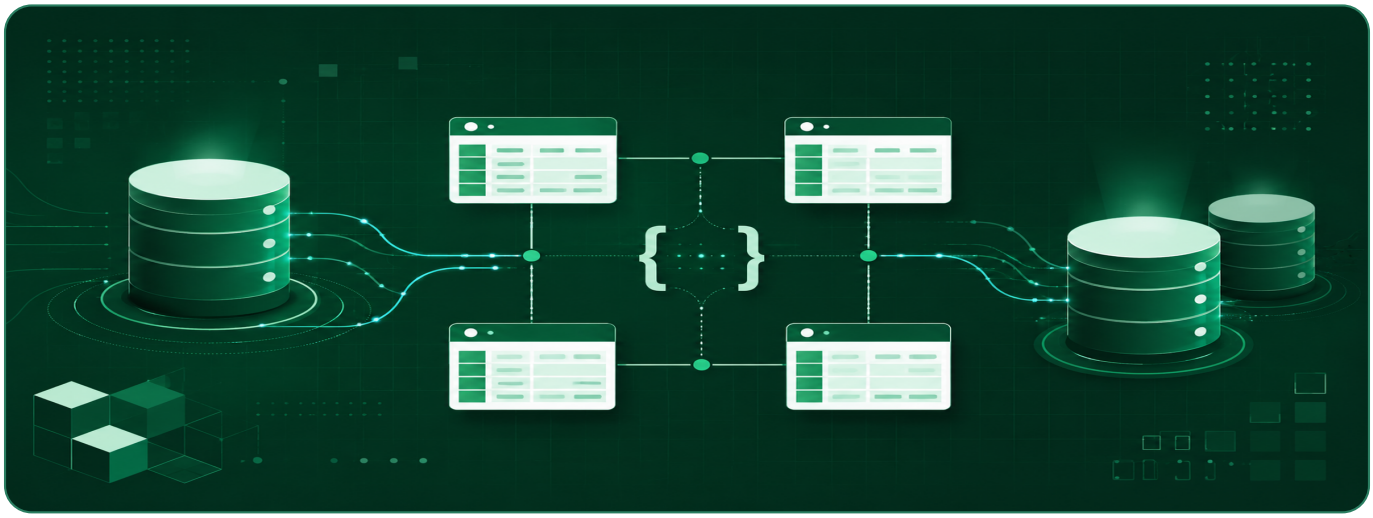
DATABASE QUERY REFERENCE

# SQL Cheat Sheet

A printable guide to queries, JOINs, filtering, grouping, functions, window analysis, transactions, and safer database workflows.



SQL CORE + ANALYSIS | PRINTABLE A4 | VERSION 1.0 - SEPTEMBER 2026

Free interactive guide: cheatsheetsilo.com/sql-cheat-sheet/

01 - BUILD A QUERY

# SELECT, clauses, and execution order

Start with the core SELECT pattern, then filter, sort, limit, and reason about the logical order in which clauses are evaluated.

01 - CORE PATTERN

## Anatomy of a SELECT query

SELECT PATTERN

```
SELECT DISTINCT column_a, aggregate(column_b) AS alias
FROM table_name AS t
JOIN other_table AS o ON o.id = t.other_id
WHERE condition
GROUP BY column_a
HAVING aggregate(column_b) > 100
ORDER BY alias DESC
LIMIT 10;
```

02 - LOGICAL ORDER

## How the database evaluates the query

| STEP | CLAUSE | PURPOSE |
|---|---|---|
| 1 | FROM + JOIN | Build the input row set |
| 2 | WHERE | Remove rows before grouping |
| 3 | GROUP BY | Create groups |
| 4 | HAVING | Remove groups |
| 5 | SELECT + DISTINCT | Calculate and shape output |
| 6 | ORDER BY + LIMIT | Sort and return final rows |

03 - COMMON CLAUSES

## Fast reference

| CLAUSE | USE | EXAMPLE |
|---|---|---|
| DISTINCT | Remove duplicate output rows | `SELECT DISTINCT country FROM customers;` |
| AS | Create a readable alias | `SELECT total AS revenue FROM orders;` |
| ORDER BY | Sort ascending or descending | `ORDER BY created_at DESC;` |
| LIMIT/OFFSET | Return a result slice | `LIMIT 25 OFFSET 50;` |

**DIALECT** LIMIT is common in PostgreSQL, MySQL, and SQLite. SQL Server commonly uses TOP or OFFSET ... FETCH. Check your database dialect.

