SQL: My Notes

Databases and SQL for Data Science with Python

Tristan Daret

November 28, 2025

Table of Contents

SQL Commands Cheatsheet

DQL ⇒ Data Query Language

Command	Applies to	Description
SELECT	Table	Retrieve data from database tables
WHERE	Rows	Filter rows based on conditions
JOIN	Tables	Combine rows from two or more tables
GROUP BY	Rows	Group rows for aggregation
HAVING	Groups	Filter groups after aggregation
ORDER BY	Result set	Sort results (ASC/DESC)
COUNT	Rows	Count number of rows (aggregate)
SUM	Rows	Sum values (aggregate)
AVG	Rows	Average values (aggregate)
MIN	Rows	Minimum value (aggregate)
MAX	Rows	Maximum value (aggregate)
DISTINCT	Rows	Return unique values only
BETWEEN	Rows	Range selection (inclusive)
IN	Rows	Match any value from a list or subquery
LIKE	Rows	Pattern matching with wildcards
REGEXP	Rows	Regex pattern matching (DB-specific)
LIMIT	Result set	Restrict number of rows returned
OFFSET	Result set	Skip a specified number of rows
AS	Columns/Expressions	Give an alias to a column or expression
ILIKE	Rows	Case-insensitive LIKE (PostgreSQL)

SQL Commands Reference - DML

DML $\Rightarrow$ Data Manipulation Language

Command	Applies to	Description
INSERT	Rows	Add new rows to a table
UPDATE	Rows	Modify existing rows
DELETE	Rows	Remove rows from a table
SET	Columns	Specify column values in UPDATE

SQL Commands Reference - DDL

DDL $\Rightarrow$ Data Definition Language

Command	Applies to	Description
CREATE	Table/DB	Create new database objects (tables, etc.)
ALTER	Table	Modify structure of existing objects
ADD	Column	Add new columns to existing table
DROP	Table/DB	Delete database objects permanently
TRUNCATE	Table	Remove all rows from table (keep structure)
RENAME	Table/Column	Rename tables or columns
MODIFY	Column	Change column definition (ALTER variant)
SET CONSTRAINTS	Session	Control timing of constraint checks (IMMEDIATE/DEFERRED)

SQL Commands Reference - DCL

DCL $\Rightarrow$ Data Control Language

GRANT	Permissions	Give privileges to users or roles
REVOKE	Permissions	Remove privileges from users or roles

SQL Commands Reference - Constraints & Options

Command	Applies to	Description
NOT NULL	Column	Column must have a value (no NULL)
PRIMARY KEY	Column	Unique identifier for each row
FOREIGN KEY	Column	Link to PRIMARY KEY in another table
UNIQUE	Column	Enforce unique values in a column
CHECK	Column	Custom condition that values must satisfy
DEFAULT	Column	Default value when none provided

SQL Key concepts

Database keys

Primary Key

- **Uniquely identifies** each row in a table
- Analogy: Like a particle's unique quantum state identifier
- Must be NOT NULL and UNIQUE
- Only ONE primary key per table

Foreign Key

- Column(s) that **reference** a Primary Key in another table
- Analogy: Like conservation laws linking different processes
- Enforces **referential integrity**
- Can have multiple foreign keys in one table
- Prevents orphaned records

Integrity Constraints

Entity Integrity

Purpose: Guarantees each row is uniquely identifiable

- Enforced via PRIMARY KEY constraint
- No NULL values allowed in primary key
- Prevents duplicate entities in table
- Example: Each particle has unique ID

Domain Constraints

Purpose: Restrict values to valid domain/range

- Data type enforcement (INT, VARCHAR, DATE, etc.)
- NOT NULL: requires a value
- CHECK: custom conditions (e.g., mass > 0)
- UNIQUE: no duplicates allowed
- DEFAULT: provides fallback value

Referential Integrity

Purpose: Maintains consistency between related tables

- FOREIGN KEY references PRIMARY KEY in parent table
- Cannot insert orphaned records (child without parent)
- Cascade actions: ON DELETE/UPDATE CASCADE, SET NULL, RESTRICT
- Example: Experiment must reference existing particle

Key points

These three constraint types work together to ensure data quality, consistency, and validity throughout the database.

