

# SQL: My Notes

Databases and SQL for Data Science with Python

Tristan Daret

November 28, 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 - DQL

## 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** ⇒ 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** ⇒ 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** ⇒ Data Control Language

<b>Command</b>	<b>Applies to</b>	<b>Description</b>
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

## 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.

# 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.

# Aliases - Rename Columns/Expressions

## Syntax:

```
-- Column alias (AS optional)
SELECT column AS alias_name
FROM table_name;

-- Table alias (shorten long
-- names)
SELECT t.column
FROM very_long_table_name AS t;

-- Alias for derived table (
-- required)
SELECT d.*
FROM (
    SELECT id, SUM(value) AS
        total
    FROM measurements
    GROUP BY id
) AS d;
```

## Examples:

```
-- Rename expression
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM people;

-- Use alias in ORDER BY /
-- GROUP BY
SELECT product, SUM(qty) AS
    total_qty
FROM sales
GROUP BY product
ORDER BY total_qty DESC;
```

## Notes:

- AS is optional in many DBs:  
"SELECT col alias" works.
- Use quotes/backticks for aliases with spaces or special chars.

- Table aliases are mandatory for derived tables and helpful to disambiguate columns in joins.
- Aliases are for the query result only – they don't rename underlying schema objects.

# FROM and JOINS - Table Sources

## Basic syntax / examples:

```
-- Simple source
FROM table_a

-- INNER JOIN (only matching rows)
FROM a
INNER JOIN b
    ON a.id = b.a_id

-- LEFT / RIGHT / FULL outer joins
FROM a
LEFT JOIN b ON a.id = b.a_id

-- CROSS JOIN (cartesian product)
FROM a
CROSS JOIN b

-- USING (shorthand when column names
-- match)
FROM a
JOIN b USING (id)

-- Derived table (see aliases slide)
FROM (
    SELECT id, SUM(x) AS total
    FROM measurements
    GROUP BY id
) AS d
```

## Notes and semantics:

- INNER JOIN: returns rows present in both tables (matches on ON/USING).
- LEFT JOIN: all rows from left table; NULLs for non-matching right-side columns.
- RIGHT / FULL JOIN: symmetric variants (FULL returns rows from either side).
- CROSS JOIN: multiplies rows (use with care – can blow up result size).
- ON vs USING:
  - ON allows arbitrary join conditions ( $a.col = b.other\_col$ , complex expressions).
  - USING(column) is concise when join column names are identical; it coalesces the column in output.
- Derived tables require an alias (see the aliases slide). Useful for pre-aggregating or hiding complexity.
- Beware implicit joins (comma-separated FROM): they produce a cartesian product unless paired with WHERE conditions.
- Performance: push filters into the earliest source possible (WHERE / ON) and prefer indexed join 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

# Set Selection - IN and Subqueries

## Syntax:

```
-- 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:

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

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

# Range Selection - BETWEEN

## Syntax:

```
-- 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:

```
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';
```

# NULLs and EXISTS

## Syntax / Examples:

```
-- NULL checks
WHERE col IS NULL;
WHERE col IS NOT NULL;

-- NULL-safe equality (
    PostgreSQL)
WHERE col IS NOT DISTINCT FROM ,
    value';

-- Coalesce to provide defaults
WHERE COALESCE(col, 0) = 0;

-- EXISTS: true if subquery
    returns any row
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.id
);

-- Prefer NOT EXISTS over NOT IN
    when NULLs possible
WHERE NOT EXISTS (
    SELECT 1 FROM parents p
```

## Notes and pitfalls:

- Comparisons with NULL (e.g., col = NULL) yield UNKNOWN and are filtered out; use IS NULL / IS NOT NULL.
- NOT IN with a subquery can return no rows if the subquery yields any NULLs: WHERE x NOT IN (SELECT val FROM t) – dangerous if val contains NULL
- NOT EXISTS is NULL-safe and recommended for exclusion by subquery.
- Use COALESCE to substitute a default for NULL when appropriate (but be careful about semantics).
- NULLs form their own group in GROUP BY and are ignored by many aggregates (except

# 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.

# String functions - Overview

- **Purpose:** manipulate, inspect and transform text values – normalize, extract, format, and validate strings.
- **Common categories:** case conversion, extraction (substring/position), modification (replace(concat), trimming, and searching/matching.
- **Behavior notes:** function names and exact behavior vary by DB; prefer standard names (e.g. LOWER/UPPER) for portability. Collation/locale affects comparisons and case rules.
- **NULLs:** many string functions return NULL when input is NULL – handle explicitly with COALESCE when needed.
- **Performance:** applying functions to table columns may prevent index use; consider functional/indexed expressions or storing normalized values for large-scale queries.
- **Use-cases:** case-insensitive comparisons, parsing structured identifiers, cleaning user input, and preparing display-friendly text.

