1. Quick overview

PostgreSQL is an open-source relational database. You interact with it using SQL (Structured Query Language). This tutorial uses the `psql` command-line client and includes notes about pgAdmin (GUI). We'll create a small sample database for a fictional company that has `employees`, `departments`, and `projects`.

---

2. Installation (Windows / macOS / Linux)

Windows

1. Download the installer from the PostgreSQL website (choose latest stable).
2. Run the installer — choose components: PostgreSQL server, pgAdmin, psql, Stack Builder (optional).
3. Set a password for the `postgres` superuser during install.
4. Start pgAdmin or use `psql` from the Start Menu.

macOS (Homebrew)

```
# install via Homebrew
brew update
brew install postgresql

# start service
brew services start postgresql

# create superuser (if needed)
createuser -s $(whoami)
# or use psql as default user
psql postgres
```

Linux (Ubuntu/Debian)

```
sudo apt update
sudo apt install postgresql postgresql-contrib
```

```
# start/enable service
sudo systemctl enable --now postgresql

# switch to postgres user and create DB/user
sudo -i -u postgres
psql
# inside psql:
\password postgres   -- set password
CREATE ROLE myuser WITH LOGIN PASSWORD 'mypassword';
CREATE DATABASE mydb OWNER myuser;
\q
exit
```

---

3. Connect with psql and basic commands

Open terminal / command prompt and run:

```
# connect (replace user/db as needed)
psql -U myuser -d mydb -h localhost
# or if local and same user:
psql mydb
```

Useful `psql` meta-commands:

- `\l` — list databases
- `\c dbname` — connect to database
- `\dt` — list tables in current schema
- `\d table_name` — describe table structure
- `\q` — quit

---

4. Create sample database & schema (step-by-step)

Run these SQL statements in `psql` (or in pgAdmin -> Query Tool).

```
-- Create tables
CREATE TABLE departments (
  dept_id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  location VARCHAR(100)
);

CREATE TABLE employees (
  emp_id SERIAL PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  email VARCHAR(100) UNIQUE,
  hire_date DATE,
  salary NUMERIC(12,2),
  dept_id INT REFERENCES departments(dept_id)
);

CREATE TABLE projects (
  project_id SERIAL PRIMARY KEY,
  name VARCHAR(150),
  start_date DATE,
  end_date DATE
);

CREATE TABLE employee_projects (
  emp_id INT REFERENCES employees(emp_id),
  project_id INT REFERENCES projects(project_id),
  role VARCHAR(100),
  PRIMARY KEY (emp_id, project_id)
);
```

Insert sample rows:

```
INSERT INTO departments (name, location) VALUES
('Engineering', 'Bengaluru'),
('Sales', 'Mumbai'),
('HR', 'Chennai');

INSERT INTO employees (first_name, last_name, email, hire_date,
salary, dept_id) VALUES
('Asha','Kumar','asha.kumar@example.com','2023-01-15',70000,1),
('Vikram','Singh','vikram.singh@example.com','2022-06-10',85000,1),
('Priya','Reddy','priya.reddy@example.com','2021-11-01',60000,2);

INSERT INTO projects (name, start_date, end_date) VALUES
('Platform Rework','2024-01-01','2024-12-31'),
('CRM Upgrade','2023-05-01','2024-05-31');

INSERT INTO employee_projects (emp_id, project_id, role) VALUES
(1,1,'Developer'),
(2,1,'Lead'),
(2,2,'Consultant'),
(3,2,'Sales Lead');
```

---

5. BASIC SQL (SELECT / CRUD / filters / sorting)

Select everything:

```
SELECT * FROM employees;
```

Select specific columns:

```
SELECT first_name, last_name, salary FROM employees;
```

Filter with WHERE:

```
SELECT * FROM employees WHERE salary > 65000;
```

Use AND/OR/NOT:

```
SELECT * FROM employees WHERE dept_id = 1 AND salary >= 70000;
```

Sort results:

```
SELECT * FROM employees ORDER BY hire_date DESC;
```

Limit rows:

```
SELECT * FROM employees ORDER BY hire_date DESC LIMIT 2;
```

Insert (we already used INSERT, but single row):

```
INSERT INTO employees
(first_name,last_name,email,hire_date,salary,dept_id)
VALUES ('Ramesh','K','ramesh.k@example.com','2024-03-01',50000,3);
```

Update:

```
UPDATE employees SET salary = salary * 1.05 WHERE emp_id = 1;
```

Delete:

```
DELETE FROM employees WHERE emp_id = 4; -- example id
```

Aggregate functions:

```
SELECT COUNT(*) AS total_employees, AVG(salary) AS avg_salary FROM
employees;
```

GROUP BY:

```
SELECT dept_id, COUNT(*) AS headcount, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept_id;
```

HAVING (filter groups):

```
SELECT dept_id, COUNT(*) AS headcount
FROM employees
GROUP BY dept_id
HAVING COUNT(*) > 1;
```

---

6. INTERMEDIATE SQL (joins, subqueries, CTEs)

INNER JOIN:

```
SELECT e.first_name, e.last_name, d.name AS department
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
```

LEFT JOIN (keep all left rows):

```
SELECT e.first_name, e.last_name, d.name AS department
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id;
```

Many-to-many join (employees → employee_projects → projects):

```
SELECT e.first_name, e.last_name, p.name AS project_name, ep.role
FROM employees e
JOIN employee_projects ep ON e.emp_id = ep.emp_id
JOIN projects p ON ep.project_id = p.project_id;
```

Subquery in SELECT:

```
SELECT first_name, last_name,
  (SELECT name FROM departments WHERE dept_id = employees.dept_id) AS
dept_name
```

```
FROM employees;
```

