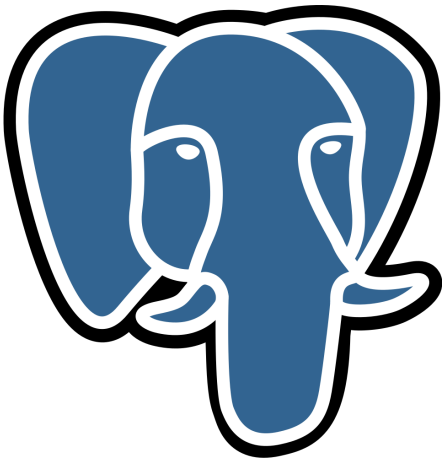
INTRODUCCIÓN A SQL CON PostgreSQL



Contenido del Curso

Módulo I: Cimientos y Diseño

- **Introducción a las Bases de Datos:** Conceptos de datos vs. información y el rol de los SGBD.
- **Diseño y Modelado:** Creación de tablas, tipos de datos y reglas de integridad.

Módulo II: Manipulación y Consulta

- **Gestión de Información (DML):** Inserción, actualización y borrado de registros.
- **Consultas y Reportes (DQL):** Filtrado, ordenación y funciones de agregación.

Módulo III: Análisis Avanzado

- **Relaciones Avanzadas:** Combinación de tablas mediante el uso de *Joins*.
- **Optimización y Estructura:** Uso de subconsultas y creación de vistas.

Módulo IV: Integridad y Administración

- **Fiabilidad de los Datos:** El marco de las Propiedades ACID.
- **Gestión mediante Terminal:** Uso de herramientas de consola (psql) y comandos esenciales.
- **Portabilidad y Backups en PostgreSQL:** Uso de herramientas de línea de comandos para el respaldo, restauración e intercambio de datos.

Capítulo 0: Introducción a las Bases de Datos

Este capítulo sienta las bases teóricas necesarias para comprender la importancia de la organización de los datos y el rol de las herramientas tecnológicas en el mundo moderno.

0.1. El Concepto de Base de Datos (BD)

Una **Base de Datos** es una colección organizada de información estructurada, almacenada electrónicamente de forma que pueda ser consultada, recuperada y gestionada rápidamente.

- **Dato vs. Información**: Un dato es una unidad mínima (como un número o una palabra); la información es el resultado de procesar y organizar esos datos para que tengan sentido.
- **Caso de Uso**: Imagina una biblioteca; los libros son los datos, pero el catálogo que te permite buscarlos por autor o género es el sistema que organiza la base de datos.

0.2. ¿Qué es un SGBD o Motor de Base de Datos?

Los datos por sí solos no sirven de nada si no tenemos una herramienta para interactuar con ellos. El **Sistema de Gestión de Bases de Datos (SGBD)**, conocido comúnmente como **Motor**, es el software que actúa como intermediario.

- **Definición**: Es el software que permite a los usuarios **definir** (crear estructuras), **manipular** (insertar o cambiar datos) y **controlar** (gestionar seguridad) la información.
- **Importancia**: El motor se encarga de tareas complejas como asegurar que dos personas no modifiquen el mismo dato al mismo tiempo y proteger la información contra fallos del sistema.

0.3. *¿Qué significan las siglas SQL?*

SQL no es un lenguaje de programación común (como Python o Java), sino un lenguaje de comunicación diseñado para bases de datos. Sus siglas significan:

- S (Structured): Estructurado. Indica que los datos deben seguir un orden y una arquitectura rígida (tablas).
- Q (Query): Consulta. Es la función principal del lenguaje: realizar preguntas al motor para recuperar información específica.
- L (Language): Lenguaje. Es el conjunto de reglas y sintaxis universal que permite que el usuario y el servidor se entiendan.

0.4. Tipos de Bases de Datos: Relacionales y No Relacionales

No todas las bases de datos se organizan igual. Dependiendo de la necesidad, elegimos un modelo diferente:

A. Bases de Datos Relacionales (SQL)

Son el estándar más utilizado en la industria y el foco principal de este curso.

- **Estructura**: Organizan los datos en **Tablas** compuestas por filas y columnas.
- **Relaciones**: Los datos se conectan entre sí (por ejemplo, un cliente está relacionado con sus compras).
- **Integridad**: Son ideales cuando se requiere una consistencia total de los datos (como en un banco).

B. Bases de Datos No Relacionales (NoSQL)

Diseñadas para manejar datos que no tienen una estructura fija o que son masivos.

- **Flexibilidad**: Pueden almacenar documentos, listas o grafos de forma dinámica.
- **Uso**: Muy comunes en aplicaciones de redes sociales o análisis de datos a gran escala.

0.5. Arquitectura de Conexión: Usuario y Cliente

Es fundamental entender los roles involucrados en la comunicación con la base de datos:

1. **El Usuario**: Es la persona que requiere acceso a la información (administrador, programador o analista).

2. **El Cliente**: Es la interfaz o aplicación que el usuario utiliza para enviar órdenes (por ejemplo, una consola de comandos o una aplicación web).

3. **El Servidor (Motor)**: Es el software (SGBD) que reside en una computadora, recibe las órdenes del cliente, las procesa y devuelve los resultados.

