VERSIÓN 1.0 | GUÍA TÉCNICA

SUPABASE

SQL + PYTHON + JAVASCRIPT

PostgreSQL + Comandos + Conexiones con ORM

Setup, SQL avanzado, Python (psycopg2 + SQLAlchemy) y JavaScript (Prisma + supabase-js)

Crea tu Supabase, domina comandos SQL,
conecta con Python (directo y ORM),
conecta con JavaScript (directo y ORM).
Ejemplos completos listos para producción.

SQL Completo | Python + ORM | JavaScript + ORM

Guía Técnica Profesional

Para desarrolladores backend y full-stack

Enero 2026 | Actualizado

Índice general

1

Setup de Supabase

1.1 Crear Proyecto en Supabase

1.1.1 Paso 1: Registro

1. Ve a `https://supabase.com`
2. Click en "Start your project"
3. Regístrate con GitHub o email
4. Verifica tu correo

1.1.2 Paso 2: Crear Proyecto

1. Click en "New Project"
2. Completa:
   - **Name:** `mi-proyecto-db`
   - **Database Password:** Guarda esto (lo necesitarás)
   - **Region:** Selecciona el más cercano (ej: South America)
   - **Pricing Plan:** Free (suficiente para desarrollo)
3. Click en "Create new project"
4. Espera 2-3 minutos mientras se aprovisiona

Importante: Guarda la contraseña de la base de datos. No podrás verla después.

1.1.3 Paso 3: Obtener Credenciales

En el dashboard de Supabase, ve a **Settings** → **Database**:

```
# URL del proyecto
https://abcdefghijklmnop.supabase.co

# Anon/Public Key (para cliente JavaScript)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

# Service Role Key (para backend, NUNCA expongas)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

# Database Connection String
postgresql://postgres:[PASSWORD]@db.abcdefg.supabase.co:5432/postgres

# Direct Connection (para psycopg2, SQLAlchemy)
Host: db.abcdefg.supabase.co
Database: postgres
```

```
User: postgres
Password: [TU PASSWORD]
Port: 5432
```

Listing 1.1: Credenciales necesarias

Tip: Crea un archivo `.env` y guarda estas credenciales. Nunca las subas a Git.

1.2 Estructura del Dashboard

1.2.1 Tabla Explorer

Aquí creas y visualizas tablas gráficamente.

1.2.2 SQL Editor

Ejecuta queries SQL directamente. Equivalente a psql o pgAdmin.

1.2.3 Database Settings

Aquí encuentras la conexión directa a PostgreSQL.

2

Comandos SQL Esenciales

2.1 Crear Tablas (CREATE TABLE)

2.1.1 Tabla Básica

```sql
CREATE TABLE usuarios (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  edad INTEGER CHECK (edad >= 18),
  fecha_registro TIMESTAMP DEFAULT NOW(),
  activo BOOLEAN DEFAULT TRUE
);
```

Listing 2.1: Crear tabla usuarios

2.1.2 Tabla con Clave Foránea

```sql
CREATE TABLE pedidos (
  id SERIAL PRIMARY KEY,
  usuario_id INTEGER NOT NULL REFERENCES usuarios(id) ON DELETE CASCADE,
  producto VARCHAR(200),
  cantidad INTEGER DEFAULT 1,
  precio DECIMAL(10,2),
  fecha_pedido TIMESTAMP DEFAULT NOW()
);
```

Listing 2.2: Crear tabla pedidos con FK

2.1.3 Tabla con Relación Many-to-Many

```sql
CREATE TABLE cursos (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  creditos INTEGER
);

CREATE TABLE estudiantes (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE
);
```

```sql
-- Tabla de unión
CREATE TABLE inscripciones (
  id SERIAL PRIMARY KEY,
  estudiante_id INTEGER REFERENCES estudiantes(id) ON DELETE CASCADE,
  curso_id INTEGER REFERENCES cursos(id) ON DELETE CASCADE,
  nota DECIMAL(3,1),
  fecha_inscripcion DATE DEFAULT CURRENT_DATE,
  UNIQUE(estudiante_id, curso_id)
);
```

