes the average salary for each department, and the outer query selects the avg\_salary from the result of that subquery. |
|  |  |  |
| WHERE clause |  |  |
| * The WHERE clause filters the rows returned by the query, only returning those that satisfy the conditions. * It is typically used with the SELECT, UPDATE, DELETE, and INSERT INTO statements to narrow down the data based on certain criteria. | SELECT column1, column2, ...  FROM table\_name  WHERE condition; | * It can involve comparison operators (=, >, <, >=, <=, <>), * logical operators (AND, OR, NOT), * or special operators like IN, BETWEEN, LIKE, etc. |
| * This query retrieves the firstname and lastname of all students whose age is greater than 18 | SELECT firstname, lastname  FROM students  WHERE age > 18; | * This query retrieves the firstname and lastname of all students whose **age** is greater than 18. |
| * Using AND and OR | SELECT firstname, lastname  FROM students  WHERE age > 18 AND class = 'Math'; | * This query retrieves the names of students who are older than 18 **and** belong to the Math class. |
| * Using LIKE for Pattern Matching | SELECT firstname, lastname  FROM students  WHERE lastname LIKE 'Smi%'; | * This query retrieves the names of students whose last name starts with "Smi" (e.g., Smith). |
| * Using IN for Multiple Conditions | SELECT firstname, lastname  FROM students  WHERE class IN ('Math', 'Physics'); | * This query retrieves the names of students who are in either the Math or Physics classes. |
| * Using BETWEEN for Range Filtering | SELECT firstname, age  FROM students  WHERE age BETWEEN 18 AND 22; | This query retrieves the names and ages of students whose age is between 18 and 22. |
| * Using IS NULL to Filter Null Values | SELECT firstname, lastname  FROM students  WHERE age IS NULL; | * This query retrieves the names of students whose age is not recorded (i.e., the age column is NULL). |
|  |  |  |
| Conditions and Operators (Data Manipulation Language) | | |
|  |  |  |
| SQL TRUE/FALSE Condition |  |  |
| * Comparison operators: Conditions like =, >, <, >=, <=, <> (not equal) can be used to compare values. * The condition is true if the comparison is satisfied. | SELECT firstname, lastname  FROM students  WHERE age > 18; | * A condition is **true** if the data satisfies the logic expressed in the WHERE clause. * Conditions can be based on comparison, logical operations, pattern matching, or range checking. * If no WHERE clause is used, all rows are returned (considered as a **true** condition for every row). |
| * Logical operators: Conditions using logical operators such as AND, OR, and NOT can also be used. | SELECT firstname, lastname  FROM students  WHERE age > 18 AND class = 'Math'; | * Is **true** when both conditions (age greater than 18 and class being "Math") are met. |
| * Pattern matching: Using LIKE with patterns to match specific values. | SELECT firstname, lastname  FROM students  WHERE lastname LIKE 'Smi%'; | * Is **true** for rows where the last name starts with "Smi". |
| * Set membership: Using IN to check if a value is within a set of possible values. | SELECT firstname, lastname  FROM students  WHERE class IN ('Math', 'Physics'); | * Is **true** for rows where the class is either "Math" or "Physics". |
| * Range checking: Using BETWEEN to specify a range. | SELECT firstname, age  FROM students  WHERE age BETWEEN 18 AND 22; | * Is **true** for rows where the age is between 18 and 22. |
| * NULL checks: Using IS NULL or IS NOT NULL to check for NULL values. | SELECT firstname, lastname  FROM students  WHERE age IS NULL; | * Is **true** for rows where the age is **NULL.** |
| * True Condition Without Filtering | SELECT \* FROM students; | * If you don't provide any specific condition (or if you use a condition that is always **true**) |
| * The NOT and NOT IN operators in SQL both negate conditions. * They are used in different contexts and have different purposes. | SELECT firstname, lastname  FROM students  WHERE NOT age > 18; | * The NOT operator is used to negate a **single condition.** |
|  | SELECT firstname, lastname  FROM students  WHERE class NOT IN ('Math', 'Physics', 'History'); | * The **NOT IN** operator is used to **exclude values from a set**. * It is used to check if a value is **not present** in a list of values or subquery. |
| NOT IN with a Subquery. | SELECT firstname, lastname  FROM students  WHERE id NOT IN (SELECT student\_id FROM honors\_students); | This query returns students whose id is **not** present in the honors\_students table. |
| NOT IN with NULL | SELECT firstname, lastname  FROM students  WHERE class NOT IN ('Math', 'Physics', NULL); | * This query will return **no results** because the NULL in the list makes the entire comparison indeterminate. |
| * The EXISTS condition is used to check whether a subquery returns any rows. * It returns TRUE if the subquery finds at least one row, * and FALSE if the subquery returns no rows. * The EXISTS condition is typically used in combination with a subquery in the WHERE clause to test for the existence of records in a related table. | SELECT column1, column2, ...  FROM table\_name  WHERE EXISTS (subquery); | * **subquery**: This is a SELECT query that is usually related to the main query. * The **EXISTS** condition tests whether this subquery returns any rows. * The **EXISTS** condition does not return data from the subquery; it simply checks for the existence of rows and returns **TRUE** or **FALSE**. |
| Simple EXISTS Query   * Two tables: students, enrollments. * You want to select all students who are enrolled in any class. * The EXISTS condition can be used to check if a student has at least one matching row in the enrollments table. | SELECT firstname, lastname  FROM students s  WHERE EXISTS (  SELECT \*  FROM enrollments e  WHERE s.id = e.student\_id  ); | * This query selects the firstname and lastname of students where there is at least one row in the enrollments table where the student\_id matches the id from the students table. * The EXISTS condition checks if the subquery (the SELECT inside) returns at least one row. |
| Using NOT EXISTS   * It is used to return rows where the subquery returns no rows. | SELECT firstname, lastname  FROM students s  WHERE NOT EXISTS (  SELECT \*  FROM enrollments e  WHERE s.id = e.student\_id  ); | * This query selects students who are **not** enrolled in any class. * The **NOT EXISTS** condition is used to exclude students who have matching rows in the enrollments table. |
| Example with Multiple Conditions | SELECT firstname, lastname  FROM students s  WHERE EXISTS (  SELECT \*  FROM enrollments e  WHERE s.id = e.student\_id  AND e.course\_id = 101  ); | * This query returns students who are enrolled in course 101 by checking if any rows exist in the enrollments table where both the student\_id and the course\_id match. |
|  |  |  |
| Concatenate operator |  |  |
| * The concatenate operator is used to combine two or more strings into one. * The specific syntax for concatenation depends on the database system you're using, but the most common methods are:   || (Double Pipe Operator)  CONCAT() Function  + Operator | | |
| SELECT firstname || ' ' || lastname AS full\_name  FROM employees; | SELECT CONCAT(firstname, ' ', lastname) AS full\_name  FROM employees; | SELECT firstname + ' ' + lastname AS full\_name  FROM employees; |
