

SQL Cheat Sheet



Steve Nouri

@SteveNouri

SQL Commands, Functions & Clauses

-- Basic SQL commands:

SELECT - retrieves data from a database
FROM - specifies which tables to retrieve data from
WHERE - specifies which rows to retrieve based on certain conditions
GROUP BY - groups rows that have the same values in the specified columns
HAVING - filters groups based on a specified condition
ORDER BY - sorts the retrieved rows in a specified order

-- Aggregate functions:

AVG() - returns the average value of a set of values
COUNT() - returns the number of rows in a table or the number of non-null values in a column
FIRST() - returns the first value in a set of values
LAST() - returns the last value in a set of values
MAX() - returns the maximum value in a set of values
MIN() - returns the minimum value in a set of values
SUM() - returns the sum of a set of values

-- String functions:

CONCAT() - concatenates two or more strings together
INSTR() - returns the position of a substring within a string
LENGTH() - returns the length of a string
LOWER() - converts a string to lowercase
LTRIM() - removes leading spaces from a string
REPLACE() - replaces all occurrences of a specified string with another string
RTRIM() - removes trailing spaces from a string
SUBSTR() - returns a portion of a string
TRIM() - removes leading and trailing spaces from a string
UPPER() - converts a string to uppercase

-- Date functions:

CURDATE() - returns the current date
CURTIME() - returns the current time
DATE() - extracts the date portion from a date/time value
DATEADD() - adds a specified time interval to a date
DATEDIFF() - returns the difference between two dates
DATEPART() - extracts a specified part of a date/time value
DAY() - returns the day of the month for a date value
MONTH() - returns the month for a date value
NOW() - returns the current date and time
YEAR() - returns the year for a date value

-- Other functions:

COALESCE() - returns the first non-null value in a list of values
IFNULL() - returns a specified value if a value is null
NULLIF() - returns null if two values are equal

-- Joins:

INNER JOIN - returns rows that have matching values in both tables
OUTER JOIN - returns rows that have matching values in either of the two tables
LEFT JOIN - returns all rows from the left table and any matching rows from the right table
RIGHT JOIN - returns all rows from the right table and any matching rows from the left table
FULL JOIN - returns all rows from both tables, whether or not there are matching values
CROSS JOIN - returns all rows from both tables, with each row from the first table being paired with each row from the second table

-- Set operations:

UNION - combines the results of two or more SELECT statements, eliminating duplicates
UNION ALL - combines the results of two or more SELECT statements, including duplicates
INTERSECT - returns rows that are present in the results of two or more SELECT statements
EXCEPT - returns rows that are present in the first SELECT statement but not in the results of any subsequent SELECT statements

-- Subqueries:

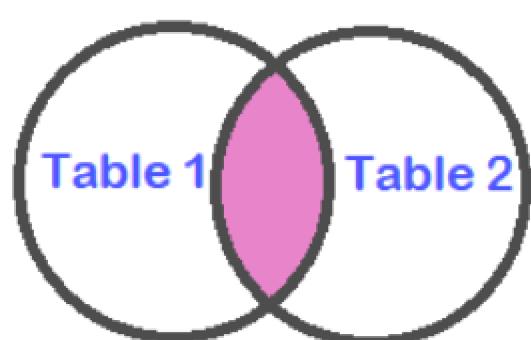
SELECT - retrieves data from a database within another SELECT statement
FROM - specifies which tables to retrieve data from within the subquery
WHERE - specifies which rows to retrieve based on certain conditions within the subquery
GROUP BY - groups rows that have the same values in the specified columns within the subquery
HAVING - filters groups based on a specified condition within the subquery

-- Data manipulation commands:

INSERT INTO - adds a new row to a table
UPDATE - modifies existing data in a table
DELETE - removes existing rows from a table
TRUNCATE TABLE - removes all rows from a table

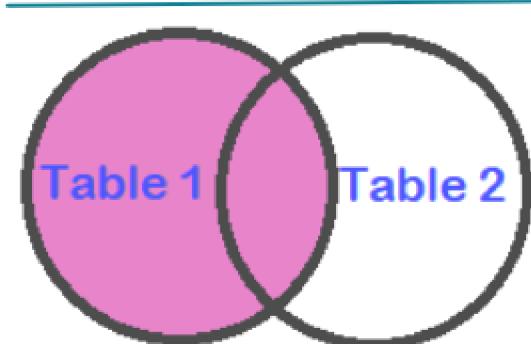
-- Data definition commands:

CREATE TABLE - creates a new table
ALTER TABLE - modifies the structure of an existing table
DROP TABLE - deletes a table
TRUNCATE TABLE - removes all rows from a table
CREATE INDEX - creates an index on a table to improve search performance
DROP INDEX - deletes an index from a table



-- INNER JOIN:

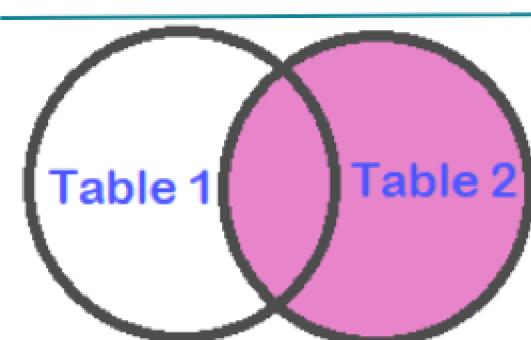
```
SELECT * FROM table1 INNER JOIN table2 ON table1.col1 = table2.col2
```



Outer Join

-- LEFT JOIN:

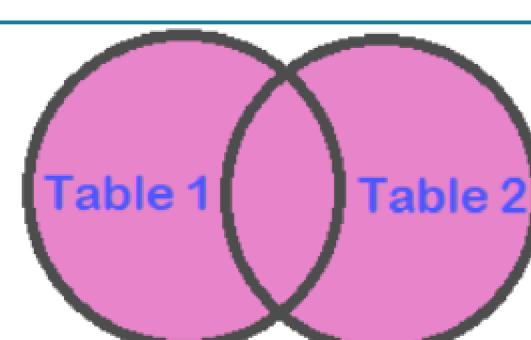
```
SELECT * FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2
```



Outer Join

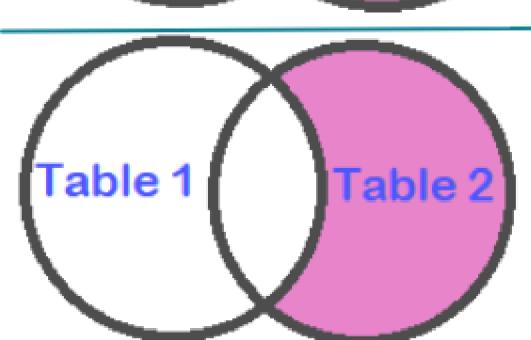
-- RIGHT JOIN:

```
SELECT * FROM table1 RIGHT JOIN table2 ON table1.col1 = table2.col2
```



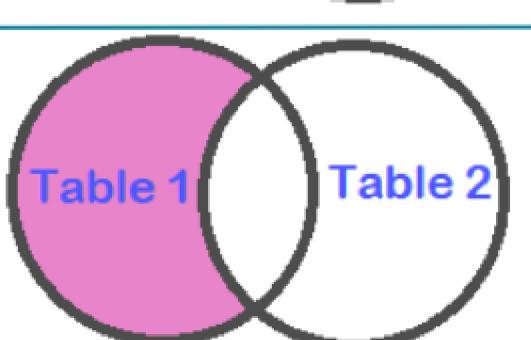
-- FULL OUTER JOIN:

```
SELECT * FROM table1 FULL OUTER JOIN table2 ON table1.col1 = table2.col2
```



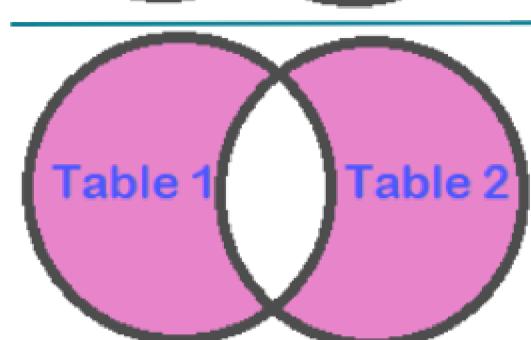
Right Outer Join with Exclusion

```
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.col1 = table2.col2
WHERE table1.col1 IS NULL;
```



