Casos Prácticos de Estudio para Diseño de Bases de Datos en PostgreSQL

Estructura

Fase	Contenido	Herramienta
1	Introducción a PostgreSQL	Editor SQL
2	DDL práctico: Crear base de datos "universidad"	Editor SQL
3	DML práctico: Poblar tablas y consultas básicas	Editor SQL
4	JOIN simple: Relacionar 2-3 tablas	Editor SQL
5	Ejercicios de consolidación	Editor SQL
6	Cierre: Buenas prácticas y preguntas	Discusión

Condiciones para los datos:

- **Comillas**: 'Simples' para texto (strings), "Dobles" para nombres de columnas o tablas con espacios.
- **Concatenación**: MySQL usa CONCAT(), PostgreSQL prefiere ||.
- **Nulos**: IFNULL (MySQL) vs COALESCE (PostgreSQL/Estándar).
- **Booleanos**: PostgreSQL tiene un tipo BOOLEAN real (TRUE/FALSE), MySQL usa TINYINT.

Caso de Estudio: Sistema de Gestión Universitaria Simple

Contexto: Universidad que necesita gestionar:

- Facultades (id, nombre, decano)
- Carreras (id, nombre, id_facultad, duracion_semestres)
- Estudiantes (id, código, nombres, apellidos, id_carrera, fecha_ingreso)
- Profesores (id, código, nombres, apellidos, id_facultad, especialidad)
- Materias (id, código, nombre, id_carrera, creditos)
- Inscripciones (id, id_estudiante, id_materia, semestre, año, nota_final)

Relaciones simples:

- Una facultad tiene muchas carreras
- Una carrera tiene muchos estudiantes y muchas materias
- Un estudiante se inscribe en muchas materias
- Un profesor pertenece a una facultad

Fase 1: Introducción a PostgreSQL

Objetivos:

- Crear una conexión al servidor local
- Conocer la interfaz: Browser, Query Editor, Data Output

Actividad guiada:

1. Crear servidor "localhost" (postgres / contraseña)

2. Explorar las bases de datos de sistema (solo ver, no tocar)

3. Crear una nueva base de datos llamada universidad_practica

Instalación en WLS Ubuntu

Actualizamos la lista o índice de repositorios en sistemas basados en Debian/Ubuntu

bash

sudo apt update

Instalamos Postgressql

bash

sudo apt install postgresql postgresql-contrib

verificamos el estado del servicio con:

bash

sudo service postgresql status

Si no ha iniciado, lo iniciamos con:

bash

sudo service postgresql start

accedemos usando

sudo -u postgres psql

Fase 2: DDL - Crear la base de datos

Modelo Físico de la Base de Datos

Usamos el diagrama ER es idéntico al del caso MySQL, adaptando tipos de datos de PostgreSQL

