Tajamul Khan

SQL Cheat Sheet

@Tajamulkhann

Data Definition

CREATE TABLE: Creates a new table.
CREATE TABLE table_name (id INT PRIMARY KEY, name VARCHAR(50));

ALTER TABLE: Modifies an existing table.
ALTER TABLE table_name ADD column2 INT;

DROP TABLE: Deletes a table.
DROP TABLE table_name;

CREATE INDEX: Creates an index on a table.
CREATE INDEX idx_name ON table_name (column1);

DROP INDEX: Removes an index.
DROP INDEX idx_name ON table_name;

CREATE VIEW: Creates virtual table based on query.
CREATE VIEW view_name AS SELECT column1, column2 FROM table_name;

DROP VIEW: Deletes a view.
DROP VIEW view_name;

RENAME TABLE: Renames an existing table.
RENAME TABLE old_table_nm TO new_table_name;

@Tajamulkhann

Select Data

SELECT: Retrieves specific column from table.
SELECT column1, column2 FROM table_name;

DISTINCT: Removes duplicate rows from result.
SELECT DISTINCT column1 FROM table_name;

WHERE: Filters rows based on a condition.
SELECT * FROM table_name WHERE column1 = 'v1';

ORDER BY: Sorts result set by one or more columns.
SELECT * FROM table_nm ORDER BY column1 ASC;

LIMIT / FETCH: Limits the number of rows returned.
SELECT * FROM table_name LIMIT 10;

LIKE: Searches for patterns in text columns.
SELECT * FROM table_name WHERE col1 LIKE 'A%';

IN: Filters rows with specific values.
SELECT * FROM table_nm WHERE col1 IN ('v1', 'v2');

BETWEEN: Filters rows within a range of values.
SELECT * FROM table WHERE c1 BETWEEN 1 AND 20;

@Tajamulkhann

Aggregate Data

COUNT(): Returns the number of rows.
SELECT COUNT(*) FROM table_name;

SUM(): Calculates the sum of a numeric column.
SELECT SUM(column1) FROM table_name;

AVG(): Calculates the average of a numeric column.
SELECT AVG(column1) FROM table_name;

MIN(): Returns the smallest value in a column.
SELECT MIN(column1) FROM table_name;

MAX(): Returns the largest value in a column.
SELECT MAX(column1) FROM table_name;

GROUP BY: Groups rows for aggregation.
SELECT col1, COUNT(*) FROM t1 GROUP BY col1;

HAVING: Filters grouped rows based on a condition.
SELECT column1, COUNT(*) FROM t1 GROUP BY column1 HAVING COUNT(*) > 5;

DISTINCT COUNT(): Counts unique values in column.
SELECT COUNT(DISTINCT col1) FROM table_name;

@Tajamulkhann

Data Manipulation

INSERT INTO: Adds new rows to a table.
INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');

UPDATE: Updates existing rows in a table.
UPDATE table_name SET col1 = 'value' WHERE id = 1;

DELETE: Removes rows from a table.
DELETE FROM table_name WHERE column1 = 'value';

MERGE: Combines INSERT, UPDATE, and DELETE based on a condition.
MERGE INTO table_name USING source_table ON condition WHEN MATCHED THEN UPDATE SET column1 = value WHEN NOT MATCHED THEN INSERT (columns) VALUES (values);

TRUNCATE: Removes all rows from a table without logging.
TRUNCATE TABLE table_name;

REPLACE: Deletes existing rows and inserts new rows (MySQL-specific).
REPLACE INTO table_name VALUES (value1, value2);

@Tajamulkhann

Transactions

Commit Transaction: Finalizes changes when all operations succeed.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
WHERE id = 2; COMMIT;
```

Execute a Stored Procedure: Undoes changes if an error occurs or the transaction is not committed.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
ROLLBACK;
```