| Handling NULL Values:   * Concatenating a NULL value with a string using || or + often results in NULL. * However, the CONCAT() function can handle NULL values and treat them as empty strings. | SELECT CONCAT(firstname, ' ', lastname) AS full\_name  FROM employees; | * If firstname or lastname contains a NULL value, CONCAT() will skip over the NULL and continue with the non-NULL values. |
| Using COALESCE() with Concatenation   * To handle NULL values explicitly with || or +, you can use COALESCE(), which replaces NULL with an empty string. | SELECT COALESCE(firstname, ' ') || ' ' || COALESCE(lastname, ' ') AS full\_name  FROM employees; |  |
|  | SELECT CONCAT(city, ', ', state, ' ', zip) AS full\_address  FROM addresses; |  |
|  |  |  |
| Temporal Operator |  |  |
| * Temporal operators in SQL are used to work with temporal data—that is, data related to time. * These operators allow you to perform operations on date and time values, making it easier to manage historical, current, and future data. * Temporal operators can be used to filter, compare, and manipulate date and time values in your queries. | | |
| Comparison Operators: Used to compare date and time values.   * <, >, =, <=, >=, <>: Standard comparison operators that can be applied to temporal data. | SELECT \*  FROM events  WHERE event\_date >= '2024-01-01'; | * This retrieves all events occurring on or after January 1, 2024. |
| BETWEEN Operator: Used to filter records within a specified date range. | SELECT \*  FROM orders  WHERE order\_date BETWEEN '2024-01-01' AND '2024-12-31'; | * This selects orders placed during the entire year of 2024. |
| Date Functions: SQL provides built-in functions to manipulate date and time values. |  | **NOW()**: Returns the current date and time.  **CURRENT\_DATE**: Returns the current date.  **DATE\_ADD()**: Adds a time interval to a date.  **DATE\_SUB()**: Subtracts a time interval from a date. |
|  | SELECT DATE\_ADD(order\_date, INTERVAL 1 MONTH) AS next\_order\_date  FROM orders; | * This calculates the date one month after each order\_date. |
| Date Part Functions: Extract specific parts of a date. |  | * **YEAR(), MONTH(), DAY(), HOUR(), MINUTE(), SECOND**(): Functions to retrieve respective components. |
|  | SELECT YEAR(event\_date) AS event\_year  FROM events; | * This retrieves the year from each event\_date. |
| DATEDIFF(): Returns the difference between two date values. | SELECT DATEDIFF('2024-12-31', '2024-01-01') AS days\_difference; | This calculates the number of days between January 1, 2024, and December 31, 2024. |
| TIMESTAMPDIFF(): Calculates the difference between two date or time values in specified units. | SELECT TIMESTAMPDIFF(DAY, '2024-01-01', '2024-12-31') AS days\_difference; | This also calculates the number of days between the two dates. |
| * Examples |  |  |
| Filter records based on a specific date: | SELECT \*  FROM employee\_attendance  WHERE attendance\_date = '2024-05-15'; |  |
| Find records within a date range: | SELECT \*  FROM meetings  WHERE meeting\_date BETWEEN '2024-01-01' AND '2024-06-30'; |  |
| Calculate the difference in days between two dates: | SELECT employee\_id, DATEDIFF(leave\_end\_date, leave\_start\_date) AS leave\_duration  FROM employee\_leaves; |  |
| Get the current date: | SELECT NOW() AS current\_datetime; | SELECT CURRENT\_DATE + INERVAL 7 DAY AS Next\_Week; |
|  |  |  |
| Data grouping, sorting, select and dates (DML) | | |
|  |  |  |
| GROUP BY clause |  |  |
| * The GROUP BY clause in SQL is used to arrange identical data into groups. * This is particularly useful when you want to aggregate data and perform calculations on it, such as counting, summing, The GROUP BY clause typically works in conjunction with aggregate functions like COUNT(), SUM(), AVG(), MIN(), and MAX(). | | |
|  | SELECT column1, aggregate\_function(column2)  FROM table\_name  WHERE condition (optional)  GROUP BY column1, column2, ...; | * **Aggregate Functions**: Functions like COUNT(), SUM(), etc., that perform calculations on grouped data. * **GROUP BY Clause**: Specifies the columns that the results should be grouped by. |
| Counting Employees in Each Department | SELECT department, COUNT(\*) AS employee\_count  FROM employees  GROUP BY department; | * This query counts how many employees belong to each department. |
| Total Sales per Product | SELECT product\_id, SUM(sales\_amount) AS total\_sales  FROM sales  GROUP BY product\_id; | * This sums the sales amounts for each product, returning the total sales per product. |
| Average Salary by Job Title | SELECT job\_title, AVG(salary) AS average\_salary  FROM employees  GROUP BY job\_title; | * This calculates the average salary for each job title in the employees table. |
| Grouping by Multiple Columns | SELECT country, city, COUNT(\*) AS customer\_count  FROM customers  GROUP BY country, city; | * This counts the number of customers in each city within each country. |
| HAVING clause | **Using HAVING with GROUP BY** | SELECT column1, aggregate\_function(column2)  FROM table\_name  GROUP BY column1  HAVING condition; |
| * The HAVING clause filters records after grouping. * Works with Aggregate Functions: It is generally used to apply conditions on the results of aggregate functions. | | |
| Filtering Groups Based on Aggregate Functions | SELECT department, COUNT(\*) AS employee\_count  FROM employees  GROUP BY department  HAVING COUNT(\*) > 5; | * This query retrieves departments that have more than 10 employees. * The filtering is done on the grouped data after applying the COUNT() function. |
| Using HAVING with SUM() | SELECT product\_id, SUM(sales\_amount) AS total\_sales  FROM sales  GROUP BY product\_id  HAVING SUM(sales\_amount) > 10000; | * This query returns only those products that have total sales greater than 10,000. |
| Combining WHERE and HAVING | SELECT product\_id, SUM(sales\_amount) AS total\_sales  FROM sales  WHERE sale\_date >= '2024-01-01'  GROUP BY product\_id  HAVING SUM(sales\_amount) > 5000; | * The WHERE clause filters rows with sales from 2024 onwards, and then the HAVING clause ensures that only products with total sales exceeding 5,000 are returned. |
| Filtering by Average | SELECT department, AVG(salary) AS average\_salary  FROM employees  GROUP BY department  HAVING AVG(salary) > 50000; | * This query shows departments where the average salary is greater than 50,000. |
|  |  |  |
| Key Differences Between WHERE and HAVING | * **WHERE**: Filters rows before any aggregation occurs. * **HAVING**: Filters groups after aggregation has occurred. |  |
|  |  |  |
| ORDER BY clause | SELECT column1, column2, ...  FROM table\_name  WHERE condition  ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...; |  |
| * The ORDER BY clause is used to sort query results in ascending or descending order. * It allows sorting by one or more columns. * Sorting can be done on both text and numeric columns, and the default sorting order is ascending (ASC). * It is frequently used in reports and queries where the presentation order of the data matters. | | |
| Sorting by One Column (Ascending Order) | SELECT firstname, lastname, age  FROM students  ORDER BY age; | * This query selects the first name, last name, and age of all students and sorts the results by age in ascending order. |
| Sorting by One Column (Descending Order) | SELECT firstname, lastname, age  FROM students  ORDER BY age DESC; | * This query returns the same data but sorts the students by age in descending order. |