02 - FILTER PRECISELY

# Operators, NULL, and conditional logic

Express conditions clearly, group mixed AND/OR logic with parentheses, and treat NULL as an unknown value rather than ordinary data.

04 - COMPARISON

## Operators you use every day

| OPERATOR | MEANING | EXAMPLE |
|---|---|---|
| =, <>, != | Equal / not equal | `status <> 'Closed'` |
| <, <=, >, >= | Numeric or date comparison | `total >= 1000` |
| BETWEEN | Inclusive range | `created_at BETWEEN '2026-01-01' AND '2026-12-31'` |
| IN | Match a list | `country IN ('SE','NO','DK')` |
| LIKE | Pattern match | `email LIKE '%@example.com'` |
| IS NULL | Test missing values | `shipped_at IS NULL` |

05 - BOOLEAN LOGIC

## Combine conditions safely

| KEYWORD | MEANING | EXAMPLE |
|---|---|---|
| AND | Every condition must be true | `active = TRUE AND total > 500` |
| OR | At least one condition is true | `region = 'East' OR region = 'West'` |
| NOT | Reverse a condition | `NOT status = 'Cancelled'` |
| Parentheses | Control mixed logic | `active = TRUE AND (tier = 'A' OR tier = 'B')` |

06 - NULL RULES

## Avoid three-valued-logic surprises

CORRECT NULL CHECKS

```
WHERE manager_id IS NULL
WHERE shipped_at IS NOT NULL
```

FALLBACK VALUES

```
SELECT COALESCE(phone, 'Unknown')
FROM customers;
```

**RULE** Never write column = NULL. Comparisons with NULL evaluate to unknown. Use IS NULL, IS NOT NULL, or COALESCE when you need a fallback.

03 - CONNECT TABLES

# JOINs and set operations

Choose a JOIN based on which unmatched rows must survive, and verify that join keys have the expected uniqueness.

07 - JOIN TYPES

## Control which rows are retained

| JOIN | RETURNS | TYPICAL USE |
|---|---|---|
| INNER JOIN | Only matches on both sides | Customers that have orders |
| LEFT JOIN | Every left row plus matches | All customers, including no-order customers |
| RIGHT JOIN | Every right row plus matches | All right-side records; less portable in practice |
| FULL OUTER JOIN | All rows from both sides | Reconcile two systems |
| CROSS JOIN | Every possible pair | Generate combinations |
| SELF JOIN | A table joined to itself | Employee and manager hierarchy |

08 - JOIN PATTERN

## Keep keys explicit

LEFT JOIN

```
SELECT c.customer_id, c.name, o.order_id, o.total
FROM customers AS c
LEFT JOIN orders AS o
  ON o.customer_id = c.customer_id
 AND o.status = 'Paid';
```

09 - SET OPERATIONS

## Stack compatible result sets

| OPERATOR | RESULT |
|---|---|
| UNION | Combine and remove duplicates |
| UNION ALL | Combine and retain duplicates |
| INTERSECT | Rows present in both results |
| EXCEPT | Rows in first result but not second |

**DEBUG** Unexpected duplicates usually mean the join key is not unique. Inspect each table's key cardinality before adding DISTINCT, which may hide the real issue.

04 - SUMMARIZE DATA

# Aggregation, grouping, and functions

Reduce rows into useful metrics, filter groups with HAVING, and use portable functions where possible.

10 - AGGREGATES

## Turn rows into metrics

| FUNCTION | RESULT | EXAMPLE |
|---|---|---|
| COUNT(*) | Count rows | `COUNT(*) AS order_count` |
| COUNT(col) | Count non-NULL values | `COUNT(shipped_at)` |
| SUM | Add numeric values | `SUM(total) AS revenue` |
| AVG | Calculate the mean | `AVG(total) AS avg_order` |
| MIN / MAX | Find extremes | `MAX(created_at) AS latest_order` |

11 - GROUPING

## WHERE filters rows; HAVING filters groups

REVENUE BY REGION

```
SELECT region, COUNT(*) AS orders, SUM(total) AS revenue
FROM orders
WHERE status = 'Paid'
GROUP BY region
HAVING SUM(total) >= 10000
ORDER BY revenue DESC;
```