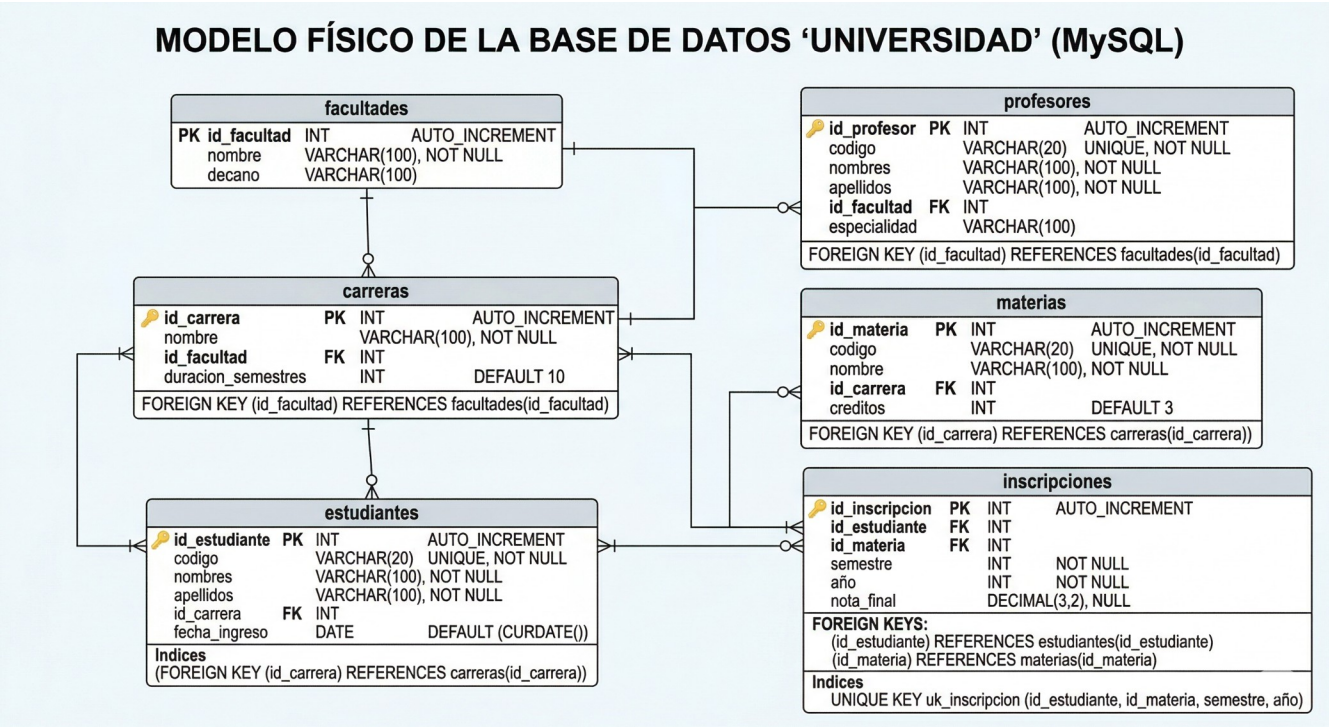
Script completo SQL:

```
-- Ver bases de datos creadas

      \l+

o

      \list+

Para salir q

-- Crear base de datos (ejecutar en psql o pgAdmin con usuario postgres)

      CREATE DATABASE universidad;

-- Conectar a la base de datos

      \connect universidad;

ó
      \c universidad;

Para salir de una base de datos y volver a postgres :

      \connect postgres

Cuidado que \c es diferente a \C. \C universidad, indica que partir de ese momento en que ves ese mensaje, cualquier tabla o resultado que imprimas en la terminal llevará el encabezado "universidad" en la parte superior. Es útil cuando estás generando reportes o quieres identificar rápidamente qué datos estás viendo. Para eliminarlo :

      \C

ó

      \pset title

Creación de las tablas. Serial es similar a AUTO_INCREMENT, pero ademas:
```

- Crea un objeto tipo Sequence (un contador).
- Cambia el tipo de dato a integer.
- Le asigna un valor por defecto (DEFAULT) que llama a la función nextval() de esa secuencia.

Si no pueden usar WSL, simularlo en:

https://onecompiler.com/postgresql



-- Tabla facultades

```
CREATE TABLE facultades (
 id_facultad SERIAL PRIMARY KEY,
 nombre VARCHAR(100) NOT NULL,
    decano VARCHAR(100)
);
```

Para ver las tablas que has creado en la base de datos a la que estás conectado, utiliza el meta-comando:

```
\dt
```

Ademas

\dt: Muestra una lista de todas las tablas en el esquema actual (por defecto, el esquema public).

\dt+: Si quieres ver más detalles, como el tamaño de la tabla y su descripción, añade el signo más.

\d nombre_tabla: Similar a *describe* de MySQL. Si quieres ver la estructura específica de una tabla (sus columnas, tipos de datos e índices), usa este comando.

\d+ nombre_tabla: Similar a \d, pero con mas información.

-- Tabla carreras

```
CREATE TABLE carreras (
 id_carrera SERIAL PRIMARY KEY,
 nombre VARCHAR(100) NOT NULL,
 id_facultad INTEGER,
 duracion_semestres INTEGER DEFAULT 10,
 CONSTRAINT fk_carreras_facultad
 FOREIGN KEY (id_facultad)
 REFERENCES facultades(id_facultad)
 ON DELETE SET NULL
);
```

ON DELETE SET NULL es una regla de integridad referencial que se aplica a las llaves foráneas (Foreign Keys).

