IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 15

Bases de datos e IA

Avanzado

iacedemy.com — 2026

MÓDULO 15

Bases de datos e IA

Nivel: Avanzado

Autor: Ricardo Gutierrez

Publicación: Mayo 2026

Plataforma: iacedemy.com

SQL que necesitas saber

No necesitas ser un administrador de bases de datos. Pero necesitas dominar las operaciones fundamentales porque sin ellas no puedes verificar lo que la IA genera. La IA es excelente escribiendo SQL, pero si no entiendes un JOIN o una window function, no puedes saber si el resultado es correcto.

SELECT con filtros

```
 -- Obtener tareas de un usuario, ordenadas por prioridad
SELECT id, title, priority, status, created_at
FROM tasks
WHERE user_id = 'abc-123'
  AND status != 'cancelled'
  AND priority >= 3
ORDER BY priority DESC, created_at DESC
LIMIT 20
OFFSET 0;  -- Para paginacion: OFFSET = (page - 1) * limit
```

JOIN para combinar tablas

```
 -- Tareas con nombre del usuario asignado y proyecto
SELECT
    t.id,
    t.title,
    t.priority,
    u.name AS assignee,
    p.name AS project_name
FROM tasks t
INNER JOIN users u ON t.assigned_to = u.id
INNER JOIN projects p ON t.project_id = p.id
WHERE t.status = 'pending'
ORDER BY t.priority DESC;

-- LEFT JOIN: incluir tareas SIN usuario asignado
SELECT t.title, u.name AS assignee
FROM tasks t
```

```
LEFT JOIN users u ON t.assigned_to = u.id;
-- Las tareas sin asignar tendran assignee = NULL
```

INNER JOIN: solo filas que existen en ambas tablas. **LEFT JOIN**: todas las filas de la tabla izquierda, aunque no tengan match en la derecha. El 90% de tus JOINs seran uno de estos dos.

GROUP BY para agregar

```
 -- Cuantas tareas hay por estado y prioridad media
SELECT
    status,
    COUNT(*) AS total_tasks,
    AVG(priority) AS avg_priority,
    MIN(created_at) AS oldest_task
FROM tasks
WHERE project_id = 42
GROUP BY status
HAVING COUNT(*) > 5  -- Solo estados con mas de 5 tareas
ORDER BY total_tasks DESC;
```

Subqueries

```
 -- Usuarios que tienen tareas de prioridad critica
SELECT name, email
FROM users
WHERE id IN (
    SELECT DISTINCT assigned_to
    FROM tasks
    WHERE priority = 5 AND status = 'pending'
);

-- Tareas que tardan mas que el promedio
SELECT title, created_at, completed_at
FROM tasks
WHERE EXTRACT(EPOCH FROM completed_at - created_at) > (
    SELECT AVG(EXTRACT(EPOCH FROM completed_at - created_at))
    FROM tasks
    WHERE completed_at IS NOT NULL
);
```

Window functions

Las window functions son el superpoder de SQL. Permiten calcular agregaciones sin colapsar las filas. Extremadamente utiles para rankings, totales acumulados, y comparaciones.

```
 -- Ranking de tareas por prioridad dentro de cada proyecto
SELECT
    project_id,
    title,
    priority,
    ROW_NUMBER() OVER (
        PARTITION BY project_id
        ORDER BY priority DESC
    ) AS rank_in_project,
    COUNT(*) OVER (PARTITION BY project_id) AS total_in_project,
    SUM(CASE WHEN status = 'done' THEN 1 ELSE 0 END)
        OVER (PARTITION BY project_id) AS done_in_project
FROM tasks
WHERE status != 'cancelled';

-- Acumulado de tareas completadas por dia
SELECT
    DATE(completed_at) AS day,
    COUNT(*) AS completed_today,
    SUM(COUNT(*)) OVER (ORDER BY DATE(completed_at)) AS cumulative_total
FROM tasks
WHERE completed_at IS NOT NULL
GROUP BY DATE(completed_at)
ORDER BY day;
```

