SQL Cheat Sheet

(Quick Guide To Essential Commands)

SQL Data Types

The data types for specific database management systems (DBMS) can vary (e.g. Microsoft SQL Server vs MySQL). However, there are several that are found on most systems. You can split these into three categories:

- **Numeric**
- **Date And Time**
- **String**

Numeric Types

These are the most common numeric types:

- **INTEGER:** A whole number without a decimal point.
- **SMALLINT:** A smaller range of whole numbers
- **BIGINT:** A larger range of whole numbers.
- **DECIMAL(p, s) or NUMERIC(p, s):** For example, a decimal(5,2) would fit 123.45.
- **REAL:** A floating-point number, with a precision of at least 6 decimal digits.
- **FLOAT(n):** A floating-point number, with a precision of at least n digits.

Date And Time Types

- **DATE:** A date value, typically in the format 'YYYY-MM-DD'.
- **TIME:** A time value, typically in the format 'HH:MM:SS'.
- **DATETIME or TIMESTAMP:** A combination of date and time values.

String Types

- **CHAR(n):** A fixed-length string with n characters.
- **VARCHAR(n) or CHARACTER VARYING(n):** A variable-length string.
- **DATETIME or TIMESTAMP:** A combination of date and time values.

SELECT Statement

The SELECT statement is used to retrieve data from one or more tables. You can specify the columns you want to retrieve and from which table. A basic SELECT statement looks like this:

```
SELECT column1, column2
FROM table;
```

To retrieve all records from the columns 'name' and 'country_id' from the 'city' table, your SQL query looks like this:

```
SELECT name, country_id
FROM city;
```

WHERE Clause

The WHERE clause allows you to filter the results of a SELECT statement based on specific conditions.

```
SELECT column1, column2
FROM table
WHERE condition;
```

To retrieve records from the 'city' table where the 'population' is greater than 1,000,000, your query looks like this:

```
SELECT name, population
FROM city
WHERE population > 1000000;
```

ORDER BY Clause

The ORDER BY clause allows you to sort the results of a SELECT statement by one or more columns. You can sort the results in ascending (ASC) or descending (DESC) order:

```
SELECT column1, column2
FROM table
ORDER BY column1 ASC, column2 DESC;
```

For example, to retrieve records from the 'city' table sorted by 'population' in descending order, your query looks like this:

```
SELECT name, population
FROM city
ORDER BY population DESC;
```

Joining Multiple Tables In SQL

There are four commonly used joins in SQL:

- **INNER JOIN**
- **LEFT JOIN**
- **RIGHT JOIN**
- **FULL JOIN**

INNER JOIN

An INNER JOIN retrieves records that have matching values in both tables.

Let's take an example of a database of artists and albums, and you want to find all artist and album combinations. This is the INNER JOIN:

```
SELECT *
FROM artists AS a
INNER JOIN albums AS b
ON a.artist_id = b.artist_id;
```

With an INNER JOIN, only the rows with matching values in the specified fields will be returned in the results.

LEFT JOIN

A LEFT JOIN is also known as a LEFT OUTER JOIN. It returns all records from the left table and the matched records from the right table. If there is no match in the right table, the result will contain NULL values.

For example, to get a list of all artists and their respective albums (if they have any), you can use a LEFT JOIN:

```
SELECT *
FROM artists AS a
LEFT JOIN albums AS b
ON a.artist_id = b.artist_id;
```

This query will return all artists, even if they don't have any albums associated with them in the albums table.

RIGHT JOIN

A RIGHT JOIN is also known as a RIGHT OUTER JOIN. It returns all records from the right table and the matched records from the left table. If there is no match in the left table, the result will contain NULL values.

For example, to get information about all albums and their associated artists (if they exist), you would use a RIGHT JOIN:

```
SELECT *
FROM artists AS a
RIGHT JOIN albums AS b
ON a.artist_id = b.artist_id;
```

This query will return all albums, even if they don't have associated artists in the artists table.

FULL JOIN

A FULL JOIN is also known as a FULL OUTER JOIN. It combines the results of both LEFT and RIGHT joins. In other words, it returns all rows from the left and right tables and fills in the missing values with NULLs when there is no match.

Here's an example using the artists and albums tables:

```
SELECT *
FROM artists AS a
FULL JOIN albums AS b
ON a.artist_id = b.artist_id;
```

This query returns all rows from both tables, filling in NULLs where there is no match in either table.