0.6. Líderes del Mercado de Bases de Datos Relacionales

En la industria, existen diferentes motores que implementan el estándar SQL. Conocerlos te ayudará a identificar qué herramientas se usan en el mundo real:

- **Motores Comerciales (Pago)**:
  - **Oracle Database**: El gigante de la industria, usado en la mayoría de las empresas de la lista Fortune 500.
  - **Microsoft SQL Server**: Muy potente y con gran integración en entornos empresariales de Windows.
  - **IBM DB2**: Un motor histórico, extremadamente fiable para grandes volúmenes de transacciones.
- **Motores Open Source (Gratuitos)**:
  - **PostgreSQL**: Considerado el motor libre más avanzado y fiel a los estándares.
  - **MySQL**: El más popular para desarrollo web; es el motor detrás de WordPress y Facebook.
  - **SQLite**: Un motor ligero que se encuentra dentro de casi todos los teléfonos móviles y navegadores del mundo.

Capítulo 1: Diseño y Modelado de Datos

Este capítulo enseña a transformar un problema del mundo real en una estructura lógica que el motor de la base de datos pueda procesar eficientemente.

1.1. Presentación del caso de estudio: TechNova Solutions

Actualmente, TechNova Solutions sufre de pérdida de información y duplicidad de datos porque utilizan hojas de cálculo desordenadas. El reto del capítulo consiste en diseñar una estructura robusta que no solo almacene sus activos, sino que permita registrar el flujo completo de su negocio a través de **seis pilares fundamentales**:

1. **Clientes:** Personas o empresas que adquieren los servicios y productos.

2. **Productos:** El inventario detallado de equipos tecnológicos disponibles para la venta.

3. **Categorías:** Clasificaciones lógicas para organizar el inventario (ej. Laptops, Servidores).

4. **Empleados:** Los responsables de gestionar y realizar las ventas.

5. **Pedidos:** Registro de cada transacción comercial. Cada pedido debe estar vinculado obligatoriamente a un **cliente** (quien compra) y a un **vendedor** (el empleado que gestiona la venta), además de contener la fecha y el monto total.

6. **Detalles de Pedido (Pivote):** La pieza clave que permite vincular múltiples productos a un solo pedido, registrando cantidades y precios unitarios históricos para asegurar que la información contable sea inalterable.

1.2. De la realidad a las tablas

El primer paso es entender la nomenclatura que utiliza el SQL para representar la realidad:

- **Tabla:** Es la estructura de almacenamiento que representa un objeto (ej. la tabla `clientes`).
- **Columna (Atributo):** Representa cada característica del objeto (ej. el `nombre` del cliente o su `identificacion_fiscal`).
- **Fila (Tupla):** Es el registro de un objeto individual (ej. los datos de un cliente específico en una fila).

1.3. Identificación de claves (Keys)

Para que los datos de TechNova sean confiables y no se mezclen, cada registro debe ser único:

- **Clave Primaria (Primary Key):** Es la columna (o conjunto de ellas) que identifica de forma única a cada fila. No puede repetirse ni ser nula.

- **Clave Foránea (Foreign Key):** Es una columna que establece un vínculo con otra tabla para organizar los datos (ej. vincular un **producto** con su **categoría** correspondiente, como "Electrónica").

1.4. Guía de conectividad: ¿Cómo se relacionan las tablas?

En el diseño relacional, las tablas no están aisladas; se conectan entre sí para dar sentido al flujo del negocio. Existen tres tipos fundamentales de relaciones que debemos identificar antes de escribir cualquier código SQL.

1.4.1. Relación uno a muchos (1:N)

Es la relación más común en el mundo de las bases de datos. Ocurre cuando un registro de una **Tabla A** puede estar relacionado con varios registros de una **Tabla B**, pero cada registro de la **Tabla B** solo puede pertenecer a un único registro de la **Tabla A**.

Ejemplos en TechNova Solutions:

- **Clientes y Pedidos:** Un **Cliente** puede realizar muchos **Pedidos** a lo largo del tiempo. Sin embargo, cada **Pedido** específico (identificado por su número de factura) pertenece a un único **Cliente**.

- **Empleados y Pedidos:** Un **Empleado** (vendedor) puede gestionar múltiples **Pedidos**. No obstante, cada **Pedido** es procesado y asignado a un único **Vendedor** responsable.

- **Categorías y Productos:** Una **Categoría** (como "Periféricos") puede albergar muchos **Productos** (teclados, ratones, webcams). Pero cada **Producto** individual está clasificado bajo una única **Categoría** principal para mantener el orden del inventario.

1.4.2. Relación muchos a muchos (N:M)

Ocurre cuando múltiples registros de la **Tabla A** se vinculan con múltiples registros de la **Tabla B**. A diferencia de las anteriores, esta relación no se puede crear directamente en SQL, ya que causaría duplicidad y desorden. Para resolverlo, se requiere una **Tabla Intermedia** (también llamada tabla pivote o de unión).

- **En TechNova:** Un **Pedido** puede contener muchos **Productos**, y un **Producto** puede estar presente en muchos **Pedidos**.

- **La Solución (Tabla Pivote):** Creamos la tabla `detalles_pedido`. Esta tabla actúa como un puente que registra la combinación exacta de cada producto dentro de cada pedido, permitiendo además guardar el precio histórico de la venta.

1.4.3. Relación uno a uno (1:1)

Menos frecuente, se usa para separar datos por seguridad o rendimiento. Un registro de la tabla A solo puede tener un registro coincidente en la tabla B.