SQL statement categories

DDL - Data Definition Language

Defines and modifies database **structure**

- CREATE, ALTER, DROP, TRUNCATE
- Schema operations
- Usually **irreversible**

DML - Data Manipulation Language

Manipulates the **data** within structures

- INSERT, UPDATE, DELETE
- Data operations
- Can be rolled back

DQL - Data Query Language

Retrieves data from database

- SELECT (with WHERE, JOIN, etc.)
- Read-only operations
- No data modification

DCL - Data Control Language

Controls access and permissions

- GRANT, REVOKE
- User permissions
- Security management

Data Query Language (DQL)

SELECT - Retrieve Data

Syntax:

```sql
SELECT column1, column2, ...
FROM table_name;

-- All columns
SELECT *
FROM table_name;
```

Example:

```sql
SELECT particle_name, mass
FROM particles;

-- All data
SELECT *
FROM particles;
```

Description: The fundamental query command - retrieves data from one or more tables.

Note: SELECT * can be inefficient for large tables - specify only needed columns.

WHERE - Filter Rows

Syntax:

```sql
SELECT column1, column2
FROM table_name
WHERE condition;

-- Multiple conditions
WHERE condition1
  AND condition2
  OR condition3;
```

Example:

```sql
SELECT particle_name
FROM particles
WHERE mass > 100;

-- Combined
WHERE charge = 0
  AND spin = 0.5;
```

Description: Filters rows based on specified conditions (like applying selection cuts in analysis).

Operators: $=$, $<$, $>$, $<=$, $>=$, $<>$, AND, OR, NOT, IN, LIKE, BETWEEN

Pattern Matching - LIKE and Wildcards

Syntax:

```
-- Basic LIKE patterns
WHERE col LIKE 'abc%'     -- starts with 'abc'
WHERE col LIKE '%xyz'     -- ends with 'xyz'
WHERE col LIKE '%mid%'    -- contains 'mid'
WHERE col LIKE 'h_m'      -- h + any single char + m

-- Escape literal % or _
WHERE col LIKE '50\%%' ESCAPE '\'
```

Wildcards:

- % : any sequence of characters (including empty)
- _ : exactly one character
- Use ESCAPE to treat wildcard characters literally
- ILIKE (PostgreSQL) : case-insensitive LIKE

Examples:

```
SELECT name FROM particles
WHERE name LIKE 'mu%';

SELECT name FROM particles
WHERE name ILIKE '%on%'; -- PostgreSQL
```

Advanced Patterns - Regex and Performance

Regex Syntax (DB-specific):

```
-- MySQL
WHERE col REGEXP '^A[0-9]{3}$'

-- PostgreSQL
WHERE col ~ '^[A-Z]{2}[0-9]+$'
WHERE col ~* 'pattern'  -- case-insensitive
```

Performance Notes:

- Leading % (e.g., '%term') prevents index use
- Prefer 'prefix%' patterns to leverage indexes
- Regex is more flexible but usually slower than LIKE
- For repeated workloads consider indexed/generated columns

Use cases: validation, structured codes, exploratory substring search.

Range Selection - BETWEEN

Syntax:

```sql
-- Inclusive range
WHERE col BETWEEN low AND high

-- Equivalent
WHERE col >= low AND col <=
    high

-- Exclude range
WHERE col NOT BETWEEN low AND
    high
```

Notes:

- BETWEEN is inclusive of both endpoints
- Works for numbers, dates, and strings (DB-specific ordering)
- For exclusive bounds use >/< explicitly

Examples:

```sql
SELECT * FROM experiments
WHERE mass BETWEEN 10 AND 100;

-- Date range
SELECT * FROM experiments
WHERE start_date BETWEEN '2020-01-01' AND '2020-12-31';
```

Set Selection - IN and Subqueries

Syntax:

```sql
-- Match any value in the list
WHERE col IN (val1, val2, ...)

-- Using a subquery
WHERE col IN (SELECT id FROM
    allowed_ids)

-- Negation
WHERE col NOT IN (val1, val2)
```

Notes:

- `IN` is shorthand for multiple ORs
- Subquery must return a single column
- Beware `NULL` with `NOT IN` (can yield no rows) – prefer `NOT EXISTS`

Examples:

```sql
SELECT name FROM particles
WHERE charge IN (-1, 0, 1);

SELECT * FROM results
WHERE experiment_id IN (
  SELECT id FROM experiments WHERE status = 'active'
);
```

COUNT - Count Rows

Syntax:

```sql
SELECT COUNT(column_name)
FROM table_name;

-- Count all rows
SELECT COUNT(*)
FROM table_name;

-- With condition
SELECT COUNT(*)
FROM table_name
WHERE condition;
```

Example:

```sql
SELECT COUNT(particle_id)
FROM particles;

-- All particles
SELECT COUNT(*)
FROM particles;

-- Neutral particles
SELECT COUNT(*)
FROM particles
WHERE charge = 0;
```

Description: Returns the number of rows (like counting events in a detector).
Note: COUNT(*) includes NULL values; COUNT(column) excludes NULLs.