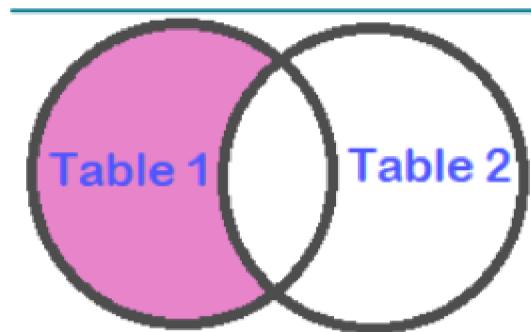
Left Outer Join with Exclusion

```
SELECT *
FROM table1
LEFT JOIN table2 ON table1.col1 = table2.col2
WHERE table2.col1 IS NULL;
```



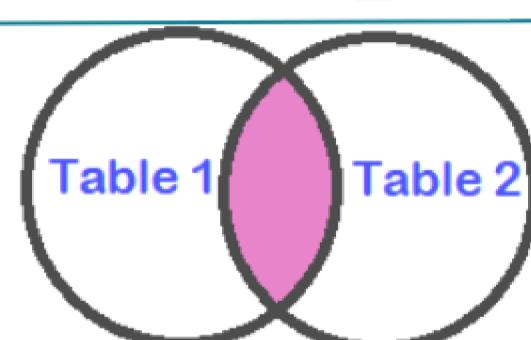
Full Outer Join with Exclusion

```
SELECT *
FROM table1
FULL OUTER JOIN table2 ON table1.col1 = table2.col2
WHERE table1.col1 IS NULL OR table2.col1 IS NULL;
```



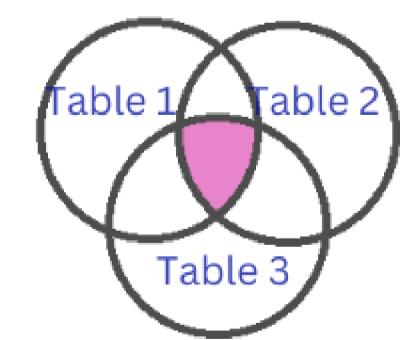
Anti Semi Join

```
SELECT *
FROM table1
WHERE NOT EXISTS (SELECT 1 FROM table2 WHERE table1.col1 = table2.col2);
```



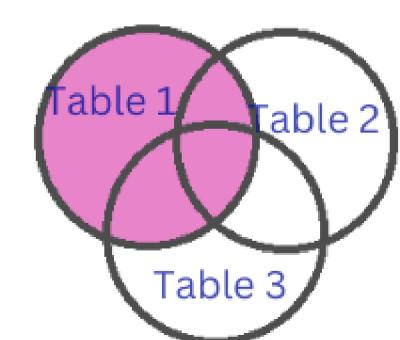
Semi Join (Less duplication than Inner Join)

```
SELECT *
FROM table1
WHERE EXISTS (SELECT 1 FROM table2 WHERE table1.col1 = table2.col2);
```



Two Inner Joins

```
SELECT *
FROM table1 INNER JOIN table2 ON table1.col1 = table2.col2
LEFT JOIN table3 ON table2.col3 = table3.col4;
```



Two Left Outer Joins

```
SELECT *
FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2
LEFT JOIN table3 ON table2.col3 = table3.col4;
```

-- NATURAL JOIN:

```
SELECT * FROM table1 NATURAL JOIN table2;
```

This type of join automatically matches rows from both tables based on their common columns. The common columns must have the same name and data type in both tables.

-- OUTER APPLY:

```
SELECT t1.*, t2.*
FROM table1 t1 OUTER APPLY table_valued_function(t1.col1) t2;
```

This type of join is similar to CROSS APPLY, but it returns all rows from the left table, even if there are no matching rows in the right table.

-- CROSS JOIN:

```
SELECT * FROM table1 CROSS JOIN table2 CROSS JOIN table3;
```

This type of join allows you to apply a table-valued function to each row in a table and join the results to the input rows.

-- SELF JOIN:

```
SELECT t1.col1, t2.col2
FROM table1 t1 INNER JOIN table1 t2 ON t1.col3 = t2.col4;
```

Returns all rows from the left table that do not have a match in the right table. Similar to a LEFT JOIN, but it only returns the rows from the left table that do not have a match in the right table,

Returns only the rows from the left table that have a match in the right table. Similar to an INNER JOIN, but it only returns the rows from the left table that have a match in the right table, rather than returning all rows from both.