- **Ejemplo:** Si TechNova decidiera separar la información privada de los empleados, podría existir una tabla `empleados` y otra `expedientes_confidenciales`, vinculadas por un único ID compartido.

1.4.4. Criterios para definir relaciones

Para decidir qué tipo de conexión aplicar al modelar una base de datos, utiliza siempre la técnica de la **"Pregunta en Doble Sentido"**:

1. **Hacia adelante:** ¿Cuántos registros de la **Tabla B** puede tener un solo registro de la **Tabla A**?

2. **Hacia atrás:** ¿Cuántos registros de la **Tabla A** puede tener un solo registro de la **Tabla B**?

Resultados del análisis:

- Si la respuesta es **Uno** en un lado y **Muchos** en el otro: Es una relación **1:N**. No necesitas tablas extra, solo una Clave Foránea en la tabla "Muchos".

- Si la respuesta es **Muchos** en ambos sentidos: Es una relación **N:M**. Debes aplicar la **Regla de la Tabla Pivote** (como nuestra tabla `detalles_pedido`) para conectar ambas entidades correctamente.

Capítulo 2: Estructura y Definición de Datos (DDL)

El lenguaje **DDL (Data Definition Language)** es el conjunto de comandos que permite crear, modificar y eliminar las estructuras de una base de datos. En este capítulo se explica cómo transformar el diseño lógico en objetos reales dentro del motor.

2.1. Reglas de nomenclatura (naming conventions)

Antes de nombrar cualquier objeto (base de datos, tabla o columna), es fundamental respetar las restricciones técnicas del motor para evitar errores de sintaxis:

- **Sin espacios:** Los identificadores no pueden contener espacios en blanco. Se utiliza el guión bajo (_) para separar palabras (estilo *snake_case*).
  - *Correcto:* `sistema_ventas`
  - *Incorrecto:* `sistema ventas`

- **Inicio válido:** Todo nombre debe comenzar obligatoriamente con una letra (a-z) o un guión bajo (_). Nunca puede comenzar con un número.

- **Caracteres permitidos:** Utiliza únicamente letras, números y guiones bajos. Evita el uso de tildes, la letra "ñ" y símbolos especiales (como `-`, `$`, `@`, `/`).

- **Longitud máxima:** En PostgreSQL, los nombres tienen un límite técnico de 63 caracteres. Se recomienda ser conciso y descriptivo.

- **Sensibilidad a mayúsculas:** Aunque SQL permite mayúsculas, PostgreSQL convierte automáticamente los nombres a minúsculas a menos que se usen comillas dobles. Por convención y facilidad, escribiremos siempre los nombres de los objetos en minúsculas.

2.2. Estructura de la base de datos y accesos

Antes de definir tablas, es necesario crear el entorno donde residirán los datos y quién tendrá permiso para acceder a ellos.

- **Crear una Base de Datos:**

SQL
```
CREATE DATABASE sistema_ventas;
```

- **Creación de Usuarios y Permisos:**

En entornos profesionales, no se recomienda usar la cuenta de administrador para tareas diarias.

SQL
```
CREATE USER gestor_datos WITH PASSWORD 'clave_segura';
GRANT ALL PRIVILEGES ON DATABASE sistema_ventas TO
gestor_datos;
```

2.3. Tipos de datos fundamentales

Cada columna debe tener un tipo de dato asignado para que el motor gestione la información de forma eficiente.

Tipo de Dato	Descripción	Uso Común
VARCHAR(n)	Texto de longitud variable (máximo *n*).	Nombres, correos.
INTEGER	Números enteros de 32 bits.	Cantidades, edades.
DECIMAL(p,s)	Precisión exacta (*p* dígitos, *s* decimales).	Precios, salarios.
BOOLEAN	Valores lógicos (**TRUE** o **FALSE**).	Estados (activo, pagado).

2.3.1 Tipos de datos de fecha y hora

Tipo de Dato	Formato Estándar	Función por Defecto	Uso Sugerido
DATE	`YYYY-MM-DD`	`CURRENT_DATE`	Fechas de pedidos o registros de clientes.
TIME	`HH:MM:SS`	`CURRENT_TIME`	Horarios de turnos de empleados.
TIMESTAMP	`YYYY-MM-DD HH:MM:SS`	`CURRENT_TIMESTAMP`	Auditoría de ventas (momento exacto del pago).

Nota técnica: El tipo **BOOLEAN** se utiliza en este manual aprovechando su implementación nativa en PostgreSQL. En otros motores, este tipo puede representarse mediante valores numéricos (0 y 1) o tipos de bits.

2.4. Identificadores automáticos en la industria

En este curso hemos optado por utilizar el estándar moderno **`GENERATED ALWAYS AS IDENTITY`**, ya que es la norma oficial de SQL y la práctica recomendada en entornos profesionales actuales. Sin embargo, al trabajar en proyectos reales o con sistemas más antiguos, te encontrarás con otras variantes para generar identificadores automáticos.

2.5. Anatomía de una columna

Antes de proceder a la creación de nuestras tablas, es vital entender el orden jerárquico que debe llevar cada línea de código. En SQL, el motor lee de izquierda a derecha siguiendo este esquema obligatorio:

1. **Nombre de la Columna:** El identificador (ej. `monto_total`).
2. **Tipo de Dato:** Define qué se puede guardar (ej. `DECIMAL`).
3. **Valor por Defecto (`DEFAULT`):** La red de seguridad. Si el usuario olvida este dato, el sistema lo llena solo (ej. `DEFAULT 0`).