Listing 2.3: Sistema de cursos y estudiantes

2.2 Insertar Datos (INSERT)

2.2.1 Insert Simple

```sql
INSERT INTO usuarios (nombre, email, edad)
VALUES ('Juan Pérez', 'juan@email.com', 25);
```

Listing 2.4: Insertar usuario

2.2.2 Insert Múltiple

```sql
INSERT INTO usuarios (nombre, email, edad) VALUES
  ('María García', 'maria@email.com', 30),
  ('Carlos López', 'carlos@email.com', 28),
  ('Ana Martínez', 'ana@email.com', 22);
```

Listing 2.5: Insertar varios registros

2.2.3 Insert con RETURNING

```sql
INSERT INTO usuarios (nombre, email, edad)
VALUES ('Pedro Sánchez', 'pedro@email.com', 35)
RETURNING id, nombre, fecha_registro;
```

Listing 2.6: Insert y devolver ID

2.3 Consultar Datos (SELECT)

2.3.1 Select Básico

```sql
-- Todos los campos
SELECT * FROM usuarios;

-- Campos específicos
SELECT nombre, email FROM usuarios;
```

```sql
-- Con filtro
SELECT * FROM usuarios WHERE edad > 25;

-- Con múltiples condiciones
SELECT * FROM usuarios
WHERE edad >= 25 AND activo = TRUE;

-- Ordenamiento
SELECT * FROM usuarios ORDER BY fecha_registro DESC;

-- Limitado
SELECT * FROM usuarios LIMIT 10;

-- Con paginación
SELECT * FROM usuarios
ORDER BY id
LIMIT 10 OFFSET 20;
```

Listing 2.7: Consultas básicas

2.3.2 Operadores de Comparación

```sql
-- Igualdad
SELECT * FROM usuarios WHERE edad = 25;

-- Mayor/menor
SELECT * FROM usuarios WHERE edad > 25;
SELECT * FROM usuarios WHERE edad <= 30;

-- LIKE (patrón)
SELECT * FROM usuarios WHERE nombre LIKE 'Juan%';
SELECT * FROM usuarios WHERE email LIKE '%@gmail.com';

-- IN (lista)
SELECT * FROM usuarios WHERE edad IN (25, 30, 35);

-- BETWEEN (rango)
SELECT * FROM usuarios WHERE edad BETWEEN 20 AND 30;

-- IS NULL
SELECT * FROM usuarios WHERE email IS NULL;

-- IS NOT NULL
SELECT * FROM usuarios WHERE email IS NOT NULL;
```

Listing 2.8: Operadores SQL

2.3.3 Funciones de Agregación

```sql
-- Contar registros
SELECT COUNT(*) FROM usuarios;

-- Contar por condición
SELECT COUNT(*) FROM usuarios WHERE activo = TRUE;

-- Suma
SELECT SUM(precio) FROM pedidos;

-- Promedio
SELECT AVG(edad) FROM usuarios;

-- Mínimo y máximo
SELECT MIN(edad), MAX(edad) FROM usuarios;

-- Agrupación
SELECT edad, COUNT(*) as cantidad
FROM usuarios
GROUP BY edad
ORDER BY cantidad DESC;

-- Con filtro HAVING
SELECT edad, COUNT(*) as cantidad
FROM usuarios
GROUP BY edad
HAVING COUNT(*) > 5;
```

Listing 2.9: COUNT

2.4 JOINs

2.4.1 INNER JOIN

```sql
SELECT
  u.nombre,
  u.email,
  p.producto,
  p.cantidad,
  p.precio
FROM usuarios u
INNER JOIN pedidos p ON u.id = p.usuario_id;
```

Listing 2.10: Usuarios con pedidos

2.4.2 LEFT JOIN

```sql
SELECT
  u.nombre,
  u.email,
  COUNT(p.id) as total_pedidos
```

```
FROM usuarios u
LEFT JOIN pedidos p ON u.id = p.usuario_id
GROUP BY u.id, u.nombre, u.email;
```