INSERT, UPDATE, DELETE

```
 -- Insertar
INSERT INTO tasks (title, priority, user_id, project_id)
VALUES ('Deploy v2.0', 5, 'abc-123', 42)
RETURNING id, created_at;  -- PostgreSQL: devuelve los campos generados

-- Actualizar
UPDATE tasks
SET status = 'done',
    completed_at = NOW(),
```

```
    updated_at = NOW()
WHERE id = 123 AND user_id = 'abc-123'  -- Siempre filtrar por user_id
RETURNING *;

-- Soft delete (RECOMENDADO)
UPDATE tasks
SET deleted_at = NOW(), updated_at = NOW()
WHERE id = 123;

-- Hard delete (solo para purga)
DELETE FROM tasks
WHERE deleted_at IS NOT NULL
  AND deleted_at < NOW() - INTERVAL '90 days';
```

RETURNING es tu amigo

En PostgreSQL, `RETURNING` despues de INSERT/UPDATE/DELETE devuelve las filas afectadas. Evita una query extra para obtener el registro recien creado/actualizado.

PostgreSQL como eleccion por defecto

Para el 95% de los proyectos, PostgreSQL es la respuesta correcta. Open source, madura (35+ anios), con soporte para tipos avanzados (JSON, arrays, geometria), full-text search, extensiones, y una comunidad enorme.

Cuando **no** usar PostgreSQL:

- Necesitas un key-value store de altisimo rendimiento: Redis
- Necesitas busqueda vectorial especializada: Qdrant, Pinecone (aunque pgvector cubre muchos casos)
- Necesitas una base de datos embebida sin servidor: SQLite
- Necesitas una base de datos de series temporales: TimescaleDB (que es una extension de PostgreSQL)

Para todo lo demas, PostgreSQL. No pierdas tiempo evaluando 15 bases de datos para un MVP.

Supabase: PostgreSQL con superpoderes

Supabase te da una PostgreSQL completa en la nube con extras que ahorran semanas de desarrollo:

- **Auth**: login con email, Google, GitHub, magic link. Configuracion en minutos.
- **API REST auto-generada**: cada tabla tiene endpoints REST automaticos via PostgREST.
- **Realtime**: cambios en la base de datos se envian a los clientes por websockets.
- **Storage**: almacenamiento de archivos con permisos y CDN.
- **Row Level Security**: seguridad a nivel de fila directamente en la base de datos.
- **Edge Functions**: funciones serverless en Deno para logica custom.
- **Vault**: almacen cifrado de secretos.

Conectar desde Python

```
 from supabase import create_client
import os

# Crear cliente (usa ANON key, no service_role)
supabase = create_client(
    os.getenv("SUPABASE_URL"),
    os.getenv("SUPABASE_ANON_KEY")
)

# Insertar un registro
result = supabase.table("tasks").insert({
    "title": "Revisar PR #42",
    "priority": 3,
    "project_id": 1
}).execute()

print(f"Tarea creada con id: {result.data[0]['id']}")

# Consultar con filtros
tasks = supabase.table("tasks") \
    .select("id, title, priority, users(name)") \
    .eq("status", "pending") \
    .gte("priority", 3) \
    .order("priority", desc=True) \
```

```
    .limit(20) \
    .execute()

# Actualizar
supabase.table("tasks") \
    .update({"status": "done", "completed_at": "now()"}) \
    .eq("id", 123) \
    .execute()

# Soft delete
supabase.table("tasks") \
    .update({"deleted_at": "now()"}) \
    .eq("id", 123) \
    .execute()
```

Conectar desde JavaScript/TypeScript

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

const supabase = createClient(
  process.env.NEXT_PUBLIC_SUPABASE_URL!,
  process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!
)