Su función es simple: si borras un registro en la tabla "padre", en lugar de borrar también los registros relacionados en la tabla "hijo" (como hace el CASCADE), el sistema pone el valor en NULL en esos registros.

-- Tabla estudiantes

```
CREATE TABLE estudiantes (
 id_estudiante SERIAL PRIMARY KEY,
 codigo VARCHAR(20) UNIQUE NOT NULL,
 nombres VARCHAR(100) NOT NULL,
 apellidos VARCHAR(100) NOT NULL,
 id_carrera INTEGER,
 fecha_ingreso DATE DEFAULT CURRENT_DATE,
 CONSTRAINT fk_estudiantes_carrera
 FOREIGN KEY (id_carrera)
 REFERENCES carreras(id_carrera)
 ON DELETE SET NULL
);
```

CONSTRAINT fk_estudiantes_carrera : Establece una relación referencial entre la tabla estudiantes y la tabla carreras, asegurando que cada valor en la columna id_carrera de estudiantes corresponda a un id_carrera existente en la tabla carreras.

```
-- Tabla profesores

    CREATE TABLE profesores (
     id_profesor SERIAL PRIMARY KEY,
     codigo VARCHAR(20) UNIQUE NOT NULL,
     nombres VARCHAR(100) NOT NULL,
     apellidos VARCHAR(100) NOT NULL,
     id_facultad INTEGER,
     especialidad VARCHAR(100),
     CONSTRAINT fk_profesores_facultad
     FOREIGN KEY (id_facultad)
     REFERENCES facultades(id_facultad)
     ON DELETE SET NULL
    );


-- Tabla materias

    CREATE TABLE materias (
     id_materia SERIAL PRIMARY KEY,
     codigo VARCHAR(20) UNIQUE NOT NULL,
     nombre VARCHAR(100) NOT NULL,
     id_carrera INTEGER,
     creditos INTEGER DEFAULT 3,
     CONSTRAINT fk_materias_carrera
     FOREIGN KEY (id_carrera)
     REFERENCES carreras(id_carrera)
     ON DELETE CASCADE
    );

-- Tabla inscripciones (tabla de relación muchos a muchos)

    CREATE TABLE inscripciones (
     id_inscripcion SERIAL PRIMARY KEY,
     id_estudiante INTEGER NOT NULL,
        id_materia INTEGER NOT NULL,
        semestre INTEGER NOT NULL,
        anio INTEGER NOT NULL,
        nota_final DECIMAL(3,2) CHECK (nota_final >= 0 AND
    nota_final <= 5),
        CONSTRAINT fk_inscripciones_estudiante
            FOREIGN KEY (id_estudiante)
```

```
        REFERENCES estudiantes(id_estudiante)
        ON DELETE CASCADE,
    CONSTRAINT fk_inscripciones_materia
        FOREIGN KEY (id_materia)
        REFERENCES materias(id_materia)
        ON DELETE CASCADE,
    -- Evitar inscripción duplicada
    CONSTRAINT uk_inscripcion
        UNIQUE (id_estudiante, id_materia, semestre, anio)
);
```

El comando CHECK es una regla de validación que actúa como un "filtro" de seguridad directamente en la base de datos.

En este caso específico, hace lo siguiente:

- Limita el rango: Asegura que nadie pueda ingresar una nota negativa (menor a 0) ni una nota superior a 5.00.
- Protege la integridad: Si intentas hacer un INSERT o un UPDATE con una nota de 5.5 o -1.0, PostgreSQL rechazará la operación y lanzará un error, impidiendo que se guarden datos inválidos.
- Define la lógica de negocio: Establece permanentemente que, para tu sistema, la escala de calificación es estrictamente de 0 a 5.

Nota sobre DEFAULT CURRENT_DATE

La función CURRENT_DATE devuelve la fecha actual del sistema (sin la hora). Al usarla como valor por defecto:

• Si insertas un estudiante sin especificar fecha_ingreso, PostgreSQL guarda automáticamente la fecha de hoy.