Listing 2.11: Todos los usuarios

2.4.3 JOIN con Múltiples Tablas

```
SELECT
  e.nombre as estudiante,
  c.nombre as curso,
  i.nota,
  i.fecha_inscripcion
FROM estudiantes e
INNER JOIN inscripciones i ON e.id = i.estudiante_id
INNER JOIN cursos c ON c.id = i.curso_id
WHERE i.nota >= 3.5
ORDER BY e.nombre, c.nombre;
```

Listing 2.12: Estudiantes con cursos e inscripciones

2.5 Actualizar Datos (UPDATE)

2.5.1 Update Básico

```
UPDATE usuarios
SET edad = 26
WHERE id = 1;
```

Listing 2.13: Actualizar usuario

2.5.2 Update Múltiples Campos

```
UPDATE usuarios
SET
  nombre = 'Juan Carlos Pérez',
  email = 'juancarlos@email.com',
  edad = 26
WHERE id = 1;
```

Listing 2.14: Actualizar varios campos

2.5.3 Update con Condiciones

```
UPDATE usuarios
SET activo = FALSE
WHERE fecha_registro < '2023-01-01';
```

Listing 2.15: Actualizar por condición

2.6 Eliminar Datos (DELETE)

2.6.1 Delete con WHERE

```sql
DELETE FROM usuarios WHERE id = 10;
```

Listing 2.16: Eliminar usuario específico

Cuidado: Siempre usa WHERE en DELETE. Sin WHERE, eliminas TODA la tabla.

2.6.2 Delete con Condiciones

```sql
DELETE FROM usuarios
WHERE activo = FALSE
AND fecha_registro < '2020-01-01';
```

Listing 2.17: Eliminar usuarios inactivos

2.7 Índices para Performance

```sql
-- Índice simple
CREATE INDEX idx_usuarios_email ON usuarios(email);

-- Índice compuesto
CREATE INDEX idx_pedidos_usuario_fecha
ON pedidos(usuario_id, fecha_pedido);

-- Índice único
CREATE UNIQUE INDEX idx_usuarios_email_unique
ON usuarios(email);

-- Ver índices existentes
SELECT * FROM pg_indexes WHERE tablename = 'usuarios';
```

Listing 2.18: Crear índices

3

Conexión con Python

3.1 Método 1: Conexión Directa con psycopg2

3.1.1 Instalación

```
pip install psycopg2-binary python-dotenv
```

Listing 3.1: Instalar psycopg2

3.1.2 Configuración

Crea archivo `.env`:

```
DB_HOST=db.abcdefghijklmno.supabase.co
DB_NAME=postgres
DB_USER=postgres
DB_PASSWORD=tu_password_aqui
DB_PORT=5432
```

Listing 3.2: .env

3.1.3 Conexión Básica

```
import psycopg2
from dotenv import load_dotenv
import os

load_dotenv()

# Conectar
conn = psycopg2.connect(
    host=os.getenv("DB_HOST"),
    database=os.getenv("DB_NAME"),
    user=os.getenv("DB_USER"),
    password=os.getenv("DB_PASSWORD"),
    port=os.getenv("DB_PORT")
)

# Crear cursor
cur = conn.cursor()

print("Conexión exitosa a Supabase!")
```

```
# Cerrar
cur.close()
conn.close()
```