DISTINCT - Unique Values

Syntax:

```sql
SELECT DISTINCT column1
FROM table_name;

-- Multiple columns
SELECT DISTINCT
    column1, column2
FROM table_name;
```

Example:

```sql
SELECT DISTINCT charge
FROM particles;

-- Unique combinations
SELECT DISTINCT
    charge, spin
FROM particles;
```

Description: Returns only unique values, eliminating duplicates.
Note: With multiple columns, DISTINCT applies to the *combination* of values.

GROUP BY - Aggregate Rows

Syntax / Examples:

```sql
-- Basic grouping with one
    aggregate
SELECT grouping_col, COUNT(*) AS
    cnt
FROM table_name
GROUP BY grouping_col;

-- Multiple aggregates
SELECT col1, SUM(amount) AS
    total, AVG(score) AS
    avg_score
FROM table_name
GROUP BY col1;

-- Multiple grouping columns
SELECT col1, col2, SUM(value)
FROM table_name
GROUP BY col1, col2;

-- Advanced: rollup / grouping
    sets (DB-specific)
SELECT col1, col2, SUM(value)
FROM table_name
GROUP BY GROUPING SETS ((col1,
    col2), (col1), ());
```

What GROUP BY does:

- Combines input rows into groups based on the values of the grouping column(s).
- Produces one output row per distinct group; aggregate functions then compute summary values for each group.

Key rules and notes:

- Every output expression must be either a grouping column or an aggregate expression (standard SQL rule).
- Aggregates ignore NULLs (except COUNT(*) which counts rows); NULLs compare equal for grouping purposes (NULLs form their own group).
- `WHERE` filters rows *before* grouping; `HAVING` filters groups *after* aggregation.
- Functional dependencies and some DB extensions may relax the strict "grouping column" rule (DB-specific behavior).
- Use grouping extensions (`ROLLUP`, `CUBE`, `GROUPING SETS`) to produce subtotal/total rows.

ORDER BY - Sort Results

Syntax / Examples:

```sql
-- Simple ordering
SELECT col1, col2
FROM table_name
ORDER BY col1 ASC, col2 DESC;

-- Order aggregated results (use
    alias or expression)
SELECT category, SUM(amount) AS
    total
FROM table_name
GROUP BY category
ORDER BY total DESC;

-- Use NULLS FIRST / LAST (DB-
    specific)
SELECT name FROM table_name
ORDER BY score DESC NULLS LAST;

-- Order by position (ordinal)
    instead of name
SELECT col1, col2 FROM
    table_name
ORDER BY 2, 1;  -- orders by
    col2 then col1
```

Behavior and best practices:

- `ORDER BY` is performed after any `GROUP BY` and after projection of SELECT expressions.
- You can order by column names, aliases, expressions, or column ordinals (position numbers).
- Collation and locale affect sort order; be explicit when needed for reproducible results.
- Large sorts can be expensive; use indexes or limit rows before sorting when possible.
- Use `NULLS FIRST` / `NULLS LAST` when your DB supports them to control NULL ordering.

Ordering within groups (preserve rows + rank):

```sql
-- Use window functions to rank/order
    rows inside partitions
SELECT id, partition_col, value,
       ROW_NUMBER() OVER (PARTITION
           BY partition_col ORDER BY
            value DESC) AS rn
FROM table_name
ORDER BY partition_col, rn;
```

COUNT and Aliases - Label Aggregates

Syntax / Examples:

```sql
-- Count rows per category and
     name the column
SELECT category, COUNT(*) AS
    cnt
FROM items
GROUP BY category;

-- Multiple aggregates
SELECT user_id,
       COUNT(*) AS num_orders,
       SUM(total) AS
           total_spent
FROM orders
GROUP BY user_id;
```

Notes:

- AS gives a readable name to aggregate output columns
- Aggregates ignore NULLs (except COUNT(*))
- Use aliases in ORDER BY or in outer queries for clarity

HAVING - Filter Groups (vs WHERE)

Syntax:

```
SELECT category, COUNT(*) AS
    cnt
FROM items
GROUP BY category
HAVING COUNT(*) > 10;

-- Combining WHERE and HAVING
SELECT user_id, SUM(amount) AS
     total
FROM payments
WHERE status = 'completed'
    -- filters rows first
GROUP BY user_id
    -- then groups the
    filtered rows
HAVING SUM(amount) > 1000;
    -- filters groups by
    aggregate
```

Key Differences:

- WHERE filters raw rows before grouping
- HAVING filters groups after aggregation
- HAVING typically references aggregate functions

LIMIT - Restrict Result Set

Syntax:

```sql
SELECT column1, column2
FROM table_name
LIMIT number;

-- With ORDER BY
SELECT column1
FROM table_name
ORDER BY column1
LIMIT number;
```

Example:

```sql
SELECT particle_name
FROM particles
LIMIT 10;

-- Top 5 heaviest
SELECT particle_name
FROM particles
ORDER BY mass DESC
LIMIT 5;
```

Description: Restricts the number of rows returned (useful for large datasets).
Note: Without ORDER BY, which rows are returned is undefined. Different syntax in some databases (TOP in SQL Server).