• Si insertas un estudiante especificando una fecha, se usa la que tú indiques.

Ejemplo:

```
-- Guarda la fecha de hoy automáticamente, porque no le
ingresaste el campo fecha_ingreso
INSERT INTO estudiantes (codigo, nombres, apellidos,
id_carrera) VALUES ('202410001', 'Luis', 'Torres', 1);

-- Guarda la fecha que tú especifiques
```

```
INSERT INTO estudiantes (codigo, nombres, apellidos,
id_carrera, fecha_ingreso) VALUES ('202310006', 'Sofia',
'Ruiz', 2, '2023-02-15');
```

Ejercicios durante la fase:

- Ejecutar el script completo
- Verificar con \dt
- Describir una tabla: \d estudiantes

Fase 3: DML - Poblar y consultar

Inserción de datos:

```
-- Facultades

    INSERT INTO facultades (nombre, decano) VALUES
    ('Ingeniería', 'Dr. Carlos Martínez'),
    ('Ciencias Básicas', 'Dra. Ana López'),
    ('Ciencias Sociales', 'Dr. Pedro Gómez');

-- Carreras

    INSERT INTO carreras (nombre, id_facultad,
    duracion_semestres) VALUES
    ('Ingeniería de Sistemas', 1, 10),
    ('Tecnología en Desarrollo de Software', 1, 6),
    ('Matemáticas', 2, 10),
    ('Psicología', 3, 10);

-- Estudiantes

    INSERT INTO estudiantes (codigo, nombres, apellidos,
    id_carrera, fecha_ingreso) VALUES
    ('202310001', 'Juan Carlos', 'Rodríguez Pérez', 2, '2023-
    02-01'),
    ('202310002', 'María Fernanda', 'López García', 2, '2023-
    02-01'),
    ('202310003', 'Carlos Alberto', 'Martínez Silva', 1, '2022-
    02-01'),
    ('202320001', 'Ana Lucía', 'Gómez Torres', 3, '2023-08-
    01');

-- Profesores

    INSERT INTO profesores (codigo, nombres, apellidos,
    id_facultad, especialidad) VALUES
    ('P001', 'Luis Eduardo', 'Fernández Ruiz', 1, 'Bases de
    Datos'),
    ('P002', 'Diana Patricia', 'Castro Mendoza', 1,
    'Programación Web'),
    ('P003', 'Roberto', 'Gutiérrez Vargas', 2, 'Estadística');

-- Materias

    INSERT INTO materias (codigo, nombre, id_carrera, creditos)
    VALUES
    ('BD101', 'Base de Datos I', 2, 3),
```

```
    ('BD201', 'Base de Datos II', 2, 3),
    ('PG101', 'Programación I', 2, 4),
    ('PG102', 'Programación II', 1, 4),
    ('MT101', 'Cálculo I', 3, 4);

-- Inscripciones

    INSERT INTO inscripciones (id_estudiante, id_materia,
    semestre, anio, nota_final) VALUES
    (1, 1, 1, 2023, 4.5),  -- Juan Carlos en BD I, nota 4.5
    (1, 3, 1, 2023, 4.0),  -- Juan Carlos en Programación I
    (2, 1, 1, 2023, 3.8),  -- María en BD I
    (2, 3, 1, 2023, 4.2),  -- María en Programación I
    (3, 4, 4, 2023, 4.7);  -- Carlos en Programación II
```

Consultas básicas:

Comprobar:

1. Comprobar la estructura de la tabla estudiantes

   \d estudiantes;

2. Ver los datos de la tabla estudiantes

   *SELECT * FROM estudiantes;*

Consultas básicas:

1. Listar todos los estudiantes

   *SELECT * FROM estudiantes;*

2. Obtiene solo apellidos de la tabla estudiantes

   SELECT apellidos FROM estudiantes;

3. Obtiene nombre completo (nombre y apellido juntos)

   SELECT nombres, apellidos FROM estudiantes;

   Concatenando

   SELECT nombres || ' ' || apellidos FROM estudiantes;

4. Obtiene todos los identificadores y nombres de la tabla estudiantes

   SELECT id_estudiante, nombres FROM estudiantes;