4. **Restricciones de Integridad:** Las reglas que el dato debe cumplir para ser aceptado (ej. `NOT NULL` o `CHECK`).

2.6. Formas de declarar claves foráneas (Foreign Keys)

En SQL, existen dos caminos para establecer la relación entre una Tabla A y una Tabla B. Ambas son correctas y aceptadas por la mayoría de los motores (PostgreSQL, MySQL, SQL Server), pero se usan en contextos distintos:

A. Declaración Directa (Abreviada) Se escribe en la misma línea donde defines la columna. Es ideal para scripts rápidos y para mantener el código limpio.

SQL
```
categoria_id INT REFERENCES categorias(categoria_id)
```

- **Cuándo usarla:** Cuando quieres brevedad y no necesitas darle un nombre específico a la regla de integridad.

B. Declaración con CONSTRAINT (Explícita) Se define generalmente al final de la tabla, después de todas las columnas.

SQL
```
CONSTRAINT fk_productos_categorias FOREIGN KEY (categoria_id)
REFERENCES categorias(categoria_id)
```

- **Cuándo usarla:** En entornos profesionales, ya que permite nombrar la restricción (`fk_productos_categorias`). Esto facilita el mantenimiento de la base de datos y hace que los mensajes de error sean mucho más claros si algo falla.

2.7. Creación de tablas (CREATE TABLE)

Antes de escribir código, debemos entender que una **Entidad** es cualquier objeto o concepto del mundo real relevante para **TechNova Solutions** del cual necesitamos guardar información (por ejemplo: un Producto, un Cliente o una Categoría). En el diseño lógico, las entidades son los "sustantivos" de nuestro sistema.

El comando **`CREATE TABLE`** es el encargado de transformar esas entidades en estructuras físicas dentro de la base de datos. Al crear estas tablas, es fundamental respetar el orden de ejecución debido a las **Claves Foráneas**: las tablas que representan entidades independientes o "maestras" (como **Categorías**) deben existir antes que aquellas que dependen de ellas (como **Productos**).

A. Tabla de categorías (Tabla Base)

SQL

```
CREATE TABLE categorias (
   categoria_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY
KEY,
   nombre VARCHAR(50) NOT NULL UNIQUE
);
```

B. Tabla de productos (con clave foránea y boolean)

```
SQL
CREATE TABLE productos (
   producto_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY
KEY,
   nombre VARCHAR(100) NOT NULL,
   descripcion VARCHAR(500),
   precio DECIMAL(10,2) CHECK (precio > 0),
   stock INTEGER DEFAULT 0,
   en_stock BOOLEAN DEFAULT TRUE,
   fecha_alta DATE DEFAULT CURRENT_DATE,
   -- Definición del vínculo directa (sin CONSTRAINT)
   categoria_id INTEGER NOT NULL REFERENCES
categorias(categoria_id)
);
```

C. Tablas de gestión (clientes y empleados)

SQL

```
CREATE TABLE clientes (
   cliente_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY
KEY,
   identificacion_fiscal VARCHAR(20) NOT NULL UNIQUE,
   nombre VARCHAR(150) NOT NULL,
   apellido VARCHAR(150) NOT NULL,
   email VARCHAR(100) UNIQUE
);

CREATE TABLE empleados (
   empleado_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY
KEY,
   nombre VARCHAR(100) NOT NULL,
   apellido VARCHAR(100) NOT NULL,
   cargo VARCHAR(50),
   salario DECIMAL(10,2) CHECK (salario > 0)
);
```

D. Tablas de transacciones (pedidos y detalles)

Finalmente, necesitamos las tablas donde se registrará la actividad del negocio. Observa el orden: primero creamos la tabla principal (`pedidos`) y luego la tabla que depende de ella (`detalles`).

SQL

```
CREATE TABLE pedidos (
   pedido_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
   fecha_pedido DATE DEFAULT CURRENT_DATE,
   monto_total DECIMAL(10,2) DEFAULT 0,
   pagado BOOLEAN DEFAULT FALSE,
   -- Definición de vínculos directa (sin CONSTRAINT)
   cliente_id INTEGER NOT NULL REFERENCES
clientes(cliente_id),
   empleado_id INTEGER NOT NULL REFERENCES
empleados(empleado_id)
);
```

```
CREATE TABLE detalles_pedido (
   detalle_id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY
KEY,
   pedido_id INTEGER NOT NULL REFERENCES pedidos(pedido_id),
   producto_id INTEGER NOT NULL REFERENCES
productos(producto_id),
   cantidad INTEGER NOT NULL CHECK (cantidad > 0),
   precio_unitario DECIMAL(10,2) NOT NULL
);
```

2.8. Modificación de estructuras (ALTER TABLE)

Permite ajustar la estructura sin necesidad de borrar la tabla y perder los datos.

- **Agregar una columna:**

SQL
```
ALTER TABLE productos ADD COLUMN en_promocion BOOLEAN DEFAULT
FALSE;
```

- **Cambiar el nombre de una columna:**

SQL
```
ALTER TABLE productos RENAME COLUMN nombre TO nombre_producto;
```

2.9. Eliminación de bbjetos (DROP TABLE)

Elimina permanentemente una tabla y todo su contenido.