# Case conversion - LCASE / UCASE

## Syntax / Examples:

```
-- MySQL
SELECT LCASE(name), UCASE(name)
    FROM people;

-- Standard SQL / PostgreSQL
SELECT LOWER(name), UPPER(name)
    FROM people;

-- Use in WHERE
SELECT * FROM people
WHERE LOWER(name) = 'alice';
```

## Notes:

- LCASE/UCASE are MySQL aliases; standard functions are LOWER/UPPER.
- Useful to normalize case for comparisons or grouping.
- Applying functions to columns can prevent index usage – consider functional/indexed expressions when available.

# Modification - CONCAT / REPLACE / TRIM

## Syntax / Examples:

```
-- Concatenate (portable / MySQL)
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM people;
-- Alternatively (SQL standard / some DBs): first_name
-- || ' ' || last_name

-- Replace substrings
SELECT REPLACE(notes, 'foo', 'bar') FROM logs;

-- Trim whitespace
SELECT TRIM(name) FROM people;
-- LTRIM / RTRIM
available
```

## Notes:

- Concatenation operator/name varies by DB (use CONCAT for portability).
- Use TRIM/LTRIM/RTRIM to clean input before comparisons.
- REPLACE is literal-substring based; use regex functions for complex transformations.

# Locate / Search - POSITION / INSTR / CHARINDEX

## Syntax / Examples:

```
-- Standard SQL
SELECT POSITION('sub' IN col)
FROM t;

-- MySQL
SELECT INSTR(col, 'sub') FROM
t;

-- SQL Server
SELECT CHARINDEX('sub', col)
FROM t;
```

## Notes:

- Return semantics (0 vs 1-based) differ by DB—check docs.
- Combine with SUBSTRING to extract surrounding text.
- These functions are cheap per-row but can be costly at scale if used without supporting indexes or precomputed columns.

## Aggregations ⇒ Summarize multiple rows into single values

- Aggregate functions compute a single result from a set of rows (overall or per-group).
- Typically used with GROUP BY to produce summaries per category, or without GROUP BY to summarize the whole table.
- Common aggregates: **COUNT**, **MIN**, **MAX**, **AVG**, **SUM**.
- Notes: aggregates generally ignore NULLs (**COUNT(\*)** is the exception); use WHERE to filter rows before aggregation and HAVING to filter groups after aggregation.

# 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.

# MIN / MAX - Extremes

## Syntax / Examples:

```
-- Smallest / largest single value
SELECT MIN(value_col) FROM
    table_name;
SELECT MAX(value_col) FROM
    table_name;

-- Per-group extremes
SELECT category,
    MIN(price) AS min_price,
    MAX(price) AS max_price
FROM products
GROUP BY category;
```

## Notes:

- MIN returns the smallest non-NULL value; MAX returns the largest non-NULL value.
- Applicable to numbers, dates, and strings (string order uses collation).
- Use with GROUP BY to obtain extremes per group.
- Leading indexes can help speed MIN/MAX queries when filtering by range.

# SUM - Total Values

## Syntax / Examples:

```
-- Sum a column
SELECT SUM(amount) FROM
    transactions;

-- Sum per category
SELECT category, SUM(quantity)
    AS total_qty
FROM sales
GROUP BY category;

-- Treat NULL as zero
SELECT SUM(COALESCE(quantity,0))
    FROM sales;
```

## Notes:

- SUM ignores NULLs by default; use COALESCE to include NULLs as zeros.
- Intended for numeric types; cast non-numeric values before summing.
- Watch for integer overflow on very large totals—use an appropriate numeric type.
- Combine with GROUP BY and HAVING to filter aggregated totals.