// Consultar con tipos TypeScript
const { data: tasks, error } = await supabase
  .from('tasks')
  .select('id, title, priority, users(name)')
  .eq('status', 'pending')
  .gte('priority', 3)
  .order('priority', { ascending: false })
  .limit(20)

if (error) {
  console.error('Error:', error.message)
} else {
  console.log('Tareas:', tasks)
}
```

Row Level Security

RLS es la funcionalidad mas importante de Supabase para aplicaciones multi-tenant o cualquier aplicacion donde distintos usuarios tienen acceso a distintos datos. La base de datos **filtra automaticamente** las filas segun quien hace la peticion.

Sin RLS vs con RLS

```
 -- SIN RLS: un bug en tu codigo puede exponer datos de todos los usuarios
-- Si olvidas el filtro WHERE user_id = ..., se devuelven TODAS las filas
SELECT * FROM tasks;  -- Devuelve las 50,000 tareas de todos los usuarios

-- CON RLS: la base de datos filtra automaticamente
-- Aunque olvides el WHERE, solo se devuelven las filas del usuario
autenticado
SELECT * FROM tasks;  -- Solo devuelve las tareas del usuario actual
```

Configurar RLS paso a paso

```
 -- 1. Activar RLS en la tabla
ALTER TABLE tasks ENABLE ROW LEVEL SECURITY;

-- 2. Politica de SELECT: cada usuario ve solo sus tareas
CREATE POLICY "users_select_own_tasks"
ON tasks FOR SELECT
USING (user_id = auth.uid());

-- 3. Politica de INSERT: cada usuario crea tareas para si mismo
CREATE POLICY "users_insert_own_tasks"
ON tasks FOR INSERT
WITH CHECK (user_id = auth.uid());

-- 4. Politica de UPDATE: cada usuario edita solo sus tareas
CREATE POLICY "users_update_own_tasks"
ON tasks FOR UPDATE
USING (user_id = auth.uid())
WITH CHECK (user_id = auth.uid());

-- 5. Politica de DELETE: cada usuario elimina solo sus tareas
CREATE POLICY "users_delete_own_tasks"
```

```
ON tasks FOR DELETE
USING (user_id = auth.uid());
```

RLS para multi-tenant con organizaciones

```
 -- Politica basada en organizacion: los miembros ven datos de su org
CREATE POLICY "org_members_select"
ON tasks FOR SELECT
USING (
    project_id IN (
        SELECT p.id FROM projects p
        JOIN organization_members om ON om.organization_id =
p.organization_id
        WHERE om.user_id = auth.uid()
    )
);

-- Politica con roles: solo admins pueden eliminar
CREATE POLICY "org_admins_delete"
ON tasks FOR DELETE
USING (
    EXISTS (
        SELECT 1 FROM organization_members om
        JOIN projects p ON p.organization_id = om.organization_id
        WHERE om.user_id = auth.uid()
          AND om.role = 'admin'
          AND p.id = tasks.project_id
    )
);
```

service_role bypasa RLS

La clave `service_role` de Supabase ignora todas las politicas RLS. Es como tener root. **NUNCA** la expongas en el frontend. Solo se usa en el backend, en funciones de administracion, con maximo cuidado.

Migraciones

Tu esquema de base de datos va a cambiar. Nuevas tablas, nuevas columnas, nuevos indices, cambios en relaciones. Las migraciones son archivos que describen cada cambio, en orden, para que cualquiera pueda recrear tu base de datos desde cero.

Con Alembic (Python/FastAPI)

```
 # Instalar
pip install alembic sqlalchemy

# Inicializar
alembic init migrations

# Configurar alembic.ini con tu DATABASE_URL
# sqlalchemy.url = postgresql://user:pass@host:5432/db

# Crear una migracion
alembic revision --autogenerate -m "add_priority_column_to_tasks"

# Aplicar migraciones pendientes
alembic upgrade head

# Rollback una migracion
alembic downgrade -1

# Ver estado actual
alembic current
```

Archivo de migracion generado

```
 # migrations/versions/001_add_priority_column.py
"""add priority column to tasks"""

from alembic import op
import sqlalchemy as sa

revision = '001'
down_revision = None

def upgrade():
```

```
    op.add_column('tasks',
        sa.Column('priority', sa.Integer(), nullable=False,
server_default='1')
    )
    op.create_index('ix_tasks_priority', 'tasks', ['priority'])

def downgrade():
    op.drop_index('ix_tasks_priority', 'tasks')
    op.drop_column('tasks', 'priority')
```

Con Prisma (TypeScript/Next.js)