Subquery in WHERE:

```
SELECT * FROM employees WHERE dept_id IN (SELECT dept_id FROM
departments WHERE location='Bengaluru');
```

Common Table Expressions (CTE) — readable temporary result:

```
WITH high_paid AS (
  SELECT emp_id, first_name, salary FROM employees WHERE salary >
65000
)
SELECT * FROM high_paid;
```

CTE + aggregation:

```
WITH dept_stats AS (
  SELECT dept_id, COUNT(*) AS cnt, AVG(salary) AS avg_sal
  FROM employees
  GROUP BY dept_id
)
SELECT d.name, ds.cnt, ds.avg_sal
FROM dept_stats ds JOIN departments d ON ds.dept_id = d.dept_id;
```

Window functions:

```
SELECT emp_id, first_name, salary,
  RANK() OVER (ORDER BY salary DESC) AS salary_rank,
  AVG(salary) OVER () AS avg_company_salary
FROM employees;
```

---

7. ADVANCED SQL (indexes, transactions, views, materialized views, performance)

Create an index:

```
CREATE INDEX idx_employees_email ON employees (email);
```

When to index: columns used in WHERE, JOIN, ORDER BY frequently. Avoid over-indexing.

Transactions (atomic operations):

```
BEGIN;
UPDATE employees SET salary = salary * 1.10 WHERE dept_id = 1;
UPDATE departments SET location = 'Bengaluru HQ' WHERE dept_id = 1;
-- if all OK:
COMMIT;
-- if something goes wrong:
ROLLBACK;
```

Views (virtual saved queries):

```
CREATE VIEW employee_overview AS
SELECT e.emp_id, e.first_name, e.last_name, d.name AS department,
e.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
-- use:
SELECT * FROM employee_overview WHERE salary > 65000;
```

Materialized view (stores results — refresh manually):

```
CREATE MATERIALIZED VIEW mv_dept_salary AS
SELECT dept_id, AVG(salary) AS avg_salary FROM employees GROUP BY
dept_id;

-- refresh:
REFRESH MATERIALIZED VIEW mv_dept_salary;
```

Explain plans (see query performance):

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE salary > 50000;
```

Partitioning (big tables): PostgreSQL supports range/list/hash partitioning. Example (simplified):

```
-- create partitioned table
CREATE TABLE events (
  id serial primary key,
  event_date date,
  details text
) PARTITION BY RANGE (event_date);

-- create partitions
CREATE TABLE events_2024 PARTITION OF events FOR VALUES FROM
('2024-01-01') TO ('2025-01-01');
```

Advanced features: JSONB columns, full-text search, stored procedures (PL/pgSQL), extensions (PostGIS, pg_trgm).

JSONB example:

```
CREATE TABLE products (id serial primary key, data jsonb);
INSERT INTO products (data) VALUES
('{"name":"Widget","price":19.99,"tags":["sale","new"]}');
SELECT data->>'name' AS name, (data->>'price')::numeric AS price FROM
products;
```

---

8. Backup and restore

Backup with `pg_dump`:

```
# SQL text format
pg_dump -U myuser -d mydb -f mydb_dump.sql

# compressed custom format
pg_dump -U myuser -d mydb -Fc -f mydb_dump.dump
```

Restore with `psql` (SQL format) or `pg_restore` (custom):

```
psql -U myuser -d newdb -f mydb_dump.sql
pg_restore -U myuser -d newdb mydb_dump.dump
```

---

9. Security & roles (basic)

Create role:

```
CREATE ROLE analyst WITH LOGIN PASSWORD 'analystpwd';
GRANT CONNECT ON DATABASE mydb TO analyst;
GRANT USAGE ON SCHEMA public TO analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;
-- set default privileges for future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO
analyst;
```

---

10. Practical exercises (practice these)

1. List employees hired after Jan 1, 2023 and show department name.
2. Find top-2 highest paid employees and their projects.
3. Increase salary by 8% for employees in Engineering department. (use transaction)
4. Create a view showing employee full name and department.
5. Write a query that shows department name and number of projects employees in that department are assigned to.
6. Create a materialized view for average salary per department and refresh it.

Solutions (sketch)

- Use SELECT with JOIN and WHERE.
- Use ORDER BY salary DESC LIMIT 2 and JOIN employee_projects -> projects.
- Use BEGIN; UPDATE ... WHERE dept_id=...; COMMIT;
- `CREATE VIEW ... AS SELECT first_name || ' ' || last_name AS full_name, d.name ...`
- Use joins and `COUNT(DISTINCT ep.project_id)` grouped by department.

---

11. Common pitfalls & tips

- Always use transactions for multiple related writes.
- Use parametrized queries (from apps) to avoid SQL injection — never concatenate user input into SQL.
- Index the columns used often in WHERE/JOIN; avoid indexing small cardinality columns like boolean flags.
- Use `EXPLAIN ANALYZE` to check slow queries.
- Keep backups and test restores regularly.

---

12. Quick cheat-sheet (commands)

- `SELECT`, `INSERT`, `UPDATE`, `DELETE`
- `JOIN` (INNER, LEFT, RIGHT, FULL)
- `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`
- `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
- `CREATE INDEX`, `DROP INDEX`
- `BEGIN`, `COMMIT`, `ROLLBACK`
- `EXPLAIN ANALYZE <query>`
- `pg_dump`, `pg_restore` / `psql -f`

---

13. Where to go next (learning path)

1. Master JOINs and subqueries.
2. Learn window functions deeply (ROW_NUMBER, RANK, LAG, LEAD).
3. Learn query planning and indexes (`EXPLAIN ANALYZE`).
4. Explore PL/pgSQL for stored procedures.
5. Learn replication, backups, and performance tuning.
6. Build CRUD API services (Node/Python/Java) that connect to PostgreSQL.