| Sorting by Multiple Columns | SELECT firstname, lastname, age  FROM students  ORDER BY lastname ASC, firstname ASC; | * This query sorts the result first by the lastname column in ascending order, * and if there are students with the same last name, it sorts them by firstname in ascending order. |
| Sorting with WHERE Clause | SELECT firstname, lastname, grade  FROM students  WHERE grade >= 75  ORDER BY grade DESC; | * This query selects students who scored 75 or higher and sorts them by their grades in descending order. |
| Combining ORDER BY with GROUP BY | SELECT department, COUNT(\*) AS employee\_count  FROM employees  GROUP BY department  ORDER BY employee\_count DESC; | * This query groups employees by department, counts the number of employees in each department, and then sorts the departments by the number of employees in descending order. |
|  |  |  |
| LIMIT or TOP clause | SELECT column1, column2, ...  FROM table\_name  LIMIT number\_of\_rows OFFSET offset\_value; | * **number\_of\_rows:** Specifies the maximum number of rows to return. * **OFFSET (optional):** Skips a specified number of rows before starting to return rows. |
| * LIMIT is used in MySQL, PostgreSQL, and SQLite. * TOP is used in SQL Server. * OFFSET can be used with LIMIT to skip a specific number of rows. * whereas SQL Server uses OFFSET-FETCH for similar functionality. | | |
| Limiting results to 5 rows | SELECT \*  FROM employees  LIMIT 5; | * This query will return only **the first 5 rows** from the employees table. |
| Using LIMIT with OFFSET | SELECT \*  FROM employees  LIMIT 5 OFFSET 10; | * This query **skips the first 10 rows** and then **returns the next 5 rows**. |
|  |  |  |
| TOP Clause (SQL Server) |  |  |
| Limiting to top 3 rows in SQL Server | SELECT TOP 3 \*  FROM employees; | * This query will **return the first 3 rows** from the employees table. |
| Using TOP with ORDER BY (SQL Server) | SELECT TOP 5 \*  FROM employees  ORDER BY salary DESC; | * This query returns the **top 5 highest-paid employees**. |
|  |  |  |
| SQL wildcards |  |  |
| * SQL wildcards are used with the LIKE operator in WHERE clauses to search for patterns in strings. * They are highly useful for filtering data based on partial matches. | | |
| Basic LIKE Syntax | SELECT column1, column2, ...  FROM table\_name  WHERE column LIKE pattern; |  |
| Using % (any number of characters): | SELECT \*  FROM employees  WHERE last\_name LIKE 'S%'; | * This query returns all employees whose last name starts with 'S' (e.g., 'Smith', 'Sanders') |
| Using \_ (exactly one character) | SELECT \*  FROM employees  WHERE last\_name LIKE 'J\_n'; | * This query returns all employees whose last name has three characters and starts with 'J' and ends with 'n' (e.g., 'Jan', 'Jon'). |
| Using % for matching multiple characters | SELECT \*  FROM employees  WHERE first\_name LIKE '%a%'; | * This returns all employees whose first name contains the letter 'a' anywhere (e.g., 'Maria', 'James'). |
| Using \_ for a single character match | SELECT \*  FROM products  WHERE product\_code LIKE 'A\_3'; | * This query matches product codes starting with 'A', followed by any character, and ending with '3' (e.g., 'AB3', 'AC3'). |
| Using [] to match specific characters (SQL Server). | SELECT \*  FROM customers  WHERE state\_code LIKE '[CK]A'; | * This matches all customers whose state codes are either 'CA' or 'KA'. |
| Using [^] to exclude specific characters (SQL Server): | SELECT \*  FROM customers  WHERE state\_code LIKE 'A[^BC]'; | * This matches state codes that start with 'A' but do not have 'B' or 'C' as the second letter (e.g., 'AK', 'AZ'). |
| Wildcards with NOT LIKE | SELECT \*  FROM employees  WHERE first\_name NOT LIKE '%a%'; | * This query returns all employees whose first name does **not** contain the letter 'a'. |
|  |  |  |
| SQL Aliases |  |  |
| * In SQL, aliases are used to give temporary names to columns or tables, making queries easier to read and manage, especially when working with complex or multi-joined queries. * They do not alter the original name of the table or column; instead, they serve as a short-term reference within a query. | | |
| 1. Column Aliases | SELECT column\_name AS alias\_name  FROM table\_name; | * **AS**: This keyword is optional. * You can directly specify the alias name after the column name, though using AS improves readability. |
| * A column alias assigns a temporary name to a column in the result set. * This is especially useful for renaming columns with long or complex names, or when performing calculations or aggregations. | | |
|  | SELECT first\_name AS fname, last\_name AS lname  FROM employees; | * This query returns **fname** and **lname instead** of **first\_name** and **last\_name** in the result set. |
| You can also use aliases for calculated fields | SELECT first\_name, last\_name, (salary \* 1.1) AS new\_salary  FROM employees; | * Here, **new\_salary** is an alias for the calculated column |
|  |  |  |
| 2. Table Aliases | SELECT column1, column2, ...  FROM table\_name AS alias\_name; | * Similar to column aliases, the **AS** keyword is optional but often used for clarity. |
| * A table alias provides a temporary name for a table within a query. * This is commonly used in complex queries, especially when you need to reference the same table multiple times (e.g., in self-joins) or to simplify long table names. | | |
|  | SELECT e.first\_name, e.last\_name, d.department\_name  FROM employees AS e  JOIN departments AS d ON e.department\_id = d.department\_id; | * **e** is used as an alias for the employees table. * **d** is an alias for the departments table. |
|  |  |  |
| 3. Aliases With JOINS |  |  |
| * In more complex queries with multiple tables, aliases significantly enhance readability and reduce code repetition. | | |
| Multiple Joins | SELECT e.first\_name, e.last\_name, d.department\_name, m.first\_name AS manager\_first\_name  FROM employees AS e  JOIN departments AS d ON e.department\_id = d.department\_id  JOIN employees AS m ON e.manager\_id = m.employee\_id; | * **e** is the alias for the employees table, referring to regular employees. * **m** is another alias for the employees table, but here it's used to reference the managers. |
| 4. Aliases Without AS | SELECT first\_name fname, last\_name lname  FROM employees e; | * Both versions are valid, but using AS is generally preferred for readability, especially in complex queries. |
| 4. Using Aliases in Subqueries |  |  |
| * Aliases are essential when working with subqueries, allowing you to refer to subquery results in the outer query. | | |
|  | SELECT avg\_salary.department\_id, avg\_salary.avg\_dept\_salary  FROM (SELECT department\_id, AVG(salary) AS avg\_dept\_salary  FROM employees  GROUP BY department\_id) AS avg\_salary  ORDER BY avg\_salary.avg\_dept\_salary DESC; | * **avg\_salary** is the alias for the subquery. * The alias allows the outer query to reference the results of the subquery. |
|  |  |  |
| SQL Dates |  |  |
| Most SQL databases have specific data types for storing dates and times. The exact types may vary depending on the database system, but here are the most common:   * DATE: Stores only the date (year, month, and day). Example: '2024-10-16'. * TIME: Stores only the time (hours, minutes, seconds). Example: '14:30:00'. * DATETIME or TIMESTAMP: Stores both date and time (year, month, day, hours, minutes, seconds). Example: '2024-10-16 14:30:00'. * YEAR: Stores only the year (typically a 4-digit number). Example: 2024.   Dates are typically stored in the YYYY-MM-DD format in SQL, although the exact format depends on the database you're using. Some databases accept other formats, such as MM-DD-YYYY or DD-MM-YYYY, but ISO 8601 (YYYY-MM-DD) is standard. | | |