5. Cambia el nombre de la columna mostrada a "Nombre_Estudiante"

   SELECT nombres AS Nombre_Estudiante FROM estudiantes;

ó

   SELECT nombres AS ‘Nombre Estudiante’ FROM estudiantes;

6. Múltiples alias

   SELECT id_estudiante AS ID, nombres AS Nombre_Estudiante, apellidos AS Apellido_Estudiante FROM estudiantes;

Ejercicios de modificación de estructura (ALTER TABLE)

1. Agregar una columna `email` de tipo `VARCHAR(150)` a la tabla `estudiantes`.

ALTER TABLE estudiantes ADD COLUMN correo_electronico *VARCHAR(150);*

2. Cambia el tipo de dato de correo_electronico a VARCHAR(200) para permitir emails más largos.

ALTER TABLE estudiantes SET DATA TYPE correo_electronico VARCHAR(200);

3. Renombra la columna correo_electronico a email.

ALTER TABLE estudiantes RENAME COLUMN correo_electronico TO email;

4. Agregar una columna `correo` de tipo `VARCHAR(150)` a la tabla `estudiantes`, para despues borrarlo.

ALTER TABLE estudiantes ADD COLUMN correo VARCHAR(150);

ALTER TABLE estudiantes DROP COLUMN correo;

Precaución: DROP COLUMN elimina permanentemente los datos. No tiene "deshacer".

Adicionamos datos en email

UPDATE estudiantes SET email='juancarlos@gmail.com' WHERE id_estudian

```
te=1;
```

Caso	MySQL	PostgreSQL
Manejo de NULL	IFNULL(email, 'Sin email')	COALESCE(email, 'Sin email')
Fecha Actual	CURDATE()	CURRENT_DATE
Extraer Año	YEAR(fecha)	EXTRACT(YEAR FROM fecha)

7. Mostrar email, pero si es NULL muestra 'Sin email'.

```
SELECT nombres, apellidos, COALESCE(email, 'Sin email') FROM estudiantes;
```

8. Muestra email, pero si es NULL muestra 'Sin email'. Cambiamos el titulo del campo Email a estudiantes sin email.

```
SELECT nombres, apellidos, COALESCE(email, 'Sin email') AS "estudiantes sin email" FROM estudiantes;
```

9. Solo usuarios llamados Juan Carlos

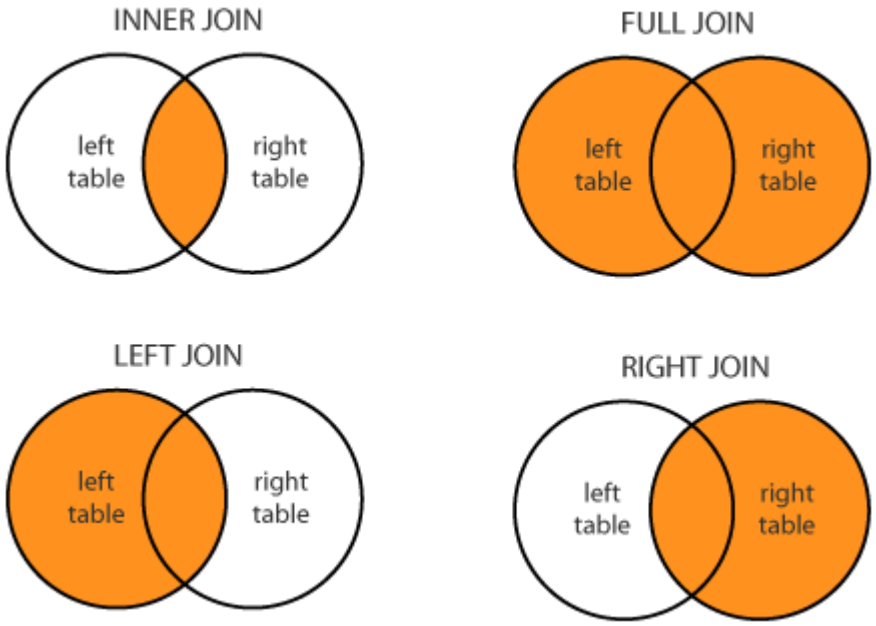
```
SELECT * FROM estudiantes WHERE nombres = 'Juan Carlos';
```