SQL
```
DROP TABLE productos;
```

Advertencia: Esta acción es irreversible.

2.10. Restricciones de integridad (Constraints)

Reglas esenciales para asegurar la coherencia:

- **NOT NULL:** Impide valores vacíos.
- **UNIQUE:** Evita duplicados (ej. email, identificación fiscal).
- **CHECK:** Valida condiciones (ej. `precio > 0`).

Capítulo 3: Manipulación de datos (DML)

En este capítulo se estudia el **DML** (*Data Manipulation Language*), el conjunto de sentencias que permiten gestionar los registros dentro de las tablas. Mientras que el DDL (Capítulo 2) define la arquitectura, el DML se encarga de dar vida a la base de datos gestionando su información.

3.1. Inserción de Registros (INSERT INTO)

Una base de datos vacía es solo una estructura inerte. Para que TechNova Solutions pueda operar, necesitamos poblarla con información real: productos, clientes y empleados. La sentencia SQL encargada de esta tarea es `INSERT INTO`.

3.1.1. Sintaxis Básica

La instrucción le dice al motor: *"En esta tabla, dentro de estas columnas, guarda estos valores"*.

SQL
```
INSERT INTO nombre_tabla (columna1, columna2, ...)
VALUES (valor1, valor2, ...);
```

Regla de Oro: El orden de los valores en el paréntesis `VALUES` debe coincidir exactamente con el orden de las columnas que listaste.

3.1.2. Poblando las tablas maestras

En el Capítulo 2 definimos tablas que no dependen de nadie (Categorías, Clientes, Empleados). Debemos empezar por ellas para evitar errores de integridad.

A. Insertando Categorías Como definimos `categoria_id` como `SERIAL`, no necesitamos escribir el número; PostgreSQL lo generará automáticamente (1, 2, 3...).

SQL
```
-- Insertar una sola fila
INSERT INTO categorias (nombre)
VALUES ('Laptops');

-- Insertar múltiples filas a la vez (Más eficiente)
INSERT INTO categorias (nombre)
```

```
VALUES
  ('Periféricos'),
   ('Monitores'),
   ('Componentes');
```

B. Registrando al Personal y Clientes Vamos a dar de alta al primer vendedor y a un cliente corporativo para preparar el terreno.

SQL

```
-- Ingreso de los primeros empleados
INSERT INTO empleados (nombre, apellido, cargo, salario)
VALUES ('Carlos', 'Ruiz', 'Vendedor', 2500.00);

INSERT INTO empleados (nombre, apellido, cargo, salario)
VALUES ('Andrea', 'Martínez', 'Gerente', 6500.00);

-- Ingreso de un cliente (Note que usamos comillas simples
para textos)
INSERT INTO clientes (identificacion_fiscal, nombre, email)
VALUES ('20-12345678-9', 'Consultora Tecnológica Sur',
'contacto@surtech.com');
```

3.1.3. Insertando con Claves Foráneas (Relaciones)

Aquí es donde el diseño relacional cobra vida. Para registrar un **Producto**, necesitamos saber a qué **Categoría** pertenece.

Supongamos que:

- `Laptops` recibió el ID **1**.
- `Periféricos` recibió el ID **2**.

Al insertar los productos, usamos esos números en la columna `categoria_id` para crear el vínculo.

SQL

```
INSERT INTO productos (nombre, descripcion, precio, stock,
categoria_id)
VALUES
  ('MacBook Pro M3', 'Potencia y portabilidad para
profesionales.', 1500.00, 10, 1),
   ('Mouse Logitech MX', 'Ergonómico e inalámbrico con alta
precisión.', 85.50, 50, 2),
```

```
   ('Teclado Mecánico', 'Interruptores azules y
retroiluminación RGB.', 120.00, 30, 2);
```

¿Qué pasa si me equivoco? Si intentas poner `categoria_id = 99` y esa categoría no existe en la tabla `categorias`, SQL bloqueará la inserción con un error de **violación de clave foránea**. Esto protege la calidad de los datos de TechNova.

3.2. Modificación de Registros (UPDATE)

En el dinamismo de **TechNova Solutions**, la información nunca es estática. Los precios de los componentes fluctúan, los empleados reciben ascensos y el inventario cambia minuto a minuto. SQL nos ofrece el comando `UPDATE` para modificar datos que ya existen en nuestras tablas sin necesidad de borrarlos y volverlos a crear.

3.2.1. Sintaxis Fundamental

La instrucción funciona bajo una lógica muy precisa: *"Ve a esta tabla, busca las filas que cumplan esta condición y cambia estos valores"*.

SQL
```
UPDATE nombre_tabla
SET columna_a_cambiar = nuevo_valor
WHERE condicion_filtro;
```

La Regla de Oro del UPDATE: Jamás olvides la cláusula `WHERE`. Si ejecutas un `UPDATE` sin filtrar, el motor asumirá que quieres cambiar **TODAS** las filas de la tabla.

- *Con WHERE:* "Cambiar el precio del Mouse a $20".
- *Sin WHERE:* "Poner el precio de TODOS los productos de la tienda a $20".

3.2.2. Escenarios Prácticos en TechNova

A continuación, veremos tres situaciones reales que ocurren en la gestión diaria de la empresa y cómo resolverlas con SQL.