```
 // prisma/schema.prisma
model Task {
  id          Int       @id @default(autoincrement())
  title       String    @db.VarChar(200)
  priority    Int       @default(1)
  status      String    @default("pending")
  userId      String    @map("user_id")
  createdAt   DateTime  @default(now()) @map("created_at")
  updatedAt   DateTime  @updatedAt @map("updated_at")
  deletedAt   DateTime? @map("deleted_at")

  user User @relation(fields: [userId], references: [id])

  @@index([userId, status])
  @@index([priority])
  @@map("tasks")
}

// Comandos
// npx prisma migrate dev --name add_priority
// npx prisma migrate deploy  (produccion)
```

Con SQL puro (Supabase)

```
 -- supabase/migrations/001_initial_schema.sql
CREATE TABLE IF NOT EXISTS tasks (
    id BIGSERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    priority INTEGER NOT NULL DEFAULT 1 CHECK (priority BETWEEN 1 AND 5),
    status VARCHAR(20) NOT NULL DEFAULT 'pending',
```

```
    user_id UUID NOT NULL REFERENCES auth.users(id),
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ,
    deleted_at TIMESTAMPTZ
);

CREATE INDEX ix_tasks_user_status ON tasks(user_id, status)
WHERE deleted_at IS NULL;

-- Aplicar con Supabase CLI
-- supabase db push
```

Regla de migraciones

Cada cambio de esquema es una migracion. **NUNCA** modificar la base de datos a mano en produccion. Si necesitas un cambio urgente, crea una migracion, revisala, y aplicala. El historial de migraciones es tu auditoria de cambios.

IA para generar y validar SQL

La IA es excelente generando SQL. Pero tiene una regla fundamental: **siempre valida antes de ejecutar**. Nunca ejecutes una query generada por IA directamente en produccion sin revisarla.

```
 # Prompt para generar SQL
"Escribe una query PostgreSQL que devuelva:
- Las 10 tareas mas antiguas que siguen en estado 'pending'
- Con el nombre del usuario asignado
- Con el nombre del proyecto
- Con cuantos dias llevan abiertas
- Solo de la organizacion con id = 5

Tabla tasks: id, title, status, priority, assigned_to (FK users),
  project_id (FK projects), created_at, deleted_at
Tabla users: id, name, email
Tabla projects: id, name, organization_id
```

```
Requisitos: excluir soft deleted, usar COALESCE para campos nullable."
```

Antes de ejecutar el resultado:

1. Lee la query linea por linea. Entiendes cada JOIN y cada WHERE?
2. Ejecuta con LIMIT 5 primero.
3. Verifica que los datos devueltos son correctos (no mas de los que deberia).
4. Si modifica datos (INSERT/UPDATE/DELETE), ejecuta en una transaccion con ROLLBACK.

JSONB para esquemas flexibles

PostgreSQL tiene un tipo JSONB que almacena JSON de forma nativa, con soporte para indices y queries. Es util cuando parte de tu esquema es flexible o cambiante.