This query will return rows from all three tables that have matching values in the joined columns. It will first join table1 and table2 based on the values in the col1 and col2 columns

It returns all rows from the leftmost table, as well as any matching rows from the other two tables. Rows from the other two tables that do not have a match in the leftmost table are NULL.

SQL Examples (Easy to Advanced)

-- Sample data for the "employees" table:

id	name	department	salary	hired_on
1	Alice	HR	55000	2020-01-01
2	Bob	Marketing	65000	2020-02-01
3	Charlie	IT	75000	2020-03-01
4	Dave	Sales	80000	2020-04-01
5	Eve	HR	60000	2020-05-01

-- Sample data for the "sales" table:

id	employee_id	product	sale_date	sale_amount
1	1	Widget	2020-06-01	1000
2	1	Gadget	2020-07-01	2000
3	2	Widget	2020-08-01	3000
4	2	Gadget	2020-09-01	4000
5	2	Thingamajig	2020-10-01	5000
6	3	Thingamajig	2020-11-01	6000
7	4	Widget	2020-12-01	7000
8	4	Gadget	2021-01-01	8000
9	4	Thingamajig	2021-02-01	9000
10	5	Widget	2021-03-01	10000

-- Sample data for the "products" table:

id	product	price
1	Widget	100
2	Gadget	200
3	Thingamajig	300

1- Retrieve all rows from the 'employees' table where the salary is greater than 65000.

```
SELECT * FROM employees WHERE salary > 65000;
```

2-Retrieve the name and department of all rows from the 'employees' table, sorted by salary in descending order.

```
SELECT name, department FROM employees ORDER BY salary DESC;
```

3-Retrieve the product and total sales for each product in the 'sales' table, sorted by total sales in descending order.

```
SELECT p.product, SUM(s.sale_amount) as total_sales FROM sales s
INNER JOIN products p ON s.product = p.product GROUP BY p.product
ORDER BY total_sales DESC;
```

4-Retrieve the name and total sales for all employees in the Marketing department from the 'employees' and 'sales' tables.

```
SELECT e.name, SUM(s.sale_amount) as total_sales FROM employees e
INNER JOIN sales s ON e.id = s.employee_id WHERE e.department =
'Marketing' GROUP BY e.name;
```

5-Retrieve the name, product, and sale date for all sales in the sales table where the sale amount is greater than 2500.

```
SELECT e.name, p.product, s.sale_date FROM employees e INNER JOIN
sales s ON e.id = s.employee_id INNER JOIN products p ON s.product =
p.product WHERE s.sale_amount > 2500;
```

6-Retrieve the name, product, and sale amount for all sales in the sales table that occurred between June 1, 2020 and December 31, 2020.

```
SELECT e.name, p.product, s.sale_amount FROM employees e INNER JOIN
sales s ON e.id = s.employee_id INNER JOIN products p ON s.product =
p.product WHERE s.sale_date BETWEEN '2020-06-01' AND '2020-12-31';
```

7-Retrieve the name, product, and sale amount for the sale with the highest sale amount in the sales table, along with the corresponding employee name from the employees table.

```
SELECT e.name, p.product, s.sale_amount FROM employees e INNER JOIN
sales s ON e.id = s.employee_id INNER JOIN products p ON s.product =
p.product WHERE s.sale_amount IN (SELECT MAX(sale_amount) FROM
sales);
```

8-Retrieve the name and manager of all employees in the employees table, showing the name of the manager from the employees table as well.

```
SELECT e1.name as manager, e2.name as employee FROM employees e1
INNER JOIN employees e2 ON e1.id = e2.manager_id;
```

9-Retrieve the name and number of employees for all managers in the employees table who have at least one employee.

```
SELECT e.name, (SELECT COUNT(*) FROM employees WHERE manager_id =
e.id) as num_employees FROM employees e WHERE num_employees > 0;
```

10-Retrieve the name of all employees in the employees table who do not have a manager.

```
SELECT e1.name, e2.name as manager FROM employees e1 LEFT JOIN
employees e2 ON e1.manager_id = e2.id WHERE e2.name IS NULL;
```

11-Retrieve the name and total sales for all employees in the employees table who have made more than 5000 in sales in the past year (from the current date).

```
SELECT e.name, SUM(s.sale_amount) as total_sales FROM employees e
INNER JOIN sales s ON e.id = s.employee_id WHERE s.sale_date BETWEEN
DATEADD(YEAR, -1, GETDATE()) AND GETDATE() GROUP BY e.name HAVING
SUM(s.sale_amount) > 5000;
```

Order of Execution

FROM
WHERE
GROUP BY
HAVING
SELECT
ORDER BY

This order can be modified by the use of subqueries or common table expressions (CTEs). In these cases, the subquery or CTE is executed first, and the results are used in the outer query.