SQL Aggregate Functions

Aggregate functions are used to compute a single result from a set of input values. They're called "aggregate" because they take multiple inputs and return a single output. The most common are:

- **COUNT**
- **SUM**
- **AVG**
- **MAX**
- **MIN**

COUNT Function

The COUNT function allows you to count the number of rows in a query result. You can use this aggregate function to determine the total number of records in a table or the number of records that match specific criteria. Here's an example:

```
SELECT COUNT(*) FROM employees;
```

This query will return the total number of employees in the 'employees' table. Keep in mind that adding a WHERE clause can refine your results:

```
SELECT COUNT(*) FROM employees WHERE
department = 'HR';
```

SUM Function

The SUM function calculates the total sum of a numeric column. It's useful when you need to calculate the total value of a particular numeric field. For example, this query returns the total sum of all employee salaries:

```
SELECT SUM(salary) FROM employees;
```

AVG Function

The AVG function computes the average value of a numeric column. This function is helpful when you want to find the average of a particular numeric field. For instance, this query returns the average salary of all employees:

```
SELECT AVG(salary) FROM employees;
```

MIN Function

Lastly, the MIN function helps you find the minimum value of a column. For example, this query returns the lowest salary:

```
SELECT MIN(salary) FROM employees;
```

Remember, you can use WHERE clauses in these queries and JOIN with multiple tables.

SQL Cheat Sheet

(Quick Guide To Essential Commands)

Common String Functions

Here are the most common string functions that are found in most SQL dialects (the exact syntax can vary):

- **LEN or LENGTH(string):** Returns the length of a string.
- **UPPER(string):** Converts a string to upper case.
- **LOWER(string):** Converts a string to lower case.
- **SUBSTR or SUBSTRING(string, start, length):** Extracts a portion from a string.
- **TRIM(string):** Removes leading and trailing spaces from a string.
- **LTRIM(string):** Removes leading spaces from a string.
- **RTRIM(string):** Removes trailing spaces from a string.

Common String Functions

Here are the most common numeric functions that are found in most SQL dialects (the exact syntax can vary):

- **ABS(number):** Returns the absolute value of a number.
- **ROUND(number, decimal_places):** Rounds a number to a certain number of decimal places.
- **FLOOR(number):** Rounds down the number to the nearest integer.
- **CEIL or CEILING(number):** Rounds up the number to the nearest integer.
- **RAND():** Returns a random float value from 0 to 1.
- **MOD(n, m):** Returns the remainder of n divided by m.
- **POWER(base, exponent):** Raises a number to the power of another number.
- **LOG(number):** Returns the natural logarithm of a number.

Common Date Functions

Here are the most common date functions that are found in most SQL dialects (the exact syntax can vary):

- **NOW():** Returns the current date and time.
- **DATE(datetime):** Extracts the date part of a date or datetime expression.
- **TIME(datetime):** Extracts the time part of a date or datetime expression.
- **YEAR(date):** Returns the year part.
- **MONTH(date):** Returns the month part.
- **DAY(date):** Returns the day of the month part.
- **HOUR(time):** Returns the hour part from a time.
- **MINUTE(time):** Returns the minute part from a time.
- **SECOND(time):** Returns the second part from a time.

GROUP BY And HAVING

When working with SQL queries, you may want to further summarize and filter your aggregated data. The GROUP BY and HAVING clauses provide this functionality.

Group By Clause

The GROUP BY clause allows you to group rows that share the same values in specified columns. It is commonly used with aggregate functions. This is the syntax:

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2;
```

For example, if you want to calculate the total sales amount for each product category, this is the query:

```
SELECT product_category, SUM(sales_amount)
FROM sales_data
GROUP BY product_category;
```

TIP: Combining GROUP BY and COUNT is a good way of finding duplicate values.

Having Clause

If you want to filter the aggregated results further, you can use the HAVING clause. The syntax is:

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING condition;
```

If you want to find product categories with total sales of more than $1,000,000, you would write:

```
SELECT product_category, SUM(sales_amount)
FROM sales_data
GROUP BY product_category
HAVING SUM(sales_amount) > 1000000;
```

Quick Tips

- Always use the GROUP BY clause before the HAVING clause.
- The SELECT statement can only contain specified column names, aggregate functions, constants, and expressions.
- When using the HAVING clause, filter conditions should be applied to the aggregate functions rather than directly to the grouped columns.

By understanding and properly applying the GROUP BY and HAVING clauses, you can better organize and analyze your data using SQL.

Subqueries

A subquery is also known as an inner or nested query. This is a query embedded within another SQL statement (such as a SELECT statement) or even inside another subquery.

Subqueries allow you to retrieve data based on the output of another query. The most common operators used with subqueries are:

- **IN**
- **EXISTS**
- **ANY**
- **ALL**

IN Operator

The IN operator tests if a value is within a set of values generated by the inner query. The syntax for using the IN operator with a subquery is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT column_name FROM other_table);
```

This returns rows from the outer query where the specified column value matches any of the values provided by the subquery.

Suppose you have an employee table and a departments table. You want to find employees who work in departments based at the head office. Here is a sample query

```
SELECT first_name, last_name
FROM employee
WHERE department IN (SELECT department FROM departments WHERE location = "HQ");
```

Far a more in-depth look, check out our article on the SQL WHERE IN syntax.

EXISTS Operator

The EXISTS operator checks if there is at least one row resulting from the subquery. You can use the EXISTS operator to filter rows based on the existence of related data in another table. This is the syntax:

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS (SELECT column_name FROM other_table WHERE condition);
```

When the subquery returns at least one row, the EXISTS operator returns true, and the relevant rows from the outer query are included in the result.