Caso A: Corrección y Ascensos (Tabla `empleados`) Carlos Ruiz, nuestro primer vendedor (ID 1), ha tenido un desempeño excelente y ha sido promovido. Necesitamos actualizar su cargo y su salario.

SQL

```
-- Actualizamos dos columnas al mismo tiempo
UPDATE empleados
SET cargo = 'Gerente de Ventas',
   salario = 3200.00
WHERE empleado_id = 1;
```

Interpretación: SQL busca al empleado con ID 1 y sobrescribe únicamente los campos de cargo y salario, dejando su nombre intacto.

Caso B: Gestión de Inventario (Tabla `productos`) Acaba de llegar un nuevo lote de mercadería al almacén. Han ingresado 20 unidades más del "Mouse Logitech MX" (que tiene el ID 2). En lugar de calcular el total mentalmente, dejamos que la base de datos haga la matemática.

SQL
```
-- Sumamos al valor actual
UPDATE productos
SET stock = stock + 20
WHERE producto_id = 2;
```

Interpretación: Si antes había 50, ahora el valor será 70. Esta forma es segura porque respeta el valor previo.

Caso C: Ajustes Masivos de Precios (Tabla `productos`) Debido a la inflación, la gerencia decide aumentar un 10% el precio de todos los productos que pertenecen a la categoría "Periféricos" (Supongamos que es la `categoria_id = 2`).

SQL
```
UPDATE productos
SET precio = precio * 1.10
WHERE categoria_id = 2;
```

Interpretación: Este comando afectará a múltiples filas (mouses, teclados, webcams) en una sola ejecución, pero dejará intactos los precios de las Laptops (Categoría 1).

3.2.3. Buenas Prácticas de Seguridad

Antes de ejecutar un cambio importante, los profesionales siguen el protocolo de **"Ver antes de Tocar"**.

1. **Paso 1 (Verificación):** Primero ejecutamos un `SELECT` con la misma condición para asegurarnos de qué vamos a modificar.

SQL
```
SELECT * FROM productos WHERE categoria_id = 2;
-- ¿Son estos los productos que quiero subir de precio? Sí.
```

2. **Paso 2 (Ejecución):** Recién entonces lanzamos el `UPDATE`.

3.3. Eliminación de Registros (DELETE)

En la gestión de una empresa como **TechNova Solutions**, eliminar información es una operación crítica pero necesaria. Ya sea porque se cargó un dato por error o porque cierta información ha quedado legalmente obsoleta, SQL nos proporciona el comando `DELETE` para retirar filas de nuestras tablas de forma permanente.

3.3.1. Sintaxis Básica

La instrucción es directa y poderosa: *"De esta tabla, borra las filas que cumplan esta condición"*.

SQL
```
DELETE FROM nombre_tabla
WHERE condicion;
```

Advertencia de Seguridad: Al igual que con el `UPDATE`, la cláusula `WHERE` es vital. Si ejecutas `DELETE FROM clientes;` sin condición, borrarás **toda** la cartera de clientes de la empresa en un instante. No existe una "Papelera de Reciclaje" estándar en SQL.

3.3.2. Casos Prácticos en TechNova

Caso A: Eliminación Simple (Corrección de Errores) Supongamos que el equipo de marketing nos pide crear una categoría para "Juguetes", pero al minuto se dan cuenta de que fue un error y debemos borrarla.

Para practicar esto, primero vamos a crear ese error intencionalmente y luego lo corregiremos:

SQL
```
-- 1. Creamos el dato erróneo
INSERT INTO categorias (nombre) VALUES ('Juguetes');

-- 2. Ahora lo eliminamos de la base de datos
DELETE FROM categorias
```

```
WHERE nombre = 'Juguetes';
```

Resultado: Verás un mensaje que dice `DELETE 1`, confirmando que la fila fue eliminada exitosamente.

Caso B: El Conflicto de Integridad (La Protección del Motor) Aquí es donde el diseño relacional que hicimos en el **Capítulo 2** demuestra su valor. Imagina que intentamos borrar al empleado "Carlos Ruiz" (ID 1), quien ya ha gestionado ventas en la empresa.

SQL
```
DELETE FROM empleados
WHERE empleado_id = 1;
```

Resultado Esperado:

Plaintext
```
ERROR: update or delete on table "empleados" violates foreign
key constraint "fk_pedidos_empleados" on table "pedidos"
Detail: Key (empleado_id)=(1) is still referenced from table
"pedidos".
```

¿Qué acaba de pasar? PostgreSQL nos ha protegido. El motor detectó que si borra a Carlos, los pedidos que él gestionó quedarían "huérfanos" (sin vendedor asociado). Por lo tanto, **bloquea** la eliminación para mantener la consistencia de los datos.

Para poder borrar a Carlos, primero tendríamos que borrar sus pedidos o reasignarlos a otro empleado.

3.3.3. Buenas Prácticas: El Protocolo de Borrado

En entornos de producción, los administradores de base de datos siguen reglas estrictas para evitar desastres:

1. **Verificación Previa ("Select antes de Delete"):** Antes de borrar, siempre consulta qué vas a eliminar para confirmar que tu filtro es correcto.