Listing 3.3: conexion.py

3.1.4 Operaciones CRUD

```
import psycopg2
from dotenv import load_dotenv
import os

load_dotenv()

def conectar():
    return psycopg2.connect(
        host=os.getenv("DB_HOST"),
        database=os.getenv("DB_NAME"),
        user=os.getenv("DB_USER"),
        password=os.getenv("DB_PASSWORD"),
        port=os.getenv("DB_PORT")
    )

# CREATE
def crear_usuario(nombre, email, edad):
    conn = conectar()
    cur = conn.cursor()

    query = """
    INSERT INTO usuarios (nombre, email, edad)
    VALUES (%s, %s, %s)
    RETURNING id;
    """

    cur.execute(query, (nombre, email, edad))
    user_id = cur.fetchone()[0]

    conn.commit()
    cur.close()
    conn.close()

    return user_id

# READ
def obtener_usuarios():
    conn = conectar()
    cur = conn.cursor()

    cur.execute("SELECT * FROM usuarios;")
    usuarios = cur.fetchall()
```

```
    cur.close()
    conn.close()

    return usuarios

# UPDATE
def actualizar_usuario(user_id, nombre, email):
    conn = conectar()
    cur = conn.cursor()

    query = """
    UPDATE usuarios
    SET nombre = %s, email = %s
    WHERE id = %s;
    """

    cur.execute(query, (nombre, email, user_id))

    conn.commit()
    cur.close()
    conn.close()

# DELETE
def eliminar_usuario(user_id):
    conn = conectar()
    cur = conn.cursor()

    cur.execute("DELETE FROM usuarios WHERE id = %s;", (user_id,))

    conn.commit()
    cur.close()
    conn.close()

# Uso
if __name__ == "__main__":
    # Crear
    nuevo_id = crear_usuario("Juan Pérez", "juan@email.com", 25)
    print(f"Usuario creado con ID: {nuevo_id}")

    # Leer
    usuarios = obtener_usuarios()
    for user in usuarios:
        print(user)

    # Actualizar
    actualizar_usuario(nuevo_id, "Juan Carlos", "juanc@email.com")

    # Eliminar
    eliminar_usuario(nuevo_id)
```

Listing 3.4: crud_psycopg2.py

3.2 Método 2: ORM con SQLAlchemy

3.2.1 Instalación

```
pip install sqlalchemy psycopg2-binary python-dotenv
```

Listing 3.5: Instalar SQLAlchemy

3.2.2 Configuración del Motor

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from dotenv import load_dotenv
import os

load_dotenv()

# Construir DATABASE_URL
DATABASE_URL = f"postgresql+psycopg2://{os.getenv('DB_USER')}:{os.getenv('
    DB_PASSWORD')}@{os.getenv('DB_HOST')}:{os.getenv('DB_PORT')}/{os.getenv('
    DB_NAME')}"

# Crear motor
engine = create_engine(DATABASE_URL, echo=True)

# Crear sesión
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Base para modelos
Base = declarative_base()
```

Listing 3.6: database.py

3.2.3 Definir Modelos

```
from sqlalchemy import Column, Integer, String, Boolean, TIMESTAMP, ForeignKey,
    DECIMAL
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from database import Base

class Usuario(Base):
    __tablename__ = "usuarios"

    id = Column(Integer, primary_key=True, index=True)
```

```
    nombre = Column(String(100), nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    edad = Column(Integer)
    fecha_registro = Column(TIMESTAMP, server_default=func.now())
    activo = Column(Boolean, default=True)

    # Relación
    pedidos = relationship("Pedido", back_populates="usuario")

class Pedido(Base):
    __tablename__ = "pedidos"

    id = Column(Integer, primary_key=True, index=True)
    usuario_id = Column(Integer, ForeignKey("usuarios.id"))
    producto = Column(String(200))
    cantidad = Column(Integer, default=1)
    precio = Column(DECIMAL(10, 2))
    fecha_pedido = Column(TIMESTAMP, server_default=func.now())

    # Relación
    usuario = relationship("Usuario", back_populates="pedidos")
```

Listing 3.7: models.py

3.2.4 Crear Tablas

```
from database import engine, Base
from models import Usuario, Pedido

# Crear todas las tablas
Base.metadata.create_all(bind=engine)
print("Tablas creadas exitosamente!")
```

Listing 3.8: create_tables.py

3.2.5 Operaciones CRUD con ORM

```
from database import SessionLocal
from models import Usuario, Pedido

# CREATE
def crear_usuario(nombre, email, edad):
    db = SessionLocal()

    nuevo_usuario = Usuario(
        nombre=nombre,
        email=email,
        edad=edad
    )

    db.add(nuevo_usuario)
```

```
    db.commit()
    db.refresh(nuevo_usuario)

    db.close()
    return nuevo_usuario.id

# READ - Todos
def obtener_usuarios():
    db = SessionLocal()
    usuarios = db.query(Usuario).all()
    db.close()
    return usuarios

# READ - Por ID
def obtener_usuario(user_id):
    db = SessionLocal()
    usuario = db.query(Usuario).filter(Usuario.id == user_id).first()
    db.close()
    return usuario

