

# SQL: My Notes

Databases and SQL for Data Science with Python

Tristan Daret

November 25, 2025

# Table of Contents

- 1 SQL Commands Cheatsheet
- 2 SQL Key concepts
- 3 Data Query Language (DQL)
- 4 Data Manipulation Language (DML)
- 5 Data Definition Language (DDL)
- 6 Constraints & Optional Parameters

# SQL Commands Cheatsheet

# SQL Commands Reference - Part 1

Command	Applies to	Description
<b>Data Query Language (DQL)</b>		
SELECT	Table	Retrieve data from database tables
WHERE	Rows	Filter rows based on conditions
COUNT	Rows	Count number of rows
DISTINCT	Rows	Return unique values only
LIMIT	Result set	Restrict number of rows returned
OFFSET	Result set	Skip a specified number of rows
<b>Data Manipulation Language (DML)</b>		
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 - Part 2

Command	Applies to	Description
<b>Data Definition Language (DDL)</b>		
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)
<b>Constraints &amp; Options</b>		
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

# SQL Key concepts

## 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:

```
SELECT column1, column2, ...  
FROM table_name;  
  
-- All columns  
SELECT *  
FROM table_name;
```

## Example:

```
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:

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

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

## Example:

```
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., ')
- Prefer 'prefix
- 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

## extbfSyntax:

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

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

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

## extbfNotes:

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

## extbfExamples:

```
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

## extbfSyntax:

```
-- 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)
```

## extbfNotes:

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

## extbfExamples:

```
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:

```
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:

```
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:

```
SELECT DISTINCT column1
FROM table_name;

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

## Example:

```
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.

# LIMIT - Restrict Result Set

## Syntax:

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

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

## Example:

```
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:

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

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

## Example:

```
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:

```
DELETE FROM table_name
WHERE condition;

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

## Example:

```
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:

```
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:

```
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:

```
-- 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:

```
-- 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:

```
-- 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:

```
-- 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:

```
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:

```
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.