| Inserting Date Values | INSERT INTO orders (order\_date)  VALUES ('2024-10-16'); |  |
| Extracting Dates | SELECT YEAR(order\_date), MONTH(order\_date), DAY(order\_date)  FROM orders; | * This query extracts the year, month, and day from the order\_date. |
| Getting the Current Date | SELECT CURDATE();  -- or  SELECT CURRENT\_DATE; | * SQL Server: SELECT GETDATE(); |
| Extracting Specific Parts of a Date | SELECT YEAR(order\_date) AS year,  MONTH(order\_date) AS month,  DAY(order\_date) AS day  FROM orders; |  |
| Calculating Date Differences | SELECT DATEDIFF('2024-10-16', '2024-01-01') AS days\_diff; |  |
| Adding/Subtracting Dates | SELECT DATE\_ADD('2024-10-16', INTERVAL 5 DAY); -- Adds 5 days  SELECT DATE\_SUB('2024-10-16', INTERVAL 1 MONTH); -- Subtracts 1 month |  |
| Comparing Dates | SELECT \*  FROM orders  WHERE order\_date >= '2024-01-01'; |  |
| Date Formatting | SELECT DATE\_FORMAT(order\_date, '%Y-%m-%d') AS formatted\_date  FROM orders; | SQL Server:  SELECT FORMAT(order\_date, 'yyyy-MM-dd') AS formatted\_date  FROM orders; |
|  |  |  |
| Insert, Read, Update, Delete the Data Rows (DML) | | |
| INSERT INTO statement | INSERT INTO table\_name (column1, column2, column3, ...)  VALUES (value1, value2, value3, ...); | * **table\_name**: The name of the table where you want to insert data. * **column1, column2, ...**: The columns in the table where values will be inserted. * **value1, value2, ...**: The values that correspond to each column. |
| * The structure of an INSERT INTO SQL statement adds new records to a table. | | |
|  | INSERT INTO employees (id, first\_name, salary)  VALUES (1, 'John', 50000); |  |
| Omitting Column Names | INSERT INTO employees  VALUES (2, 'Jane', 60000); | * If you’re inserting values for every column in the table, you can omit the column names |
| Multiple Rows | INSERT INTO employees (id, first\_name, salary)  VALUES (3, 'Alice', 55000), (4, 'Bob', 70000); | * To insert multiple rows in a single statement, separate each row with a comma. |
| Insert Data from Another Table  INSERT INTO … SELECT | INSERT INTO target\_table (column1, column2, ...)  SELECT column1, column2, ...  FROM source\_table  WHERE condition; | INSERT INTO high\_salary\_employees (id, first\_name, last\_name, salary)  SELECT id, first\_name, last\_name, salary  FROM employees  WHERE salary > 70000; |
|  |  |  |
| UPDATE statement | UPDATE table\_name  SET column1 = value1, column2 = value2, ...  WHERE condition; | * **table\_name**: The table where you want to update data. * **SET**: Specifies the columns and their new values. * **column1 = value1, column2 = value2, ...**: The columns to update and their corresponding new values. * **WHERE condition**: A condition to specify which rows to update. Without it, all rows in the table will be updated. |
| * Used to modify existing records in a table. | | |
|  | UPDATE employees  SET salary = 65000  WHERE id = 1; | * Always use a **WHERE clause** to specify which rows to update unless you intend to update every row in the table. |
| Updating Multiple Columns | UPDATE employees  SET department = 'HR', salary = 70000  WHERE id = 2; |  |
| DELETE statement | DELETE FROM table\_name  WHERE condition; | * **table\_name**: The name of the table from which you want to delete data. * **WHERE condition**: A condition to specify which rows should be deleted. “**Without this clause, all rows in the table will be deleted**.” |
| * Used to remove rows from a table. | | |
|  | DELETE FROM employees  WHERE id = 3; | * **employees**: The table from which data is deleted. * **WHERE id = 3**: Deletes only the row where id is 3. |
| Deleting All Rows | DELETE FROM employees; | * **Always use the WHERE clause** unless you intend to delete every row in the table, as this can be difficult to reverse. |
| Removing Duplicate Records | DELETE FROM customers  WHERE id NOT IN (  SELECT MIN(id)  FROM customers  GROUP BY email  ); |  |
|  |  |  |
| SQL JOINS (Data Manipulation Language) | | |
| * SQL JOINs are powerful tools in the Data Manipulation Language (DML) for combining rows from two or more tables based on a related column. * They enable users to retrieve data that’s distributed across multiple tables in a relational database. | | |
| 1. INNER JOIN | SELECT columns  FROM table1  INNER JOIN table2 ON table1.common\_column = table2.common\_column; | * The **INNER JOIN** keyword selects records with **matching values in both tables.** * If there’s no match, the records are not included in the result set. |
|  | SELECT employees.name, departments.dept\_name  FROM employees  INNER JOIN departments ON employees.dept\_id = departments.dept\_id; | * This returns rows only where there’s a match between   employees.dept\_id and departments.dept\_id.  Both tables contain a column name **“dept\_id”** |
| 1. LEFT JOIN | SELECT column1, column2, ...  FROM table1  LEFT JOIN table2 ON table1.common\_column = table2.common\_column; | * It retrieves all records from the left table (the first table specified), along with the matched records from the right table (the second table specified). * If there is no match, NULL values will be returned for columns from the right table. |
| 1. RIGHT JOIN | SELECT column1, column2, ...  FROM table1  RIGHT JOIN table2 ON table1.common\_column = table2.common\_column; | * It retrieves all records from the right table (the second table specified), along with the matched records from the left table (the first table specified). * If there is no match, NULL values will be returned for columns from the left table. |
| 1. FULL JOIN (use UNION) | SELECT column1, column2, ...  FROM table1  LEFT JOIN table2 ON table1.common\_column = table2.common\_column;  UNION  SELECT column1, column2, ...  FROM table1  RIGHT JOIN table2 ON table1.common\_column = table2.common\_column; | * It retrieves all records from both the left table and the right table, and where there is no match. * it returns NULL values for the columns from the table that lacks a corresponding entry. |
| 1. CROSS JOIN | SELECT column1, column2, ...  FROM table1  CROSS JOIN table2; | * Produces a Cartesian product of two tables, meaning it returns all possible combinations of rows from both tables. * This type of join does not require any condition to join the tables; instead, it pairs each row from the first table with every row from the second table. |
| 1. UNION operator | SELECT column1, column2, ...  FROM table1  UNION  SELECT column1, column2, ...  FROM table2; | * Used to combine the results of two or more SELECT queries into a single result set. * The queries must have the same number of columns and compatible data types in each column, as UNION aligns columns by position rather than by name. * By default, UNION removes duplicate rows from the result set. |
| 1. SQL VIEWS | CREATE VIEW view\_name AS  SELECT column1, column2, ...  FROM table\_name  WHERE condition; | * Is a virtual table created by a query that selects data from one or more underlying tables. * Unlike a physical table, a view does not store data itself; it dynamically retrieves data from the original tables each time it’s accessed. * This makes views useful for simplifying complex queries, enhancing security by restricting data access, and providing data abstraction for users. |
