SQL

Version SQL:2023 · verified 2026-07-26

A gradual, compact walk through core SQL — querying, joins, aggregation, and transactions — with the gotchas that differ across engines.

Contents

Mental model

SQL is declarative — you describe the result set you want, not the steps to produce it; the engine's query planner decides how. The order you type clauses isn't the order they run: `FROM` is evaluated first, then `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, and finally `ORDER BY`/`LIMIT` — which is why a `SELECT` alias can't be used in that same query's `WHERE`.

Querying: SELECT, WHERE, ORDER BY, LIMIT

Clause	Purpose
`SELECT col, col2`	Choose columns
`WHERE cond`	Filter rows
`ORDER BY col DESC`	Sort results
`LIMIT n`	Cap the row count

Clauses combine in a fixed shape: pick columns, filter rows, sort, then cap the count. Comparisons in `WHERE` (`=`, `<>`, `BETWEEN`, `IN`, `LIKE`) read close to plain English.

```
SELECT name, price
FROM products
WHERE price > 10
ORDER BY price DESC
LIMIT 5;
```

> **Gotcha:** `LIMIT` is PostgreSQL/MySQL/SQLite syntax, not the SQL standard. SQL Server uses `TOP n`; the portable standard form is `OFFSET m ROWS FETCH NEXT n ROWS ONLY`.

Modifying data

Statement	Effect
`INSERT INTO t (...) VALUES (...)`	Adds a row
`UPDATE t SET col = v WHERE ...`	Changes matching rows
`DELETE FROM t WHERE ...`	Removes matching rows

Each statement targets rows matched by an optional `WHERE` — omit it and the statement applies to every row in the table.

```sql
INSERT INTO products (name, price)
VALUES ('Widget', 9.99);

UPDATE products
SET price = 12.99
WHERE name = 'Widget';
```

> **Gotcha:** `UPDATE` / `DELETE` without a `WHERE` clause touches every row in the table. Run the equivalent `SELECT` first to confirm exactly which rows will change.

Joins

Join	Keeps
`INNER JOIN`	Only matching rows
`LEFT JOIN`	All of the left, matched or not
`FULL JOIN`	All rows from both sides

A join combines rows from two tables using a condition in `ON`. `INNER JOIN` drops non-matches; `LEFT JOIN` keeps every left-table row, filling unmatched columns with `NULL`.

```
SELECT o.id, c.name
FROM orders o
LEFT JOIN customers c
  ON o.customer_id = c.id;
```

> **Gotcha:** putting the join condition in `WHERE` instead of `ON` silently turns a `LEFT JOIN` into an `INNER JOIN` — `WHERE` runs after the join and drops the `NULL`-filled rows the `LEFT JOIN` was there to keep.

Aggregation & GROUP BY

Syntax	Meaning
`COUNT/SUM/AVG(col)`	One value per group
`GROUP BY col`	Defines the groups
`HAVING cond`	Filters groups, not rows

Aggregate functions collapse many rows into one per group. `WHERE` filters rows before grouping; an aggregate result can only be filtered with `HAVING`, evaluated after.

```
SELECT customer_id, COUNT(*) AS orders
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

> **Gotcha:** standard SQL requires every non-aggregated `SELECT` column to also appear in `GROUP BY`. Some engines relax this, but relying on it makes the omitted column's value arbitrary per group.

Subqueries & CTEs

Form	Where it lives
`(SELECT ...)` in `WHERE`	A value or list to compare against
`WITH x AS (SELECT ...)`	A named, reusable subquery

A CTE (`WITH`) names a subquery so the main query reads top to bottom instead of nesting. A correlated subquery references a column from the outer query and re-runs once per outer row.

```
WITH big_orders AS (
  SELECT customer_id FROM orders
  WHERE total > 100
```

```
)
SELECT * FROM customers
WHERE id IN (SELECT customer_id
             FROM big_orders);
```

> **Gotcha:** a correlated subquery runs once per outer row, so on a large table it can be far slower than an equivalent `JOIN` — check `EXPLAIN` if one feels slow.

Window functions

Syntax	Meaning
`ROW_NUMBER() OVER (...)`	Sequential number per row
`PARTITION BY col`	Restarts the window per group
`RANK()` vs `DENSE_RANK()`	Gaps after ties, or none

A window function computes across a set of related rows without collapsing them into one, unlike `GROUP BY`. `ORDER BY` inside `OVER()` sets the row order it runs in.

```
SELECT name, dept,
  RANK() OVER (
    PARTITION BY dept
    ORDER BY salary DESC
  ) AS dept_rank
FROM employees;
```

> **Gotcha:** `RANK()` leaves gaps after a tie (1, 1, 3); `DENSE_RANK()` doesn't (1, 1, 2). Picking the wrong one silently skips or duplicates a rank position downstream.

Transactions & constraints

Syntax	Meaning
`BEGIN` / `COMMIT`	Start / save a transaction
`ROLLBACK`	Undo the open transaction
`FOREIGN KEY ... REFERENCES`	Row must exist elsewhere
`ON DELETE CASCADE`	Deletes dependents automatically

A transaction groups statements so they all succeed or all roll back together. A foreign key blocks inserting a row that points nowhere — and by default also blocks deleting the row it points to.

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

> **Gotcha:** the default `FOREIGN KEY` behavior (`NO ACTION`) blocks deleting a referenced row outright. Add `ON DELETE CASCADE` or `SET NULL` explicitly — don't assume deletion just works.

Further reading

- PostgreSQL Documentation — SELECT
- PostgreSQL Documentation — Window Functions
- PostgreSQL Documentation — Constraints