Using Savepoints: Set a rollback point within a transaction, allowing partial rollback without affecting the whole transaction.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
SAVEPOINT sp1;
UPDATE accounts SET balance = 2000 WHERE id = 3;
-- Simulate failure
ROLLBACK TO SAVEPOINT sp1;
UPDATE accounts SET balance = 1000 WHERE id = 2;
COMMIT;
```

@Tajamulkhann

Set Operations

UNION: Combines results from two queries, removing duplicates.
SELECT column1 FROM table1 UNION SELECT column1 FROM table2;

UNION ALL: Combines results from two queries, including duplicates.
SELECT column1 FROM table1 UNION ALLSELECT column1 FROM table2;

INTERSECT: Returns common rows from both queries.
SELECT column1 FROM table1 INTERSECT SELECT column1 FROM table2;

EXCEPT (or MINUS): Returns rows from the first query that are not in the second query.
SELECT column1 FROM table1 EXCEPTSELECT column1 FROM table2;

@Tajamulkhann

Table Joins

INNER JOIN: matching values in both tables.
SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id;

LEFT JOIN: Returns all rows from the left table and matching rows from the right table.
SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id;

RIGHT JOIN: Returns all rows from the right table and matching rows from the left table.
SELECT * FROM table1 RIGHT JOIN table2 ON table1.id = table2.id;

FULL OUTER JOIN: Returns rows when there is a match in either table.
SELECT * FROM table1 FULL OUTER JOIN table2 ON table1.id = table2.id;

CROSS JOIN: Cartesian product of both tables.
SELECT * FROM table1 CROSS JOIN table2;

SELF JOIN: Joins a table with itself.
SELECT a.column1, b.column1 FROM table_name a, table_name b WHERE a.id = b.parent_id;

@Tajamulkhann

Other Functions

CONCAT(): Concatenates strings.
SELECT CONCAT(first_name, ' ', last_name) FROM table_name;

SUBSTRING(): Extracts a substring from a string.
SELECT SUBSTRING(column1, 1, 5) FROM table_nm;

LENGTH(): Returns the length of a string.
SELECT LENGTH(column1) FROM table_name;

ROUND(): Rounds a number to a specified number of decimal places.
SELECT ROUND(column1, 2) FROM table_name;

NOW(): Returns the current timestamp.
SELECT NOW();

DATE_ADD(): Adds a time interval to a date.
SELECT DATE_ADD(NOW(), INTERVAL 7 DAY);

COALESCE(): Returns the first non-null value.
SELECT COALESCE(column1, column2) FROM table_name;

IFNULL(): Replaces NULL values with desired value.
SELECT IFNULL(col1, 'default') FROM table_name;

@Tajamulkhann

Window Functions

ROW_NUMBER: Assigns a unique number to each row in a result set.

```
SELECT ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_num FROM employees;
```

RANK: Assigns a rank to each row, with gaps for ties.

```
SELECT RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank FROM employees;
```

DENSE_RANK: Assigns a rank to each row without gaps for ties.

```
SELECT DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dense_rank FROM employees;
```

NTILE: Divides rows into equal parts.

```
SELECT NTILE(4) OVER (ORDER BY salary) AS quartile FROM employees;
```

LEAD(): Accesses subsequent rows' data.

```
SELECT name, salary, LEAD(salary) OVER (ORDER BY salary) AS next_salary FROM employees;
```

LAG(): Accesses subsequent rows' data.

```
SELECT name, salary, LAG(salary) OVER (ORDER BY salary) AS previous_salary FROM employees;
```

@Tajamulkhann

Stored Procedures

Create a Stored Procedure:

```
CREATE PROCEDURE sp_GetEmployeeByID
@EmployeeID INT
AS
BEGIN
-- SQL statements inside the stored
procedure
SELECT * FROM Employees
WHERE EmployeeID = @EmployeeID;
```

Execute a Stored Procedure:

```
EXEC sp_GetEmployeeByID @EmployeeID = 1;
```

Stored Procedure with OUT Parameter:

```
CREATE PROCEDURE GetEmployeeCount (OUT
emp_count INT) BEGINSELECT COUNT(*) INTO
emp_count FROM employees; END;
```

Drop a Stored Procedure:

```
DROP PROCEDURE GetEmployeeDetails;
```

@Tajamulkhann

Triggers

Create a Trigger (Before Insert):

```
CREATE TRIGGER set_created_at
BEFORE INSERT ON employees
FOR EACH ROW
SET NEW.created_at = NOW();
```

After Update Trigger:

```
CREATE TRIGGER log_updates
AFTER UPDATE ON employees
FOR EACH ROW
INSERT INTO audit_log(emp_id, old_salary,
new_salary, updated_at)
VALUES (OLD.id, OLD.salary, NEW.salary,
NOW());
```

After Delete Trigger:

```
CREATE TRIGGER log_deletes
AFTER DELETE ON employees
FOR EACH ROW
INSERT INTO audit_log(emp_id, old_salary,
new_salary, deleted_at)
VALUES (OLD.id, OLD.salary, NULL, NOW());
```

@Tajamulkhann

Subquery

Scalar Subquery:

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM
employees);
```

Correlated Subquery:

```
SELECT e1.name, e1.salary
FROM employees e1
WHERE e1.salary > (SELECT AVG(e2.salary)
FROM employees e2 WHERE e1.department =
e2.department);
```

@Tajamulkhann

CTE

With a Single CTE:

```
WITH DepartmentSalary AS (
    SELECT department, AVG(salary) AS
avg_salary
    FROM employees
    GROUP BY department
)
SELECT *
FROM DepartmentSalary
WHERE avg_salary > 50000;
```

Recursive CTE:

```
WITH RECURSIVE Numbers AS (
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1
    FROM Numbers
    WHERE num < 10)
SELECT * FROM Numbers;
```

@Tajamulkhann

Indexes

Create an Index:
```
CREATE INDEX idx_department ON
employees(department);
```

Unique Index: `CREATE UNIQUE INDEX idx_unique_email ON employees(email);`

Drop an Index:
```
DROP INDEX idx_department;
```

Clustered Index (SQL Server):
```
CREATE CLUSTERED INDEX idx_salary ON
employees(salary);
```

Using EXPLAIN to Optimize:
```
EXPLAIN SELECT * FROM employees WHERE
salary > 50000;
```

@Tajamulkhann

Found Helpful ?

Repost

Follow for more!