|  | CREATE VIEW simplified\_employee\_data AS  SELECT employee\_id, employee\_name, department\_name, location  FROM employees  JOIN departments ON employees.department\_id = departments.department\_id  JOIN locations ON departments.location\_id = locations.location\_id; | * If your database has a complex structure, you can create views to simplify the user experience. |
|  | CREATE OR REPLACE VIEW view\_name AS  SELECT column1, column2, ...  FROM table\_name  WHERE condition; | * **Updating a View**: Modify the structure or conditions of a view using CREATE OR REPLACE VIEW. |
|  | DROP VIEW view\_name; | * **Dropping a View**: Remove a view when it’s no longer needed |
| 1. Derived Tables or Inline Views | SELECT column1, column2, ...  FROM (SELECT column1, column2, ...  FROM table\_name  WHERE condition) AS derived\_table\_name  WHERE outer\_condition; | * **Derived Tables**/ **Inline Views**, are subqueries used within the FROM clause of an SQL query. * They are essentially temporary tables created by a query for use within the same SELECT statement. * Derived tables are useful for breaking down complex queries, creating intermediate results, or aggregating data in nested steps. |
|  |  |  |
| SQL Functions | | |
| * Are built-in operations used to perform calculations, manipulate data, format results, or carry out operations within SQL queries. * Functions can be categorized mainly into aggregate functions, scalar functions, string functions, date functions, mathematical functions, and conversion functions. | | |
| 1. COUNT() | SELECT COUNT(\*)  FROM table\_name  WHERE condition; | * Is an aggregate function used to return the number of rows in a result set that meet specified criteria. * It’s particularly useful when you want to find out how many rows exist in a table, how many distinct values a column has, or how many entries meet a specific condition. |
|  | SELECT  COUNT(\*) AS total\_employees,  COUNT(salary) AS employees\_with\_salary,  COUNT(DISTINCT department) AS unique\_departments  FROM employees; | * Query Combining Multiple Counts |
| 1. FIRST() | SELECT \* FROM table1 LIMIT 1. | * Is used to retrieve the first value in an ordered column. * It’s worth noting that **not all SQL databases support the FIRST() function directly**; some databases, such as Oracle, MySQL, and PostgreSQL, achieve the same result through different techniques, such as using ORDER BY with LIMIT or window functions. |
| 1. LAST() | SELECT \*  FROM table\_name  ORDER BY column\_name DESC  LIMIT 1; | * Used to retrieve the *last records* in a table * You can typically use a combination of ORDER BY and LIMIT clauses, as SQL databases generally don’t have a built-in LAST() function. |
|  | SELECT \*  FROM (  SELECT \*  FROM transactions  ORDER BY transaction\_date DESC  LIMIT 5  ) AS last\_five  ORDER BY transaction\_date ASC; | * If you want these 5 transactions to appear in chronological order, you can wrap this query in a subquery |
| 1. SUM() | SELECT SUM(column\_name)  FROM table\_name  WHERE condition; | * Is an aggregate function that calculates the total sum of a numeric column across rows in a result set. * It’s commonly used for financial data, inventory counts, and other scenarios where you need the total or cumulative value of a field. |
| 1. MIN() | SELECT MIN(column\_name)  FROM table\_name  WHERE condition; | * Is an aggregate function that returns the smallest (minimum) value from a specified column. * It's particularly useful for identifying the lowest value in a set of numeric or date values, such as finding the earliest date, the lowest price, or the smallest quantity. |
| 1. MAX() | SELECT MAX(column\_name)  FROM table\_name  WHERE condition; | * Used to return the largest (maximum) value from a specified column. * It’s commonly used to find the highest value in a set of numeric or date data, such as the maximum salary, latest date, or highest score. |
| 1. AVG() | SELECT AVG(column\_name)  FROM table\_name  WHERE condition; | * It calculates the average (mean) value of a specified numeric column across a set of rows. |
|  |  |  |
| SQL Data Types | | |
| * SQL supports various data types to define the nature of data that can be stored in a database table. * Understanding these data types is essential for designing efficient and effective database schemas. | | |
| 1. Numeric Data Types | Numeric data types are used to store numerical values. |  |
| * INT: Stores integers. Size can vary based on the database system (e.g., TINYINT, SMALLINT, MEDIUMINT, BIGINT for larger integers). * FLOAT: Stores floating-point numbers. Used for approximate numeric values. * DOUBLE: Stores double-precision floating-point numbers. More precision than FLOAT. * DECIMAL(p, s): Stores exact numeric data with a specified precision (p) and scale (s).   Useful for financial calculations to avoid rounding errors | | |
|  |  |  |
| 1. String Data Types | String data types are used for storing text and character data. |  |
| * CHAR(n): Fixed-length character string. Pads with spaces if the input string is shorter than n. * VARCHAR(n): Variable-length character string. Stores up to n characters and uses only the necessary space. * TEXT: Stores large amounts of text data. No specific size limit in most systems. * NCHAR(n): Fixed-length Unicode character string. * NVARCHAR(n): Variable-length Unicode character string. | | |
|  |  |  |
| 1. Date and Time Data Types | These data types are used for storing dates and times. |  |
| * DATE: Stores date values (year, month, day). * TIME: Stores time values (hours, minutes, seconds). * DATETIME: Stores date and time values together. * TIMESTAMP: Similar to DATETIME, often used to store the current date and time when a record is created or modified. * INTERVAL: Represents a time span or duration (available in some SQL implementations). | | |
|  |  |  |
| 1. Boolean Data Types |  |  |
| * BOOLEAN: Stores a truth value (TRUE, FALSE). * In some systems, it may be represented as BIT or integers (0 for false, 1 for true). | | |
|  |  |  |
| 1. Binary Data Types | Used for storing binary data (like images, files). |  |
| * BINARY(n): Fixed-length binary data. * VARBINARY(n): Variable-length binary data. * BLOB: Large binary object, used to store binary data like images or multimedia files. | | |
|  |  |  |
| 1. Special Data Types |  |  |
| * JSON: Stores JSON (JavaScript Object Notation) formatted data. * XML: Stores XML (Extensible Markup Language) data. * UUID: Universally Unique Identifier, often used for unique identifiers. | | |
|  |  |  |
| Data Definition Language | | |
| 1. Learn how to create and display databases (DB) | | |
| Step 1: Create a DB | CREATE DATABASE database\_name; | * database\_name : Name of the DB. * Naming conventions usually avoid spaces and use underscores or camel case. |
| Step 2: Display a List of All DB | SHOW DATABASES; | * To view a list of databases. * This command lists all databases available in the database server. |
| Step 3: Select (Use) a Database | USE database\_name; | * To start working with a specific database, you need to select it. * This allows you to perform actions within that database (e.g., creating tables, inserting data). |
| Check Current Database | SELECT DATABASE(); | * Some SQL environments allow you to check which database is currently in use. |
| Delete a Database (if needed) | DROP DATABASE database\_name; | * To remove a database. * Be careful, as this action is irreversible. |
|  |  |  |
| 1. Learn how to create the database table using right method | | |