Here is an example query to show the order of execution:

```
SELECT e.name, SUM(s.sale_amount) as total_sales
FROM employees e
INNER JOIN sales s ON e.id = s.employee_id
WHERE e.department = 'Sales' AND s.sale_date BETWEEN '2020-01-01' AND
'2020-06-30'
GROUP BY e.name
HAVING SUM(s.sale_amount) > 5000
ORDER BY total_sales DESC;
```

FROM and WHERE clauses are executed first to select the relevant rows from the employees and sales tables. These rows are then grouped by employee name and aggregated using the SUM function in the GROUP BY clause. The HAVING clause filters out any groups that do not meet the specified condition. Finally, the SELECT clause selects the name and total sales for each group, and the ORDER BY clause sorts the results by total sales in descending order.

Some Advanced Topics in SQL

1-Recursive queries: These are queries that can reference themselves in order to perform a certain action, such as querying hierarchical data.

-- Recursive queries:

```
WITH RECURSIVE cte_name AS (
    SELECT ...
    UNION [ALL]
    SELECT ...
    FROM cte_name
    WHERE ...
)
SELECT ...
FROM cte_name;
```

2-Window functions: These are functions that perform a calculation over a set of rows, similar to an aggregate function, but return a value for each row in the result set.

-- Window functions:

```
SELECT col1, col2, function_name(col3) OVER (PARTITION BY col1 ORDER BY
col2) as col4
FROM table_name;
```

3-Common table expressions (CTEs): These are named temporary result sets that can be used within a SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. They are often used to simplify complex queries by breaking them up into smaller, more manageable pieces.

-- Common table expressions (CTEs):

```
WITH cte_name AS (
    SELECT ...
)
SELECT ...
FROM cte_name;
```

4-Pivot tables: These are tables that allow you to transform data from rows to columns, or vice versa, in order to generate more meaningful insights from your data.

-- Pivot tables:

```
SELECT *
FROM (
    SELECT col1, col2, col3
    FROM table_name
)
PIVOT (
    SUM(col3)
    FOR col2 IN (val1, val2, val3)
);
```

5-Analytic functions: These are functions that perform a calculation over a group of rows and return a single value for each row in the result set.

-- Analytic functions:

```
SELECT col1, col2, function_name(col3) OVER (PARTITION BY col1 ORDER BY
col2) as col4
FROM table_name;
```

6-Triggers: These are pieces of code that are automatically executed in response to certain events, such as inserting, updating, or deleting data in a table.

-- Triggers:

```
CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
    -- trigger code here
END;
```

7--Stored procedures: These are pre-compiled SQL statements that can be stored in the database and executed repeatedly with different parameters.

-- Stored procedures:

```
CREATE PROCEDURE procedure_name (IN param1 datatype, IN param2
datatype, ...)
BEGIN
    -- procedure code here
END;
```

```
CALL procedure_name(param1_value, param2_value, ...);
```

8-Indexes: These are data structures that are used to improve the performance of certain types of queries by allowing the database to quickly locate the desired data.

-- Indexes:

```
CREATE INDEX index_name ON table_name (col1, col2, ...);
```

```
DROP INDEX index_name;
```

9-Cursor-based processing: This is a method of processing data in which a cursor is used to retrieve a small batch of rows from a result set, process the rows, and then retrieve the next batch of rows until all rows have been processed. This can be useful when working with large result sets or when the processing needs to be done row by row.

-- Cursor-based processing:

```
DECLARE cursor_name CURSOR FOR SELECT col1, col2, ... FROM table_name
WHERE ...;
```

```
OPEN cursor_name;
```

```
FETCH NEXT FROM cursor_name INTO variable1, variable2, ...;
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
```

-- processing code here

```
    FETCH NEXT FROM cursor_name INTO variable1, variable2, ...;
```

```
END;
```

```
CLOSE cursor_name;
```

```
DEALLOCATE cursor_name;
```



Steve Nouri 

@SteveNouri