10. Calcula años desde la fecha de ingreso (si fecha_ingreso existe)

```
SELECT nombres, fecha_ingreso, EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM fecha_ingreso) AS años_transcurridos FROM estudiantes;
```

Fase 4: JOINs - Uniendo tablas

Quieres obtener...	Usa...
Solo registros que existen en ambas tablas	`INNER JOIN` ó `JOIN`
Todos los registros de la tabla principal, aunque no tengan relación	`LEFT JOIN`
Todos los registros de la tabla secundaria, aunque no tengan relación	`RIGHT JOIN`
Todos los registros de ambas tablas, completando con NULL donde no haya coincidencia	`FULL JOIN` (o `UNION` en MySQL)



Adicionamos el siguiente registro o tupla:

```
INSERT INTO estudiantes (codigo, nombres, apellidos, id_carrera,
fecha_ingreso) VALUES ('202310005', 'Pedro', 'Gómez', NULL, '2023-
02-01');
```

1. INNER JOIN

1.1 Con SELECT * FROM

```
SELECT * FROM estudiantes e INNER JOIN carreras c ON e.id_carrera
= c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT * FROM carreras c INNER JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

1.2 Seleccionando Campos

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM estudiantes e
INNER JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM carreras c
INNER JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

2. LEFT JOIN

2.1 Con SELECT * FROM

```
SELECT * FROM estudiantes e LEFT JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT * FROM  carreras c LEFT JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

2.2 Seleccionando Campos

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
```

```
FROM estudiantes e
LEFT JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM  carreras c
LEFT JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

3. RIGHT JOIN

3.1 Con SELECT * FROM

```
SELECT * FROM estudiantes e RIGHT JOIN carreras c ON e.id_carrera
= c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT * FROM carreras c RIGHT JOIN estudiantes e ON e.id_carrera
= c.id_carrera;
```

3.2 Seleccionando Campos

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM estudiantes e
RIGHT JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM carreras c
RIGHT JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

- El resultado es igual a alguno de los dos ejemplos del LEFT JOIN?
- Comparar el orden en nombres y carrera en el left y right join.
- Cual es el patrón del orden?

4. FULL JOIN

En PostgreSQL se usa FULL OUTER JOIN de forma nativa:

4.1 Con SELECT * FROM

```
SELECT * FROM estudiantes e
FULL OUTER JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Cual es la diferencia si invertimos el orden de las tablas en la sentencia?

```
SELECT * FROM carreras c
FULL OUTER JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

4.2 Seleccionando Campos

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM estudiantes e
FULL OUTER JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Comparar usando UNION ALL

```
SELECT e.nombres, e.apellidos, c.nombre AS carrera
FROM estudiantes e
FULL OUTER JOIN carreras c ON e.id_carrera = c.id_carrera;
```

Pregunta:

Cual es la diferencia entres estas dos consultas:

```
SELECT * FROM estudiantes e
```

```
LEFT JOIN carreras c ON e.id_carrera = c.id_carrera;
```

y

```
SELECT * FROM carreras c
RIGHT JOIN estudiantes e ON e.id_carrera = c.id_carrera;
```

Fase 5: Ejercicios de consolidación

Ejercicios :

1. Agregar 2 nuevas materias a la carrera de Tecnología en Desarrollo de Software
2. Inscribir al estudiante Carlos Alberto en Base de Datos I con nota 4.2
3. Listar todos los profesores de la facultad de Ingeniería
4. Calcular cuántos estudiantes ingresaron en 2023
5. Mostrar la carrera con más estudiantes