| * Creating a database table in SQL involves defining its structure and specifying the data types for each column. * A well-designed table considers normalization, data types, constraints, and indexing to optimize performance and maintain data integrity. | | |
| Step 1: Define the Table Structure | **Column Names**: Define clear, meaningful names.  **Data Types**: Assign appropriate data types to each column.  **Primary Key**: Designate a unique identifier column.  **Constraints**: Set constraints (e.g., NOT NULL, UNIQUE, FOREIGN KEY, CHECK) as needed. |  |
| Step 2: Use the CREATE TABLE Syntax | CREATE TABLE table\_name (  column1 datatype constraint,  column2 datatype constraint,  ...  PRIMARY KEY (column\_name),  FOREIGN KEY (column\_name) REFERENCES other\_table(column\_name)  ); | * **table\_name**: The name of your table. * **datatype**: The data type of each column (e.g., INT, VARCHAR, DATE). * **constraint**: Optional; enforces rules like NOT NULL, UNIQUE, etc. * **PRIMARY KEY**: Ensures each record in the table is unique. * **FOREIGN KEY**: Links to a primary key in another table. |
| SHOW THE TABLES | SHOW TABLES; | * Display all tables in the database. |
| Verify the Table Creation | DESCRIBE name\_table;  EXPLAIN name\_table; | * Display informations about the table. |
|  |  |  |
| Example of TABLE  Let’s create a students table with columns for student\_id, first\_name, last\_name, birth\_date, and email. | CREATE TABLE students (  student\_id INT NOT NULL AUTO\_INCREMENT,  first\_name VARCHAR(50) NOT NULL,  last\_name VARCHAR(50) NOT NULL,  birth\_date DATE,  email VARCHAR(100) UNIQUE,  PRIMARY KEY (student\_id)  ); | * **student\_id** is an integer that auto-increments for each new student, serving as the primary key. * **first\_name** and **last\_name** are text fields that cannot be empty. * **birth\_date** is a date field. * **email** is unique, ensuring no two students share the same email. |
|  |  |  |
| 1. Modify or update the database table details | | |
| * To modify or update an existing table in SQL, the ALTER TABLE statement is commonly used. * This command allows you to make a range of changes to the table’s structure, such as adding new columns, modifying existing columns, renaming the table, or deleting columns. | | |
| 1. Add a New Column | ALTER TABLE table\_name  ADD column\_name datatype constraint; | * To add a column to an existing table, use the **ADD** clause. |
| 1. Modify an Existing Column | ALTER TABLE table\_name  MODIFY column\_name new\_datatype new\_constraint; | * To change the data type, size, or constraints of an existing column, use the **MODIFY** clause. |
| 1. Rename a Column | ALTER TABLE table\_name  RENAME COLUMN old\_column\_name TO new\_column\_name; | * To rename a column, use the **RENAME COLUMN** clause |
| 1. Drop a Column | ALTER TABLE table\_name  DROP COLUMN column\_name; | * To delete a column, use the **DROP COLUMN** clause. * Be careful, as this action permanently removes the column and its data. |
| 1. Rename the Table | ALTER TABLE old\_table\_name  RENAME TO new\_table\_name; | * To rename an entire table, use the **RENAME TO** clause. |
| 1. Add Indexes   Remove Indexes | CREATE INDEX index\_name ON table\_name(column\_name);  DROP INDEX index\_name ON table\_name; | * To optimize querying, you can add an index on a column or a group of columns. |
|  |  |  |
| 1. Remove or delete an existing table | | |
| * To remove or delete an existing table in SQL, you use the DROP TABLE statement. * This command will permanently delete the table, including all of its data, structure, and associated indexes. Use this operation with caution, as it cannot be undone. | | |
|  | DROP TABLE table\_name; | **table\_name**: The name of the table you want to delete. |
| Using IF EXISTS | DROP TABLE IF EXISTS table\_name; | * This will delete the table only if it exists, preventing an error if it’s not found. |
|  |  |  |
| 1. Understand Constraints |  |  |
| * Constraints should be used to enforce business rules and maintain data integrity. * Constraints like NOT NULL, CHECK, and UNIQUE help avoid invalid or duplicate data. * Foreign key constraints ensure that relationships between tables remain consistent. | | |
| 1. NOT NULL Constraint   Is typically specified when creating or altering a table. | CREATE TABLE employees (  employee\_id INT PRIMARY KEY,  first\_name VARCHAR(50) NOT NULL,  last\_name VARCHAR(50) NOT NULL,  email VARCHAR(100)  ); | * The columns first\_name and last\_name are marked as NOT NULL, so they cannot contain NULL values. * Every employee must have these fields filled in when a new record is inserted. |
| 1. UNIQUE Constraint | CREATE TABLE employees (  employee\_id INT PRIMARY KEY,  email VARCHAR(100) UNIQUE,  phone\_number VARCHAR(20) UNIQUE  ); | * Ensures that all values in a column (or a set of columns) are distinct, preventing duplicate entries in that column. * This constraint is useful for fields where each entry must be unique, such as usernames, email addresses, or other identifiers that don’t serve as the primary key. |
| 1. PRIMARY KEY |  |  |
| * Is a constraint that uniquely identifies each record in a table. The PRIMARY KEY constraint ensures two main things:Uniqueness:   Each value in the primary key column(s) is unique.  Non-nullability: The primary key columns cannot contain NULL values.   * Only one PRIMARY KEY constraint is allowed per table, and it can be applied to a single column or multiple columns (known as a composite key). * Typically, primary keys are used to create relationships between tables in a database and to allow efficient indexing of table records. | | |
| Defining a Primary Key on a Single Column | CREATE TABLE employees (  employee\_id INT PRIMARY KEY,  first\_name VARCHAR(50),  last\_name VARCHAR(50),  email VARCHAR(100)); | * **employee\_id** is the primary key, meaning each employee must have a unique ID, and it cannot be NULL. |
| Defining a Composite Primary Key on Multiple Columns | CREATE TABLE enrollments (  student\_id INT,  course\_id INT,  enrollment\_date DATE,  PRIMARY KEY (student\_id, course\_id)  ); | * The combination of student\_id and course\_id serves as the primary key. * This ensures that each student can enroll in each course only once. |
| Adding a Primary Key to an Existing Table | ALTER TABLE employees  ADD PRIMARY KEY (employee\_id); | * This statement sets employee\_id as the primary key of the employees table. |
|  |  |  |
| 1. FOREIGN KEY |  |  |
| * A FOREIGN KEY establishes a relationship between two tables. * It enforces referential integrity by ensuring that the values in the foreign key column must exist in the referenced column. * If a foreign key references a primary key, this creates a "many-to-one" or "one-to-many" relationship, depending on the context. * The FOREIGN KEY constraint can be set either when creating a table or by altering an existing one. | | |
| Creating a Foreign Key During Table Creation | CREATE TABLE customers (  customer\_id INT PRIMARY KEY,  customer\_name VARCHAR(100)  );  CREATE TABLE orders (  order\_id INT PRIMARY KEY,  order\_date DATE,  customer\_id INT,  FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)  ); | * The customer\_id column in the orders table is a foreign key. * This key references the customer\_id column in the customers table. * The FOREIGN KEY constraint ensures that each customer\_id in orders must exist in customers. |
| Adding a Foreign Key to an Existing Table | ALTER TABLE orders  ADD CONSTRAINT fk\_customer  FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id); | * the fk\_customer foreign key constraint is added to ensure customer\_id in orders exists in customers. |