12 - COMMON FUNCTIONS

## Expect small dialect differences

| AREA | FUNCTIONS | EXAMPLE |
|---|---|---|
| Text | UPPER, LOWER, TRIM, LENGTH, CONCAT | `TRIM(customer_name)` |
| Numbers | ROUND, ABS, CEIL, FLOOR | `ROUND(total, 2)` |
| Dates | CURRENT_DATE, EXTRACT, DATE_TRUNC* | `EXTRACT(YEAR FROM created_at)` |
| Conditional | CASE, COALESCE, NULLIF | `COALESCE(discount, 0)` |

**DIALECT** DATE_TRUNC is common in PostgreSQL but not universal. Date arithmetic, concatenation, booleans, and identifier quoting vary across engines.

05 - COMPOSE ANALYSIS

# Subqueries, CTEs, and window functions

Break complex logic into named steps and calculate rankings or running totals without collapsing detail rows.

13 - CTE

## Name a reusable query step

COMMON TABLE EXPRESSION

```
WITH paid_orders AS (
  SELECT * FROM orders
  WHERE status = 'Paid'
)
SELECT customer_id, SUM(total)
FROM paid_orders
GROUP BY customer_id;
```

CORRELATED EXISTENCE TEST

```
SELECT c.name
FROM customers c
WHERE EXISTS (
  SELECT 1 FROM orders o
  WHERE o.customer_id = c.id
);
```

14 - WINDOW FUNCTIONS

## Analyze rows while preserving row detail

| PATTERN | USE | EXAMPLE |
| --- | --- | --- |
| ROW_NUMBER | Unique sequence within a partition | `ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY created_at DESC)` |
| RANK | Ranking with gaps for ties | `RANK() OVER (ORDER BY total DESC)` |
| LAG / LEAD | Previous or next row value | `LAG(total) OVER (PARTITION BY customer_id ORDER BY created_at)` |
| Running total | Cumulative metric | `SUM(total) OVER (PARTITION BY customer_id ORDER BY created_at)` |

15 - CASE

## Create conditional output

SEGMENT ORDERS

```
SELECT order_id, total,
  CASE WHEN total >= 1000 THEN 'Large'
       WHEN total >= 250 THEN 'Medium'
       ELSE 'Small' END AS order_size
FROM orders;
```

**PERFORMANCE** Use a CTE for readability, not as an automatic performance fix. Modern optimizers may inline it, but behavior depends on the database and version.

06 - CHANGE DATA SAFELY

# Writes, schema, transactions, and performance

Preview affected rows, protect multi-step changes with transactions, and verify performance with the database's query plan.

16 - DATA + SCHEMA

## Essential statements

| STATEMENT | PURPOSE | EXAMPLE |
|---|---|---|
| INSERT | Add rows | `INSERT INTO users (name,email) VALUES ('Ava','ava@example.com');` |
| UPDATE | Modify matching rows | `UPDATE users SET active = FALSE WHERE last_login < '2025-01-01';` |
| DELETE | Remove matching rows | `DELETE FROM sessions WHERE expires_at < CURRENT_DATE;` |
| CREATE TABLE | Define a table | `CREATE TABLE teams (id INT PRIMARY KEY, name VARCHAR(100));` |
| ALTER TABLE | Change a table definition | `ALTER TABLE teams ADD COLUMN active BOOLEAN;` |
| CREATE INDEX | Speed selected access paths | `CREATE INDEX idx_orders_customer ON orders(customer_id);` |

17 - TRANSACTIONS

## Keep related writes together

TRANSACTION PATTERN

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

SAFE WRITE PREVIEW

```
SELECT * FROM users
WHERE last_login < '2025-01-01';

-- Verify first, then UPDATE
-- with the same WHERE clause.
```

18 - QUICK SAFETY CHECK

## Before running production SQL

| CHECK | ACTION |
|---|---|
| Scope | Does the WHERE clause select exactly the intended rows? |
| Backup | Can the change be reversed or restored? |
| Transaction | Can related statements commit or roll back together? |
| Constraints | Will keys, foreign keys, and checks remain valid? |

### Keep the full SQL reference open

Scan for the searchable, always-updated guide, examples, and related resources.

**FREE TO USE | NO SIGN-UP REQUIRED**