```
 -- Tabla con columna JSONB para metadatos flexibles
CREATE TABLE tasks (
    id BIGSERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    priority INTEGER NOT NULL DEFAULT 1,
    metadata JSONB DEFAULT '{}'::jsonb
);

-- Insertar con metadatos
INSERT INTO tasks (title, priority, metadata) VALUES (
    'Deploy v2',
    5,
    '{"environment": "production", "estimated_hours": 4, "tags": ["deploy",
"urgent"]}'
);

-- Consultar dentro del JSONB
SELECT title, metadata->>'environment' AS env
FROM tasks
WHERE metadata->>'environment' = 'production';

-- Consultar arrays dentro del JSONB
SELECT title
FROM tasks
WHERE metadata->'tags' ? 'urgent';
```

```
-- Indice GIN para queries rapidas en JSONB
CREATE INDEX ix_tasks_metadata ON tasks USING GIN (metadata);
```

Cuando usar JSONB:

- Metadatos que varian por registro (configuraciones, preferencias, tags)
- Datos de integraciones externas con estructura variable
- Prototipos rapidos donde el esquema todavia no esta definido

Cuando **no** usar JSONB:

- Datos que necesitas en WHERE frecuentemente (mejor una columna dedicada)
- Relaciones entre entidades (mejor foreign keys)
- Datos que necesitas validar con restricciones de base de datos

Estrategias de indexacion

Un indice bien colocado puede reducir una query de 2 segundos a 5 milisegundos. Pero indices innecesarios ralentizan las escrituras. La clave es indexar lo que consultas, no todo.

Tipos de indices en PostgreSQL

```
 -- B-tree (por defecto): para igualdad y rangos
-- Usa para: WHERE col = X, WHERE col > X, ORDER BY col
CREATE INDEX ix_tasks_status ON tasks(status);
CREATE INDEX ix_tasks_created ON tasks(created_at DESC);

-- Indice compuesto: para queries con multiples columnas
CREATE INDEX ix_tasks_user_status ON tasks(user_id, status);
-- Orden importa: la query debe filtrar por user_id primero (o ambos)

-- Indice parcial: solo indexa filas que cumplen una condicion
-- Mas pequeno y rapido que indexar toda la tabla
CREATE INDEX ix_tasks_pending ON tasks(priority DESC, created_at)
WHERE status = 'pending' AND deleted_at IS NULL;

-- GIN: para JSONB, arrays y full-text search
CREATE INDEX ix_tasks_metadata ON tasks USING GIN (metadata);
```

```
CREATE INDEX ix_tasks_tags ON tasks USING GIN (metadata->'tags');

-- GiST: para geometria, rangos y busquedas de proximidad
-- Usado con PostGIS para datos geograficos
CREATE INDEX ix_locations_geom ON locations USING GIST (geom);
```

Cuando indexar

- Columnas usadas en WHERE frecuentemente
- Columnas usadas en JOIN (foreign keys)
- Columnas usadas en ORDER BY
- Columnas con alta cardinalidad (muchos valores distintos)

Cuando NO indexar

- Tablas con menos de 1000 filas (el scan secuencial es mas rapido)
- Columnas con baja cardinalidad (ej: boolean con 50/50)
- Columnas que se actualizan constantemente
- Si ya tienes 10+ indices en la tabla (revisa si realmente se usan)

Backup y recuperacion

```
 # pg_dump: backup completo
pg_dump -h host -U user -d dbname -F custom -f backup.dump

# pg_restore: restaurar
pg_restore -h host -U user -d dbname backup.dump

# Backup solo del esquema (sin datos)
pg_dump -h host -U user -d dbname --schema-only > schema.sql

# Backup de una tabla especifica
pg_dump -h host -U user -d dbname -t tasks > tasks_backup.sql
```

Supabase hace backups automaticos diarios. Los proyectos Pro tienen point-in-time recovery (PITR): puedes restaurar la base de datos a cualquier segundo en los ultimos 7 dias. Para proyectos criticos, esto es imprescindible.

Optimizacion con IA

Este es el workflow mas potente para optimizar queries con IA:

```
 -- Paso 1: Identificar la query lenta
EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)
SELECT t.*, u.name, p.name AS project
FROM tasks t
JOIN users u ON t.assigned_to = u.id
JOIN projects p ON t.project_id = p.id
WHERE t.status = 'pending'
  AND t.created_at > NOW() - INTERVAL '7 days'
  AND p.organization_id = 5
ORDER BY t.priority DESC
LIMIT 20;

-- Paso 2: Copiar el output de EXPLAIN ANALYZE

-- Paso 3: Prompt a Claude/ChatGPT:
-- "Tengo esta query en PostgreSQL 15. Tabla tasks: 500k filas.
-- Tabla users: 10k filas. Tabla projects: 500 filas.
-- La query tarda 2.3 segundos. Aqui esta el EXPLAIN ANALYZE:
-- [pegar output]
-- Sugiere indices y/o reescritura para bajar a menos de 100ms."