SQL
```
-- Primero verifico visualmente
SELECT * FROM categorias WHERE nombre = 'Hogar';

-- Si el resultado es correcto, procedo a borrar
```

```
DELETE FROM categorias WHERE nombre = 'Hogar';
```

2. **Borrado Lógico (Soft Delete):** En muchas empresas reales, rara vez se usa `DELETE`. Se prefiere agregar una columna llamada `activo` (TRUE/FALSE).
    a. *En vez de borrar:* `DELETE FROM empleados...`
    b. *Se actualiza:* `UPDATE empleados SET activo = FALSE WHERE empleado_id = 1;` Esto mantiene el historial de ventas intacto, pero el empleado ya no aparece en los listados activos del sistema.

3.4. Integridad y Control de Transacciones

En el mundo real, una operación comercial rara vez implica modificar una sola tabla. En **TechNova Solutions**, cuando se concreta una venta, suceden una serie de pasos encadenados que dependen uno del otro.

3.4.1. El Desafío: La Cadena de Eventos

Para que una venta sea válida, el sistema debe realizar tres acciones obligatorias:

1. **Generar el Pedido General:** Crear el registro con el cliente y el total (para obtener un número de pedido).
2. **Listar los Productos:** Registrar qué ítems específicos pertenecen a ese número de pedido.
3. **Descontar el Inventario:** Restar la cantidad vendida del stock en el almacén.

El Problema: Imagina que el sistema completa los pasos 1 y 2, pero justo antes del paso 3 **se corta la luz** en el servidor o se cae la red.

- **Resultado:** Tenemos un pedido cobrado y registrado, pero el sistema de stock dice que todavía tenemos la mercadería física. Esto genera un **descuadre de inventario** grave (datos corruptos).

3.4.2. La Solución: Transacciones (Atomicidad)

Para evitar esto, SQL utiliza **Transacciones**. Una transacción encapsula todos estos pasos en una "burbuja de seguridad" bajo la regla del **"Todo o Nada"**.

- **Funcionamiento:** Si **todos** los pasos se completan bien, se guardan. Si falla **uno solo** (o se apaga el servidor a la mitad), el sistema borra automáticamente todo lo que se había avanzado, volviendo al estado original como si nada hubiera pasado.

3.4.3. Comandos de Control (TCL)

Por defecto, SQL guarda cada línea que escribes al instante (modo *Autocommit*). Para activar la protección manual, usamos tres comandos:

- **`BEGIN`**: Inicia la protección. Le dice al motor: *"Pon en pausa el guardado automático. Todo lo que haga ahora es temporal"*.
- **`COMMIT`**: Confirma el éxito. Le dice al motor: *"Todo salió perfecto. Guarda permanentemente los cambios"*.
- **`ROLLBACK`**: Cancela todo. Le dice al motor: *"Hubo un error. Deshaz todo lo que hice desde el BEGIN"*.

3.4.4. Ejemplo Práctico: La Venta Segura en TechNova

Vamos a simular que el cliente "Consultora Sur" compra una "MacBook Pro".

SQL

```
-- 1. Iniciamos la transacción
BEGIN;

-- 2. Paso A: Registramos el PEDIDO GENERAL
-- Al ser la primera venta, el sistema le asignará el ID 1.
INSERT INTO pedidos (cliente_id, empleado_id, monto_total,
fecha_pedido)
VALUES (1, 1, 1500.00, '2023-11-15');

-- 3. Paso B: Agregamos los PRODUCTOS a ese pedido
-- IMPORTANTE: Usamos el ID 1 porque es el pedido que acabamos
de crear arriba.
INSERT INTO detalles_pedido (pedido_id, producto_id, cantidad,
precio_unitario)
VALUES (1, 1, 1, 1500.00);

-- 4. Paso C: Actualizamos el INVENTARIO
UPDATE productos
SET stock = stock - 1
WHERE producto_id = 1;

-- 5. Confirmación Final
COMMIT;
```

3.4.5. Análisis de Escenarios (¿Qué pasa si...?)

Escenario 1: El Apagón (Fallo del Sistema) Si se corta la luz después del paso 4 pero **antes** de llegar al paso 5 (`COMMIT`), la transacción queda incompleta.

- **Reacción del Sistema:** Al reiniciarse, la base de datos detecta que esa transacción nunca se confirmó y ejecuta un borrado automático de los pasos 2, 3 y 4. **La base de datos queda limpia, sin pedidos a medias.**

Escenario 2: Error de Lógica (Stock Insuficiente) Si en el paso 4 descubrimos que el stock es 0 y no podemos vender, nosotros (o el programa) podemos cancelar la operación voluntariamente:

SQL

```
-- Si detectamos error:
ROLLBACK;
-- El sistema borra el pedido y el detalle creados en los pasos 2 y 3.
```

3.5. Protocolos de Seguridad y Buenas Prácticas

Para trabajar en entornos profesionales, se establecen los siguientes protocolos obligatorios:

1. **Validación Previa con SELECT:** Antes de ejecutar un `UPDATE` o `DELETE`, realice un `SELECT` utilizando exactamente el mismo filtro `WHERE`. Si los registros mostrados son los correctos, proceda con la modificación.
2. **Gestión de Valores Nulos:** El valor `NULL` representa la ausencia de información. Para filtrarlos, utilice la sintaxis **`IS NULL`** o **`IS NOT NULL`**. El operador de igualdad (`= NULL`) no es válido en el estándar SQL.
3. **Nota sobre el Auto-commit:** Por estándar, `COMMIT` y `ROLLBACK` son universales, pero muchas herramientas tienen activado el "Auto-commit" (guardado automático).
4. **Control de Integridad:** El motor impedirá operaciones que violen las reglas definidas en el DDL, como intentar borrar un cliente que posee pedidos registrados (Integridad Referencial).