# READ - Con filtros
def obtener_usuarios_activos():
    db = SessionLocal()
    usuarios = db.query(Usuario).filter(Usuario.activo == True).all()
    db.close()
    return usuarios

# UPDATE
def actualizar_usuario(user_id, nombre=None, email=None):
    db = SessionLocal()
    usuario = db.query(Usuario).filter(Usuario.id == user_id).first()

    if usuario:
        if nombre:
            usuario.nombre = nombre
        if email:
            usuario.email = email

        db.commit()
        db.refresh(usuario)

    db.close()
    return usuario

# DELETE
def eliminar_usuario(user_id):
    db = SessionLocal()
    usuario = db.query(Usuario).filter(Usuario.id == user_id).first()

    if usuario:
        db.delete(usuario)
```

```
        db.commit()

    db.close()

# JOIN - Usuario con pedidos
def obtener_usuario_con_pedidos(user_id):
    db = SessionLocal()
    usuario = db.query(Usuario).filter(Usuario.id == user_id).first()

    if usuario:
        print(f"Usuario: {usuario.nombre}")
        for pedido in usuario.pedidos:
            print(f"  - Pedido: {pedido.producto}, Cantidad: {pedido.cantidad}")

    db.close()

# Uso
if __name__ == "__main__":
    # Crear
    user_id = crear_usuario("María García", "maria@email.com", 30)
    print(f"Usuario creado: {user_id}")

    # Leer todos
    usuarios = obtener_usuarios()
    for u in usuarios:
        print(f"{u.id}: {u.nombre} ({u.email})")

    # Actualizar
    actualizar_usuario(user_id, nombre="María Fernanda García")

    # Eliminar
    eliminar_usuario(user_id)
```

Listing 3.9: crud_sqlalchemy.py

4

Conexión con JavaScript

4.1 Método 1: Cliente Supabase (supabase-js)

4.1.1 Instalación

```
npm install @supabase/supabase-js dotenv
```

Listing 4.1: Instalar supabase-js

4.1.2 Configuración

Crea archivo `.env`:

```
SUPABASE_URL=https://abcdefghijklmno.supabase.co
SUPABASE_ANON_KEY=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Listing 4.2: .env

4.1.3 Conexión Básica

```
import { createClient } from '@supabase/supabase-js';
import dotenv from 'dotenv';

dotenv.config();

const supabaseUrl = process.env.SUPABASE_URL;
const supabaseKey = process.env.SUPABASE_ANON_KEY;

export const supabase = createClient(supabaseUrl, supabaseKey);

console.log("Cliente Supabase inicializado!");
```

Listing 4.3: supabaseClient.js

4.1.4 Operaciones CRUD

```
import { supabase } from './supabaseClient.js';

// CREATE
async function crearUsuario(nombre, email, edad) {
  const { data, error } = await supabase
    .from('usuarios')
```

```
    .insert([
      { nombre, email, edad }
    ])
    .select();

  if (error) {
    console.error('Error al crear:', error);
    return null;
  }

  return data[0];
}

// READ - Todos
async function obtenerUsuarios() {
  const { data, error } = await supabase
    .from('usuarios')
    .select('*');

  if (error) {
    console.error('Error al leer:', error);
    return [];
  }

  return data;
}

// READ - Por ID
async function obtenerUsuario(id) {
  const { data, error } = await supabase
    .from('usuarios')
    .select('*')
    .eq('id', id)
    .single();

  if (error) {
    console.error('Error:', error);
    return null;
  }

  return data;
}

// READ - Con filtros
async function obtenerUsuariosActivos() {
  const { data, error } = await supabase
    .from('usuarios')
    .select('*')
    .eq('activo', true)
    .order('fecha_registro', { ascending: false });
```

```
  if (error) {
    console.error('Error:', error);
    return [];
  }

  return data;
}

// UPDATE
async function actualizarUsuario(id, updates) {
  const { data, error } = await supabase
    .from('usuarios')
    .update(updates)
    .eq('id', id)
    .select();

  if (error) {
    console.error('Error al actualizar:', error);
    return null;
  }

  return data[0];
}

// DELETE
async function eliminarUsuario(id) {
  const { error } = await supabase
    .from('usuarios')
    .delete()
    .eq('id', id);

  if (error) {
    console.error('Error al eliminar:', error);
    return false;
  }

  return true;
}

// JOIN - Usuario con pedidos
async function obtenerUsuarioConPedidos(userId) {
  const { data, error } = await supabase
    .from('usuarios')
    .select(`
      id,
      nombre,
      email,
      pedidos (
        id,
        producto,
        cantidad,
```

```
        precio,
        fecha_pedido
      )
    `)
    .eq('id', userId)
    .single();

  if (error) {
    console.error('Error:', error);
    return null;
  }

  return data;
}

// Uso
(async () => {
  // Crear
  const nuevoUsuario = await crearUsuario('Juan Pérez', 'juan@email.com', 25);
  console.log('Usuario creado:', nuevoUsuario);

  // Leer todos
  const usuarios = await obtenerUsuarios();
  console.log('Usuarios:', usuarios);

  // Actualizar
  const actualizado = await actualizarUsuario(nuevoUsuario.id, {
    nombre: 'Juan Carlos Pérez'
  });
  console.log('Actualizado:', actualizado);

  // Eliminar
  await eliminarUsuario(nuevoUsuario.id);
})();
```