|  |  |  |
| Data Control Language - DCL | | |
| 1. What is Data Control Language (DCL)? | | |
| * Access Control: DCL is used by database administrators to control user access, ensuring data privacy, security, and adherence to organizational policies. * Role-Based Access: Through DCL, roles can be defined with specific permissions, allowing administrators to assign users to roles rather than managing individual permissions. | | |
| 1. The GRANT statement |  |  |
| * Is used to provide specific permissions to users or roles on database objects like tables, views, sequences, and procedures. * It allows database administrators to control access rights and ensure data security by selectively granting privileges. | | |
|  | mysql -u root -p | * Is used to log into MySQL from the command line with the **root** user account. |
| mysql: This starts the MySQL command-line tool.  -u root: Specifies the MySQL user as root. You can replace root with any other MySQL username.  -p: Prompts for the password after entering the command. | | |
|  | USE mysql; | * Specify which database should be the default for subsequent SQL statements in the session. |
|  | CREATE USER 'ahmed'@'localhost' IDENTIFIED BY 'password'; | * To create a new MySQL user named ahmed with a specified password. |
| 'ahmed': The username you are creating.  @'localhost': Specifies that this user can only connect from the localhost.  To allow remote access, replace 'localhost' with '%'.  IDENTIFIED BY 'password': Sets the password for the user. Replace 'password' with your desired password. | | |
|  | mysql -u ahmed -p | * To log into MySQL as the user ahmed |
|  |  |  |
| Grant Privileges | GRANT ALL PRIVILEGES ON \*.\* TO 'ahmed'@'localhost';  FLUSH PRIVILEGES; | * After creating the user, you might want to grant it specific privileges. * **ALL PRIVILEGES**: Grants all permissions. * **ON \*.\***: Specifies permissions on all databases and tables. |
|  |  |  |
| Grant Privileges on a Specific DB | GRANT ALL PRIVILEGES ON test\_db.\* TO 'ahmed'@'localhost'; | * To grant privileges on a particular database (e.g., test\_db) to the ahmed user. |
| Example | GRANT SELECT, INSERT, UPDATE ON test\_db.\* TO 'ahmed'@'localhost'; | * **ALL PRIVILEGES**: Grants all available privileges on the database. * Replace with specific privileges like SELECT, INSERT, UPDATE, etc., if full access is not needed. * **test\_db.\***: Specifies all tables in the test\_db database. |
| Grant Privileges on a Specific Table | GRANT SELECT, INSERT ON test\_db.orders TO 'ahmed'@'localhost'; | * To grant privileges on a particular table (e.g., orders in test\_db) * **test\_db.orders**: Specifies the orders table within the test\_db database. * **SELECT, INSERT**: Allows only SELECT and INSERT permissions on the specified table. |
| Applying and Confirming the Privileges | FLUSH PRIVILEGES; |  |
|  |  |  |
| 1. The REVOKE statement |  |  |
| * Is used to remove privileges from a user on specific databases or tables. * This is useful for managing user access as requirements change or when security needs tightening. | | |
|  | REVOKE privilege\_type ON database\_name.table\_name FROM 'username'@'host'; | * It removes specified permissions. |
| Revoke All Privileges on a DB | REVOKE ALL PRIVILEGES ON test\_db.\* FROM 'ahmed'@'localhost'; | * To revoke all privileges from a user (e.g., ahmed) on a specific database (e.g., test\_db) |
| Revoke Specific Privileges on a DB | REVOKE INSERT, UPDATE ON test\_db.\* FROM 'ahmed'@'localhost'; | * If ahmed only needs to lose specific privileges, such as INSERT and UPDATE on test\_db. |
| Revoke Privileges on a Specific Table | REVOKE SELECT ON test\_db.orders FROM 'ahmed'@'localhost'; | * To remove privileges only on a single table (e.g., orders in test\_db), specify the table |
| Applying the Changes | FLUSH PRIVILEGES; |  |
|  |  |  |
| Transaction Control Language (TCL) | | |
| * Used to manage transactions in a database. * A transaction is a logical unit of work that contains one or more SQL statements executed as a single entity. * TCL is essential for ensuring data integrity and consistency, allowing users to control how transactions are processed in a database management system. | | |
| 1. COMMIT |  |  |
| * Saves all changes made during the current transaction to the database. * Once a transaction is committed, the changes cannot be rolled back. | | |
|  | COMMIT; |  |
|  | START TRANSACTION;  UPDATE students SET age=5 WHERE studentid=1;  COMMIT; | * Update the students table, studentid=1’s age =5 |
|  |  |  |
| 1. ROLLBACK |  |  |
| * Undoes all changes made in the current transaction, reverting the database to its last committed state. * This is useful in case of errors or unexpected conditions during transaction execution. | | |
|  | ROLLBACK; |  |
| START TRANSACTION; -- Begin a transaction  -- Perform some operations  INSERT INTO accounts (account\_id, balance) VALUES (1, 1000);  INSERT INTO accounts (account\_id, balance) VALUES (2, 500);  -- Set a savepoint  SAVEPOINT transfer\_point;  -- Attempt to transfer funds  UPDATE accounts SET balance = balance - 200 WHERE account\_id = 1;  -- If the transfer fails or some condition is not met  -- ROLL BACK to the savepoint if needed  ROLLBACK TO transfer\_point;  -- Commit the transaction if everything is successful   * COMMIT; | | |
|  |  |  |
| 1. SAVEPOINT |  |  |
| * Allows you to set a point within a transaction to which you can later roll back. * This enables partial rollbacks rather than undoing the entire transaction. | | |
|  | SAVEPOINT savepoint\_name; |  |
|  |  |  |
|  |  |  |
| 1. SET TRANSACTION |  |  |
| * Configures the properties of the current transaction, such as its isolation level, which determines how transaction integrity is visible to other transactions. | | |
|  | SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; |  |
| Example | | |
| Scenario: E-Commerce Order Processing | | |
| * In this example, we'll simulate an order processing system where we update the inventory and create a record for a new order. The order processing must be handled carefully, as we want to ensure that both inventory updates and order entries are consistent. | | |
| Database Setup | Assume we have the following two tables: |  |
| 1. Products Table |  |  |
| CREATE TABLE products (  product\_id INT PRIMARY KEY,  product\_name VARCHAR(100),  stock INT  );  INSERT INTO products (product\_id, product\_name, stock) VALUES  (1, 'Laptop', 10),  (2, 'Smartphone', 20); | | |
| 1. Orders Table |  |  |
| CREATE TABLE orders (  order\_id INT PRIMARY KEY AUTO\_INCREMENT,  product\_id INT,  quantity INT,  order\_date DATETIME,  FOREIGN KEY (product\_id) REFERENCES products(product\_id)  ); | | |
| Example SQL Transaction  Now, we’ll perform the following steps within a transaction:   1. Reduce the stock of a product when an order is placed. 2. Insert a record into the orders table. 3. Use savepoints to manage partial rollbacks. 4. Set the transaction characteristics. | | |
| Step 1: Create a Database | CREATE DATABASE ecommerce; |  |
| Step 2: Select the Database | USE ecommerce; |  |
| Step 3: Create the Products Table | CREATE TABLE products (  product\_id INT PRIMARY KEY,  product\_name VARCHAR(100),  stock INT  ); |  |
| Step 4: Insert Sample Data (Optional) | INSERT INTO products (product\_id, product\_name, stock) VALUES  (1, 'Laptop', 10),  (2, 'Smartphone', 20); |  |