OFFSET - Skip Rows

Syntax:

```sql
SELECT column1, column2
FROM table_name
LIMIT number
OFFSET skip_count;

-- Alternative syntax
SELECT column1
FROM table_name
LIMIT skip_count, number;
```

Example:

```sql
SELECT particle_name
FROM particles
LIMIT 10
OFFSET 20;

-- Skip first 20,
-- return next 10
-- (rows 21-30)
```

Description: Skips specified number of rows before returning results (pagination).
Note: Often used with LIMIT for pagination. OFFSET 0 returns from the first row.

Data Manipulation Language (DML)

INSERT - Add New Rows

Syntax:

```
INSERT INTO table_name
    (col1, col2, ...)
VALUES
    (val1, val2, ...);

-- Multiple rows
INSERT INTO table_name
VALUES
    (val1, val2),
    (val3, val4);
```

Example:

```
INSERT INTO particles
    (name, mass, charge)
VALUES
    ('electron', 0.511, -1);

-- Multiple
INSERT INTO particles
VALUES
    ('muon', 105.7, -1),
    ('tau', 1776.9, -1);
```

Description: Adds new rows of data to a table.
Warning: Must satisfy all constraints (PRIMARY KEY, NOT NULL, etc.).

UPDATE - Modify Existing Rows

Syntax:

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

Example:

```
UPDATE particles
SET mass = 0.511
WHERE name = 'electron';

-- Multiple columns
UPDATE particles
SET mass = 105.658,
    spin = 0.5
WHERE name = 'muon';
```

Description: Modifies existing data in a table.
WARNING: Without WHERE clause, **ALL rows will be updated**! Always test with SELECT first.

SET - Assign Values (UPDATE)

Syntax:

```
UPDATE table_name
SET column1 = value1,
    column2 = value2,
    column3 = expression
WHERE condition;
```

Example:

```
UPDATE experiments
SET status = 'complete',
    end_date = CURDATE(),
    duration = 365
WHERE exp_id = 42;
```

Description: SET is not standalone - it's the clause in UPDATE that specifies which columns to modify and their new values.
Note: Can use expressions, functions, or values from other columns.

DELETE - Remove Rows

Syntax:

```sql
DELETE FROM table_name
WHERE condition;

-- Delete all rows
DELETE FROM table_name;
```

Example:

```sql
DELETE FROM particles
WHERE mass < 0.001;

-- Remove all data
-- (structure remains)
DELETE FROM temp_data;
```

Description: Removes rows from a table permanently.
WARNING: Without WHERE, **ALL rows deleted**! Cannot delete if foreign key constraints are violated. Use TRUNCATE for faster full-table deletion.

Data Definition Language (DDL)

CREATE - Build New Objects

Syntax:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    PRIMARY KEY (column1)
);
```

Example:

```
CREATE TABLE particles (
    particle_id INT,
    name VARCHAR(50),
    mass DECIMAL(10,3),
    charge INT,
    PRIMARY KEY (particle_id)
);
```

Description: Creates new database objects (tables, databases, indexes, etc.).
Note: Define structure carefully - changing it later requires ALTER. Common datatypes: INT, VARCHAR, DECIMAL, DATE, BOOLEAN.

ADD - Add Columns to Table

Syntax:

```sql
ALTER TABLE table_name
ADD column_name datatype;

-- With constraints
ALTER TABLE table_name
ADD column_name datatype
    constraint;

-- Multiple columns
ALTER TABLE table_name
ADD column1 datatype1,
ADD column2 datatype2;
```

Example:

```sql
ALTER TABLE particles
ADD spin DECIMAL(3,1);

-- With NOT NULL
ALTER TABLE particles
ADD mass DECIMAL(10,3)
    NOT NULL;

-- Multiple columns
ALTER TABLE particles
ADD color VARCHAR(20),
ADD discovered DATE;
```

Description: Adds new columns to an existing table without affecting existing data.

Note: New column is added with NULL values for existing rows (unless DEFAULT specified). Cannot add NOT NULL without DEFAULT on non-empty tables.

ALTER - Modify Table Structure

Syntax:

```sql
-- Add column
ALTER TABLE table_name
ADD column_name datatype;

-- Drop column
ALTER TABLE table_name
DROP COLUMN column_name;

-- Modify column
ALTER TABLE table_name
MODIFY column_name datatype;
```

Example:

```sql
-- Add spin column
ALTER TABLE particles
ADD spin DECIMAL(3,1);

-- Remove old column
ALTER TABLE particles
DROP COLUMN old_field;

-- Change type
ALTER TABLE particles
MODIFY mass FLOAT;
```

Description: Modifies the structure of an existing table.
Warning: Can cause data loss if not careful (e.g., reducing column size).

DROP - Delete Objects Permanently

Syntax:

```
DROP TABLE table_name;

DROP DATABASE database_name;

-- With safety check
DROP TABLE IF EXISTS
   table_name;
```

Example:

```
DROP TABLE temp_results;

DROP DATABASE test_db;

-- Safe version
DROP TABLE IF EXISTS
   old_particles;
```

Description: Permanently deletes database objects (table, database, index, etc.). Can be used before a fresh CREATE.
WARNING: IRREVERSIBLE! All data AND structure are destroyed. Cannot drop if foreign keys reference it. Use IF EXISTS to avoid errors.

TRUNCATE - Empty Table Fast

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE temp_events;

-- Faster than:
-- DELETE FROM temp_events;
```

Description: Removes ALL rows from a table, but keeps the structure intact. Much faster than DELETE for large tables.
Warning: Cannot be rolled back in most databases. Resets AUTO_INCREMENT counters. May fail if foreign key constraints exist.

RENAME - Rename Database Objects

Syntax:

```sql
-- Rename table
ALTER TABLE old_name
RENAME TO new_name;

-- Or (MySQL/MariaDB)
RENAME TABLE old_name
   TO new_name;

-- Rename column (MySQL)
ALTER TABLE table_name
RENAME COLUMN old_col
   TO new_col;
```

Example:

```sql
-- Rename table
ALTER TABLE temp_particles
RENAME TO particles_backup;

-- MySQL shorthand
RENAME TABLE old_data
   TO archive_data;

-- Rename column
ALTER TABLE particles
RENAME COLUMN mass
   TO particle_mass;
```

Description: Changes the name of database objects (tables, columns) without affecting data or structure.

Note: Syntax varies by database system. Update all references (views, stored procedures, application code) after renaming. Foreign keys typically remain valid.

MODIFY - Change Column Definition

Syntax:

```
-- MySQL/Oracle
ALTER TABLE table_name
MODIFY column_name
    new_datatype;

-- PostgreSQL/SQL Server
ALTER TABLE table_name
ALTER COLUMN column_name
    TYPE new_datatype;
```

Example:

```
-- MySQL
ALTER TABLE particles
MODIFY mass
    DECIMAL(15,5);

-- PostgreSQL
ALTER TABLE particles
ALTER COLUMN mass
    TYPE DOUBLE PRECISION;
```

Description: Part of ALTER TABLE - changes a column's datatype or constraints.

Note: Syntax varies by database system. May fail if existing data incompatible with new type.

IMMEDIATE - Constraint Checking Mode

Syntax:

```
-- DB2/Some systems
SET CONSTRAINTS ALL
    IMMEDIATE;

-- vs DEFERRED
SET CONSTRAINTS ALL
    DEFERRED;
```

Example:

```
SET CONSTRAINTS ALL
    IMMEDIATE;

INSERT INTO particles
VALUES (1, 'test', 0, 0);
-- Constraints checked
-- immediately
```

Description: Controls when constraint checking occurs. IMMEDIATE = check after each statement; DEFERRED = check at transaction end.
Note: Not supported in all database systems (mainly DB2, Oracle). Useful for complex multi-table operations.

Constraints & Optional Parameters

NOT NULL - Require Values

Syntax:

```sql
CREATE TABLE table_name (
    column1 datatype NOT NULL,
    column2 datatype,
    column3 datatype NOT NULL
);

-- Add to existing
ALTER TABLE table_name
MODIFY column_name
    datatype NOT NULL;
```

Example:

```sql
CREATE TABLE particles (
    id INT NOT NULL,
    name VARCHAR(50) NOT NULL,
    mass DECIMAL(10,3),
    charge INT NOT NULL
);

-- Make mandatory
ALTER TABLE particles
MODIFY name
    VARCHAR(50) NOT NULL;
```

Description: Constraint that prevents NULL values in a column - the column must have a value.

Note: Primary keys are automatically NOT NULL. Essential for critical fields like identifiers.