-- Paso 4: La IA probablemente sugerira algo como:
CREATE INDEX CONCURRENTLY ix_tasks_status_created
ON tasks (status, created_at DESC)
WHERE status = 'pending';

CREATE INDEX CONCURRENTLY ix_projects_org
ON projects (organization_id);

-- Paso 5: Aplicar y medir de nuevo
-- De 2.3 segundos a ~15ms con indices parciales
```

CONCURRENTLY es tu amigo

Siempre crea indices con `CREATE INDEX CONCURRENTLY` en produccion. Sin CONCURRENTLY, PostgreSQL bloquea la tabla durante la creacion del indice, lo que puede tumbar tu aplicacion.

Ejercicio practico

Ejercicio M15: Disenar, proteger y optimizar una base de datos

1. **Disenar esquema**: Crea un esquema PostgreSQL para una app de gestion de proyectos con 4 tablas (organizations, users, projects, tasks). Incluye timestamps, soft delete y restricciones CHECK.
2. **Configurar RLS**: Escribe politicas RLS para que cada organizacion solo vea sus datos. Incluye politicas diferenciadas por rol (admin puede eliminar, member solo leer/escribir).
3. **Crear migraciones**: Usa Alembic o SQL puro para crear 3 migraciones: esquema inicial, agregar columna, agregar indice.
4. **Queries con IA**: Usa Claude o ChatGPT para generar 5 queries complejas (JOINs, GROUP BY, window functions). Revisa cada una antes de ejecutar.
5. **Optimizar**: Ejecuta EXPLAIN ANALYZE en una query con datos de prueba. Pasa el resultado a la IA y aplica las optimizaciones sugeridas. Mide el antes y despues.
6. **JSONB**: Anade una columna JSONB para metadatos flexibles. Crea un indice GIN y escribe una query que filtre por un campo dentro del JSON.

Bonus: Configura un backup automatico con pg_dump y un cron job (o usa Supabase con backups automaticos).

Conclusiones clave

Key takeaways del M15

1. Domina 6 operaciones SQL: SELECT, JOIN, GROUP BY, subqueries, window functions, INSERT/UPDATE/DELETE. Cubren el 90% de tu trabajo diario.
2. PostgreSQL es la eleccion por defecto. Supabase te da PostgreSQL + Auth + RLS + Realtime sin gestionar servidor.
3. RLS es obligatorio para multi-tenant. Sin RLS, un bug expone datos de todos los clientes. Con RLS, la DB filtra automaticamente.
4. Cada cambio de esquema es una migracion. NUNCA modificar la base de datos a mano en produccion.
5. La IA genera SQL excelente, pero siempre valida antes de ejecutar. Especialmente INSERT/UPDATE/DELETE.
6. JSONB para metadatos flexibles, con indice GIN para queries rapidas.
7. Indexa lo que consultas: B-tree para igualdad/rangos, GIN para JSON, indices parciales para subconjuntos.
8. EXPLAIN ANALYZE + IA es la combinacion perfecta para optimizar queries lentas.

Siguiente: M16 - Git y repositorios con IA

Con la base de datos dominada, el siguiente paso es gestionar tu codigo correctamente: Git, GitHub, CI/CD, branching strategies y Claude Code integrado.

Ir al Modulo 16

IACADEMY

iacedemy.com

De fundamentos a arquitectura de IA.
12 módulos prácticos. 24 recursos descargables.
Quizzes con certificado. Vídeos profesionales.

Empieza gratis en iacedemy.com/free