Listing 4.4: crud.js

4.2 Método 2: ORM con Prisma

4.2.1 Instalación

```
npm install prisma @prisma/client
npx prisma init
```

Listing 4.5: Instalar Prisma

4.2.2 Configuración

Actualiza `prisma/schema.prisma`:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Usuario {
  id              Int       @id @default(autoincrement())
  nombre          String    @db.VarChar(100)
  email           String    @unique @db.VarChar(100)
  edad            Int?
  fechaRegistro   DateTime  @default(now()) @map("fecha_registro")
  activo          Boolean   @default(true)
  pedidos         Pedido[]

  @@map("usuarios")
}

model Pedido {
  id            Int       @id @default(autoincrement())
  usuarioId     Int       @map("usuario_id")
  producto      String?   @db.VarChar(200)
  cantidad      Int       @default(1)
  precio        Decimal   @db.Decimal(10, 2)
  fechaPedido   DateTime  @default(now()) @map("fecha_pedido")

  usuario       Usuario   @relation(fields: [usuarioId], references: [id],
   onDelete: Cascade)

  @@map("pedidos")
}
```

Listing 4.6: prisma/schema.prisma

Actualiza `.env`:

```
DATABASE_URL="postgresql://postgres:PASSWORD@db.abcdefg.supabase.co:5432/postgres
    "
```

Listing 4.7: .env

4.2.3 Generar Cliente

```
npx prisma generate
npx prisma db pull  # Sincronizar schema con BD existente
```

Listing 4.8: Generar cliente Prisma

4.2.4 Operaciones CRUD con Prisma

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

// CREATE
async function crearUsuario(nombre, email, edad) {
  const usuario = await prisma.usuario.create({
    data: {
      nombre,
      email,
      edad
    }
  });

  return usuario;
}

// READ - Todos
async function obtenerUsuarios() {
  const usuarios = await prisma.usuario.findMany();
  return usuarios;
}

// READ - Por ID
async function obtenerUsuario(id) {
  const usuario = await prisma.usuario.findUnique({
    where: { id }
  });

  return usuario;
}

// READ - Con filtros
async function obtenerUsuariosActivos() {
  const usuarios = await prisma.usuario.findMany({
    where: {
      activo: true,
      edad: {
        gte: 18
      }
    },
    orderBy: {
      fechaRegistro: 'desc'
    }
  });

  return usuarios;
}
```

```
// UPDATE
async function actualizarUsuario(id, datos) {
  const usuario = await prisma.usuario.update({
    where: { id },
    data: datos
  });

  return usuario;
}

// DELETE
async function eliminarUsuario(id) {
  await prisma.usuario.delete({
    where: { id }
  });
}

// JOIN - Usuario con pedidos
async function obtenerUsuarioConPedidos(id) {
  const usuario = await prisma.usuario.findUnique({
    where: { id },
    include: {
      pedidos: true
    }
  });

  return usuario;
}

// Transacción
async function crearUsuarioConPedido(nombre, email, edad, producto, precio) {
  const result = await prisma.$transaction(async (tx) => {
    const usuario = await tx.usuario.create({
      data: { nombre, email, edad }
    });

    const pedido = await tx.pedido.create({
      data: {
        usuarioId: usuario.id,
        producto,
        precio
      }
    });

    return { usuario, pedido };
  });

  return result;
}

// Uso
```

```
(async () => {
  try {
    // Crear
    const nuevoUsuario = await crearUsuario('María García', 'maria@email.com',
    30);
    console.log('Usuario creado:', nuevoUsuario);

    // Leer todos
    const usuarios = await obtenerUsuarios();
    console.log('Usuarios:', usuarios);

    // Actualizar
    const actualizado = await actualizarUsuario(nuevoUsuario.id, {
      nombre: 'María Fernanda García'
    });
    console.log('Actualizado:', actualizado);

    // Obtener con pedidos
    const conPedidos = await obtenerUsuarioConPedidos(nuevoUsuario.id);
    console.log('Con pedidos:', conPedidos);

    // Eliminar
    await eliminarUsuario(nuevoUsuario.id);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await prisma.$disconnect();
  }
})();
```