Soluciones propuestas:

```
-- Ejercicio 1: Agregar 2 nuevas materias
INSERT INTO materias (codigo, nombre, id_carrera, creditos)
VALUES
('WEB101', 'Desarrollo Web Frontend', 2, 3),
('MOV101', 'Desarrollo Móvil', 2, 4);

-- Ejercicio 2: Inscribir a Carlos Alberto en BD I
INSERT INTO inscripciones (id_estudiante, id_materia, semestre,
anio, nota_final)
VALUES (3, 1, 1, 2024, 4.2);

-- Ejercicio 3: Profesores de Ingeniería
SELECT codigo, nombres, apellidos, especialidad
FROM profesores p
JOIN facultades f ON p.id_facultad = f.id_facultad
WHERE f.nombre = 'Ingeniería';

-- Ejercicio 4: Estudiantes que ingresaron en 2023
SELECT COUNT(*) AS total_ingresos_2023
FROM estudiantes
WHERE EXTRACT(YEAR FROM fecha_ingreso) = 2023;

-- Alternativa usando DATE_PART
SELECT COUNT(*) AS total_ingresos_2023
```

```
FROM estudiantes
WHERE DATE_PART('year', fecha_ingreso) = 2023;

-- Ejercicio 5: Carrera con más estudiantes
SELECT
 c.nombre AS carrera,
 COUNT(e.id_estudiante) AS total_estudiantes
FROM carreras c
LEFT JOIN estudiantes e ON c.id_carrera = e.id_carrera
GROUP BY c.id_carrera, c.nombre
ORDER BY total_estudiantes DESC
LIMIT 1;
```

Apéndice: Comparación MySQL vs PostgreSQL

Diferencias clave en el script:

Característica	MySQL	PostgreSQL
Auto-incremento	AUTO_INCREMENT	SERIAL
Base de datos	CREATE DATABASE	CREATE DATABASE (o createdb)
Conexión	USE database	\c database (psql)
Fecha actual	CURDATE()	CURRENT_DATE
Listar tablas	SHOW TABLES	\dt (psql)
Describir tabla	DESCRIBE tabla	\d tabla (psql)
Concatenar	CONCAT()	CONCAT() o ||
Redondear	ROUND()	ROUND()
Año de fecha	YEAR()	EXTRACT(YEAR FROM) o DATE_PART
Restricciones CHECK	Soportado desde 8.0.16	Soportado nativamente

Comandos útiles de PostgreSQL (psql):

Comando	Descripción
\l	Listar todas las bases de datos
\c nombre_db	Conectar a una base de datos
\dt	Listar todas las tablas
\d nombre_tabla	Describir estructura de tabla
\du	Listar usuarios
\?	Ayuda de comandos psql
\h	Ayuda de comandos SQL

Notas importantes sobre la migración de MySQL a PostgreSQL

Cambios de sintaxis principales:

- AUTO_INCREMENT → SERIAL (o GENERATED ALWAYS AS IDENTITY en PostgreSQL 10+)
- CURDATE() → CURRENT_DATE
- SHOW TABLES → \dt (en psql) o consulta a information_schema
- DESCRIBE tabla → \d tabla o \d+ tabla
- CONCAT() funciona en ambos, pero PostgreSQL también permite el operador ||
- LIMIT funciona igual en ambos (PostgreSQL soporta LIMIT/OFFSET nativamente)
- YEAR(fecha) → EXTRACT(YEAR FROM fecha) o DATE_PART('year', fecha)
- GROUP BY requiere todas las columnas no agregadas en PostgreSQL (más estricto)
- Las cadenas en PostgreSQL son case-sensitive por defecto (usar ILIKE para búsquedas insensibles)

Características exclusivas de PostgreSQL útiles para este caso:

- CHECK constraints: Validación nativa de rangos (ej: nota_final CHECK (nota_final >= 0 AND nota_final <= 5))
- Constraints de unicidad compuesta: UNIQUE (id_estudiante, id_materia, semestre, anio)
- Índices parciales: CREATE INDEX idx_nota_alta ON inscripciones(nota_final) WHERE nota_final >= 4.0
- Vistas materializadas: Para reportes de promedio que no cambian frecuentemente
- Funciones de ventana: ROW_NUMBER(), RANK(), LEAD(), LAG() para análisis académico avanzado
- CTE (Common Table Expressions): Consultas recursivas y organizadas con WITH
- Arrays: Almacenar múltiples valores en una columna (ej: prerequisitos de materias)
- JSONB: Almacenar datos adicionales del estudiante de forma flexible