ANY Operator

The ANY operator is used to compare a value to any value in a set of values provided by a subquery. It's commonly used with comparison operators like =, <, >, <=, or >=.

This is the syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY (SELECT column_name FROM other_table WHERE condition);
```

This will return rows from the outer query where the specified column value meets the condition against any value from the subquery.

ALL Operator

The ALL operator compares a value to all values within a set of values provided by a subquery. The conditions must be true for every value in the subquery's result. This is the syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL (SELECT column_name FROM other_table WHERE condition);
```

This returns rows from the outer query only if the specified column value satisfies the condition against all values in the subquery's output.

Data Manipulation (DML)

Data Manipulation Language (DML) is a sub-language within SQL for managing and updating data. The most common statements are:

- **INSERT**
- **UPDATE**
- **DELETE**

INSERT Statement

The INSERT statement allows you to insert rows to a table. Here's the basic syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

For example, if you want to insert a new row into a 'users' table with columns 'id', 'name', and 'email', you would use the following query:

```
INSERT INTO users (id, name, email)
VALUES (1, 'John Doe', 'john.doe@example.com');
```

UPDATE Statement

The UPDATE statement allows you to modify existing row data in a table. This is the syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

For example, if you want to update the email address of a user with the id '1' in the 'users' table, your query would look like this:

```
UPDATE users
SET email = 'new.email@example.com'
WHERE id = 1;
```

TIP: remember to include a WHERE clause to avoid updating all rows in the table by mistake.

SQL Cheat Sheet

(Quick Guide To Essential Commands)

DELETE Statement

The DELETE statement allows you to remove rows from a table. Here's the syntax:

```
DELETE FROM table_name
WHERE condition;
```

For example, if you want to delete a user with the id '1' from the 'users' table, your query would look like this:

```
DELETE FROM users
WHERE id = 1;
```

TIP: always include a WHERE clause to specify which rows to delete and to avoid deleting all the rows in the table.

Database Management With DDL

Data Definition Language (DDL) is the SQL sub-language used for creating and altering tables and the database itself. The most common DDL statements are:

- **CREATE**
- **ALTER**
- **DROP**

CREATE Statement

The CREATE statement allows you to create new database objects, such as a new tables, views, or indexes. When creating a new table, you need to define the columns, their data types, and any constraints.

Here's an example of creating an orders table:

```
CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  product VARCHAR(255) NOT NULL,
  customer_id INT NOT NULL
);
```

TIP: choose appropriate data types and constraints to ensure data integrity in your tables.

For a more detailed look, check out our article on basic SQL table operations.

ALTER Statement

The ALTER statement helps you modify existing database objects. Common uses include:

- adding, modifying or dropping columns.
- adding or removing constraints from an existing table.
- adding a primary and foreign keys.

ADD A NEW COLUMN

```
ALTER TABLE users ADD COLUMN age INTEGER;
```

MODIFY A COLUMN'S DATA TYPE

```
ALTER TABLE users ALTER COLUMN age TYPE FLOAT;
```

DROP A COLUMN

```
ALTER TABLE users DROP COLUMN age;
```

ADD A UNIQUE CONSTRAINT

```
ALTER TABLE users ADD CONSTRAINT users_email_unique
UNIQUE(email);
```

ADD A FOREIGN KEY BETWEEN TABLES

```
ALTER TABLE users ADD FOREIGN KEY (country_id)
REFERENCES Country(country_id);
```

DROP Statement

The DROP statement allows you to remove database objects like tables, views, or indexes. Use it with caution, as it will permanently delete the specified object and all its data.

Here's an example:

```
DROP TABLE users;
```

TIP: Ensure you have proper backups in place before executing a DROP statement.

If you want to learn more about data modeling, check out this video:

Youtube Reference

TRANSACTIONS

Transactions play a crucial role in maintaining database integrity, especially when multiple related operations are executed concurrently. There are three fundamental operations in handling transactions:

- **BEGIN**
- **COMMIT**
- **ROLLBACK**

BEGIN

The BEGIN statement signifies the beginning of a transaction. Upon executing this command, you're establishing a starting point for your set of SQL statements.

```
BEGIN;
```

COMMIT

To finalize your changes and persist them in the database, use the COMMIT statement. This ensures that all the operations within the transaction are executed successfully and permanently.

```
COMMIT;
```

Here's an example of a full transaction using the classic example of transferring funds between accounts:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

ROLLBACK

When working with transactions, it's also essential to know how to undo changes when an error occurs. The ROLLBACK statement reverses all the changes made since the start of the transaction:

```
ROLLBACK;
```

Here's an example of a transaction with error handling using ROLLBACK:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
IF @@ERROR <> 0
   ROLLBACK;
ELSE
   COMMIT;
```