| Step 5: Verify the Table Creation | SHOW TABLES; -- To see all tables in the current database  DESCRIBE products; -- To see the structure of the products table |  |
|  | | |
| -- Step 1: Set the autocommit mode  SET AUTOCOMMIT = 0; -- Disable autocommit to manage transactions manually  -- Step 2: Start the transaction  START TRANSACTION;  -- Step 3: Set the transaction isolation level  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- Prevent dirty reads  -- Step 4: Check stock before placing the order  DECLARE @product\_id INT = 1;  DECLARE @quantity INT = 5;  -- Check if enough stock is available  IF (SELECT stock FROM products WHERE product\_id = @product\_id) >= @quantity THEN  -- Savepoint before updating stock  SAVEPOINT before\_stock\_update;  -- Step 5: Update product stock  UPDATE products  SET stock = stock - @quantity  WHERE product\_id = @product\_id;  -- Step 6: Insert order record  INSERT INTO orders (product\_id, quantity, order\_date)  VALUES (@product\_id, @quantity, NOW());  -- Step 7: Commit the transaction if everything is fine  COMMIT;  SELECT 'Order placed successfully. Stock updated.' AS message;  ELSE  -- If stock is insufficient, roll back to the savepoint  ROLLBACK TO SAVEPOINT before\_stock\_update;  -- Optionally rollback the entire transaction if needed  ROLLBACK;  SELECT 'Transaction failed: Insufficient stock. Changes have been rolled back.' AS message;  END IF;  -- Step 8: Set autocommit back to default if necessary  SET AUTOCOMMIT = 1; -- Re-enable autocommit for future operations | | |
|  |  |  |
| Database Relationships | | |
| * Define how tables in a relational database relate to each other. * They help to maintain data integrity, eliminate redundancy, and allow for complex queries across multiple tables. | | |
| 1. One-to-One (1:1) Relationship | | |
| * Each record in one table is associated with exactly one record in another table, and vice versa. | | |
| Example:  A user table and a user\_profile table where each user has only one profile, and each profile is linked to one user.  Implementation:  Usually implemented with a foreign key in one table that references the primary key of the other table. | | |
| -- Example Tables  CREATE TABLE user (  user\_id INT PRIMARY KEY,  username VARCHAR(50)  );  CREATE TABLE user\_profile (  profile\_id INT PRIMARY KEY,  user\_id INT,  bio TEXT,  FOREIGN KEY (user\_id) REFERENCES user(user\_id)  ); | | |
| 1. One-to-Many (1:M) Relationship | | |
| * A record in one table can be associated with multiple records in another table, but each record in the second table links back to only one record in the first table. | | |
| Example:  A department table and an employee table where each department can have multiple employees, but each employee belongs to only one department.  Implementation:  The employee table has a foreign key referencing the department table’s primary key. | | |
| -- Example Tables  CREATE TABLE department (  department\_id INT PRIMARY KEY,  department\_name VARCHAR(50)  );  CREATE TABLE employee (  employee\_id INT PRIMARY KEY,  name VARCHAR(50),  department\_id INT,  FOREIGN KEY (department\_id) REFERENCES department(department\_id)  ); | | |
| 1. Many-to-Many (M:M) | | |
| * Records in one table can be associated with multiple records in another table, and vice versa. | | |
| Example:  A student table and a course table where each student can enroll in multiple courses, and each course can have multiple students.  Implementation:  An intermediate (junction) table with foreign keys that reference the primary keys of both tables is used to handle this relationship. | | |
| -- Example Tables  CREATE TABLE student (  student\_id INT PRIMARY KEY,  student\_name VARCHAR(50)  );  CREATE TABLE course (  course\_id INT PRIMARY KEY,  course\_name VARCHAR(50)  );  CREATE TABLE enrollment (  student\_id INT,  course\_id INT,  PRIMARY KEY (student\_id, course\_id),  FOREIGN KEY (student\_id) REFERENCES student(student\_id),  FOREIGN KEY (course\_id) REFERENCES course(course\_id)  ); | | |
| 1. Self-Referencing (Recursive) Relationship | | |
| * A table has a relationship with itself, often used to represent hierarchical data structures. | | |
| Example:  An employee table where each employee can have a manager\_id that references another employee as their manager.  Implementation:  The table has a foreign key that references its primary key. | | |
| -- Example Table  CREATE TABLE employee (  employee\_id INT PRIMARY KEY,  name VARCHAR(50),  manager\_id INT,  FOREIGN KEY (manager\_id) REFERENCES employee(employee\_id)  ); | | |
|  |  |  |
| Database Normalization | | |
| * Database normalization is a systematic approach to organizing data in a relational database to minimize redundancy and improve data integrity. * The main goal of normalization is to ensure that the data is stored logically and efficiently. | | |
| Key Concepts   1. Redundancy: Storing the same data in multiple places, which can lead to inconsistencies. 2. Data Integrity: Ensuring the accuracy and consistency of data throughout its lifecycle. 3. Functional Dependency: A relationship where one attribute determines another attribute. | | |
| 1. First Normal Form (1NF) | | |
| Criteria:   * Each table cell should contain atomic (indivisible) values. * Each record needs to be unique (typically achieved using a primary key). * No repeating groups or arrays. | | |
|  | | |
| 1. Second Normal Form (2NF) | | |
| Criteria:   1. Must be in 1NF. 2. All non-key attributes must be fully functionally dependent on the primary key (no partial dependency). | | |
|  | | |
| 1. Third Normal Form (3NF) | | |
| Criteria:   * Must be in 2NF. * No transitive dependencies (non-key attributes should not depend on other non-key attributes). | | |
|  | | |
|  |  |  |
| SQL Database Export and Import | | |
| * Exporting and importing SQL databases is essential for data backup, migration, and sharing between different database systems. | | |
| 1. Exporting a Database | | |
| To export a MySQL database, you can use the mysqldump command. Here’s the basic syntax:  mysqldump -u [username] -p [database\_name] > [output\_file.sql] | | |
| Example |  |  |
| mysqldump -u root -p my\_database > my\_database\_export.sql | | |
| Explanation:   * -u root: Specifies the username. * -p: Prompts for the password. * my\_database: The name of the database you want to export. * my\_database\_export.sql: The name of the output SQL file. | | |
| Exporting Specific Tables |  |  |
| If you want to export specific tables from a database, you can list them after the database name:  mysqldump -u [username] -p [database\_name] [table1] [table2] > [output\_file.sql] | | |
| Example:  mysqldump -u root -p my\_database table1 table2 > my\_tables\_export.sql | | |
|  |  |  |
| 1. Importing a Database | | |
| To import a SQL file into a MySQL database, you can use the mysql command.  mysql -u [username] -p [database\_name] < [input\_file.sql] | | |
|  |  |  |
| Example:  mysql -u root -p my\_database < my\_database\_export.sql | | |
| Explanation:  -u root: Specifies the username.  -p: Prompts for the password.  my\_database: The name of the database where you want to import the data.  my\_database\_export.sql: The SQL file containing the exported database. | | |
|  |  |  |
| MySQL Workbench | | |
| * 1. Create a new user and connect to database using MySQL Workbench | | |
|  |  |  |
| 1. Create, Alter, Drop database using MySQL Workbench | | |
|  |  |  |
|  |  |  |