Listing 4.9: crud_prisma.js

4.3 Método 3: Conexión Directa con node-postgres

4.3.1 Instalación

```
npm install pg dotenv
```

Listing 4.10: Instalar pg

4.3.2 Conexión

```
import pg from 'pg';
import dotenv from 'dotenv';

dotenv.config();

const { Pool } = pg;
```

```
const pool = new Pool({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT
});

export default pool;
```

Listing 4.11: db.js

4.3.3 Operaciones CRUD

```
import pool from './db.js';

// CREATE
async function crearUsuario(nombre, email, edad) {
  const query = `
    INSERT INTO usuarios (nombre, email, edad)
    VALUES ($1, $2, $3)
    RETURNING *;
  `;

  const result = await pool.query(query, [nombre, email, edad]);
  return result.rows[0];
}

// READ - Todos
async function obtenerUsuarios() {
  const result = await pool.query('SELECT * FROM usuarios;');
  return result.rows;
}

// READ - Por ID
async function obtenerUsuario(id) {
  const result = await pool.query(
    'SELECT * FROM usuarios WHERE id = $1;',
    [id]
  );
  return result.rows[0];
}

// UPDATE
async function actualizarUsuario(id, nombre, email) {
  const query = `
    UPDATE usuarios
    SET nombre = $1, email = $2
    WHERE id = $3
    RETURNING *;
  `;
```

```
  const result = await pool.query(query, [nombre, email, id]);
  return result.rows[0];
}

// DELETE
async function eliminarUsuario(id) {
  await pool.query('DELETE FROM usuarios WHERE id = $1;', [id]);
}

// Uso
(async () => {
  try {
    const usuario = await crearUsuario('Carlos López', 'carlos@email.com', 28);
    console.log('Usuario creado:', usuario);

    const usuarios = await obtenerUsuarios();
    console.log('Todos los usuarios:', usuarios);

    await eliminarUsuario(usuario.id);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await pool.end();
  }
})();
```

Listing 4.12: crud_pg.js

Conclusión

Resumen de Métodos

Método	Ventajas	Cuándo usar
Python psycopg2	Control total, SQL directo	Scripts simples, migraciones
Python SQLAlchemy	ORM completo, type-safe	Apps grandes, modelos complejos
JS supabase-js	Fácil, realtime, auth	Apps web con Supabase
JS Prisma	Type-safe, migraciones	Backend TypeScript profesional
JS node-postgres	SQL directo, ligero	Microservicios, APIs simples

Recomendaciones Finales

Para proyectos reales:

- Usa **SQLAlchemy** (Python) o **Prisma** (JavaScript) para apps productivas
- Usa **psycopg2** o **node-postgres** para scripts y migraciones
- Usa **supabase-js** si necesitas realtime o auth de Supabase
- Siempre usa variables de entorno para credenciales
- Implementa connection pooling en producción
- Agrega índices a columnas frecuentemente consultadas

—

Versión 1.0 | Enero 2026
Supabase: SQL + Python + JavaScript
Guía Técnica Completa