# AVG - Averages

## Syntax / Examples:

```
-- Average of a column
SELECT AVG(score) FROM results;

-- Average per group
SELECT team, AVG(points) AS
    avg_points
FROM matches
GROUP BY team;

-- Weighted average (explicit)
SELECT SUM(value * weight) / SUM
    (weight) AS weighted_avg
FROM measurements;
```

## Notes:

- AVG returns the mean of non-NULL values; result is typically a fractional type.
- For precise control, cast operands (e.g., to DECIMAL or DOUBLE).
- To compute weighted averages, use `SUM(...) / SUM(...)`.
- Use `ROUND()` if you need a fixed number of decimal places.

# GROUP BY - Aggregate Rows

## Syntax / Examples:

```
-- 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.

# 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

# ORDER BY - Sort Results

## Syntax / Examples:

-- 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):

-- 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;
```

# LIMIT / TOP / FETCH - Restrict Result Set

## Syntax:

```
-- MySQL / SQLite / PostgreSQL  
-- (LIMIT)  
SELECT column1, column2  
FROM table_name  
LIMIT number;  
  
-- SQL Server (TOP)  
SELECT TOP (number) column1,  
    column2  
FROM table_name;  
  
-- SQL Standard / SQL Server (  
    OFFSET ... FETCH)  
SELECT column1, column2  
FROM table_name  
ORDER BY some_col  
OFFSET skip_count ROWS  
FETCH NEXT number ROWS ONLY;  
  
-- SQL standard shorthand  
SELECT column1, column2  
FROM table_name  
FETCH FIRST number ROWS ONLY.
```

## Notes on semantics:

- Use ORDER BY for deterministic results; otherwise which rows are returned is undefined.
- TOP is a SQL Server (and some DB) extension; LIMIT and FETCH are more portable (FETCH is part of the SQL standard).
- SQL Server also supports "TOP (n) PERCENT" and "WITH TIES" to include tied rows.
- OFFSET ... FETCH provides pagination when combined with ORDER BY.

# 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.

# Set operations – UNION / UNION ALL / INTERSECT / EXCEPT

## Syntax / Examples:

```
-- UNION: combine, remove duplicates
SELECT id, name FROM table_a
UNION
SELECT id, name FROM table_b;

-- UNION ALL: combine, keep duplicates (faster)
SELECT id, name FROM table_a
UNION ALL
SELECT id, name FROM table_b;

-- INTERSECT: rows present in both
SELECT id, name FROM table_a
INTERSECT
SELECT id, name FROM table_b;

-- EXCEPT (or MINUS): rows in left not in right
SELECT id, name FROM table_a
EXCEPT
```

## Behavior, ORDER BY and Notes:

- ORDER BY applies to the final combined result. Put ORDER BY after the last query.
- To order individual subqueries, wrap them as derived tables before combining.
- UNION performs duplicate-elimination (like DISTINCT) – requires sorting or hashing.
- UNION ALL avoids deduplication and is much faster for simple concatenation.
- INTERSECT / EXCEPT also usually require sort/hash; performance depends on indexes and cardinalities.
- Alternatives: use JOIN/EXISTS/NOT EXISTS for finer control or sometimes better performance (especially for anti-joins).
- Be explicit about intent: prefer UNION ALL when duplicates are acceptable or known absent.

# DQL Best Practices – Performance & Portability

- Project only the columns you need; transferring extra fields wastes I/O and memory.
- Avoid wrapping indexed columns in expressions (e.g. case conversion or functions) – this prevents index use. Create functional indexes or store normalized values when appropriate.
- Make pagination deterministic by always specifying an explicit sort key; rely on stable ordering for consistent page results.
- Be explicit about NULL semantics: test for presence/absence of values using dedicated checks and prefer existence-based exclusion patterns when subqueries may yield unknowns.
- Push selective filters as early as possible so the planner can reduce data volume before expensive operations (joins, sorts, aggregates).
- Use index-friendly patterns for string searches; leading wildcards or full regex scans can bypass indexes and degrade throughput.
- Prefer set-aware operations over row-by-row loops; leverage built-in aggregation and windowing to let the engine optimize work.
- Test and tune with the database's planning/analysis tools and representative data; measure cost and cardinality rather than guessing.
- Remember portability differences (pagination, pattern-matching, case-insensitive operators, regex flavors). When writing cross-engine SQL, favor standard constructs or isolate engine-specific clauses behind adapters.
- Use parameterized queries from application code to avoid parsing/plan instability and to improve security.

# 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.