Capítulo 4: Lenguaje de Consulta de Datos (DQL)

En este capítulo abandonamos la modificación de registros para centrarnos en su explotación. Utilizaremos el **DQL** (*Data Query Language*), el subconjunto de SQL cuya única misión es **recuperar información** de la base de datos sin alterarla.

4.1. ¿Qué es el DQL?

El **DQL (Data Query Language)** es el subconjunto de SQL dedicado exclusivamente a la **lectura y recuperación de datos**.

- **Naturaleza "Solo Lectura":** A diferencia de `INSERT` o `DELETE`, las instrucciones DQL son seguras. No importa cuántas veces ejecutes una consulta o lo compleja que sea; los datos en el disco duro permanecen intactos.
- **El Puente de Negocio:** Es la herramienta que permite responder preguntas como: *"¿Tenemos stock suficiente de monitores?"* o *"¿Quiénes son los empleados contratados este año?"*.
- **La Sentencia Madre:** Todo el DQL se basa en una única y poderosa instrucción: **`SELECT`**.

4.2. Anatomía de una Consulta (SELECT)

Para obtener datos, debemos indicar al motor qué columnas queremos y de qué tabla provienen.

Sintaxis Base

SQL

```
SELECT columna1, columna2
FROM nombre_de_tabla;
```

Caso TechNova 1: Proyección Específica

El equipo de Marketing necesita una lista de precios rápida. Solo les interesa el nombre del producto y su costo.

SQL

```
SELECT nombre, precio
FROM productos;
```

Resultado: Una tabla limpia con solo esas dos columnas, ignorando IDs o stock.

Caso TechNova 2: Selección Total (*)

Como administrador, necesitas verificar la carga completa de un nuevo empleado para ver todos sus datos (ID, cargo, fecha, etc.).

SQL
```
SELECT * FROM empleados;
```

Nota: El `*` trae todas las columnas. Úsalo con precaución en tablas muy grandes.

4.3. Filtrado de Datos (La Cláusula WHERE)

Rara vez necesitamos ver *todos* los registros. La cláusula `WHERE` actúa como un filtro que decide qué filas se muestran basándose en una condición.

Sintaxis

El `WHERE` siempre va **después** del `FROM`.

Filtros Numéricos (Stock Crítico)

El Gerente de Inventario necesita saber qué productos tienen menos de 10 unidades para reponerlos.

SQL
```
SELECT nombre, stock
FROM productos
WHERE stock < 10;
```

Filtros de Texto Exacto (Personal de Ventas)

RRHH quiere la lista exclusiva de los vendedores. *Regla:* El texto siempre va entre comillas simples (`' '`).

SQL

```
SELECT nombre, apellido
FROM empleados
WHERE cargo = 'Vendedor';
```

4.4. Operadores Lógicos

Los operadores lógicos son los componentes que permiten conectar varias condiciones dentro de una cláusula `WHERE`. Su función es evaluar múltiples criterios simultáneamente para decidir si un registro debe ser incluido en el resultado final.

4.4.1. Operador `AND` (Conjunción)

Para que un registro sea seleccionado, **todas** las condiciones conectadas por `AND` deben ser verdaderas (`TRUE`). Es el operador más restrictivo.

- **Caso de uso:** Buscar productos que pertenecen a la categoría 1 **y** que además tienen un stock menor a 10 unidades.

SQL

```
SELECT nombre, stock FROM productos
WHERE categoria_id = 1 AND stock < 10;
```

4.4.2. Operador `OR` (Disyunción)

Un registro será seleccionado si **al menos una** de las condiciones conectadas por `OR` es verdadera. Es un operador inclusivo que amplía los resultados.

- **Caso de uso:** Buscar clientes que se llamen 'Carlos' **o** que tengan un correo de 'gmail.com'.

SQL

```
SELECT nombre, email FROM clientes
WHERE nombre LIKE 'Carlos%' OR email LIKE '%@gmail.com';
```

4.4.3. Operador `NOT` (Negación)

Invierte el valor de verdad de una condición. Si algo es verdadero, `NOT` lo hace falso. Se utiliza para exclusiones masivas.

- **Caso de uso:** Listar todos los empleados que **no** tienen el cargo de 'Gerente'.

SQL

```
SELECT nombre, apellido FROM empleados
WHERE NOT cargo = 'Gerente';
```

4.4.4. El Operador de Desigualdad (<>)

Este operador (también aceptado como `!=`) se utiliza para filtrar registros que son **diferentes** a un valor dado. A diferencia de `NOT`, que niega una condición completa, `<>` es un operador de comparación directa.

- **Caso de uso:** Ver todos los pedidos, excepto aquellos que ya han sido marcados como pagados.

SQL

```
SELECT pedido_id, monto_total FROM pedidos
WHERE pagado <> TRUE;
```

Nota crítica: Al igual que otros operadores de comparación, `<>` no detecta valores `NULL`. Si buscas `email <> 'a@b.com'`, los registros con email nulo quedarán fuera del resultado a menos que uses explícitamente `OR email IS NULL`.

4.4.5. Prioridad de Operadores (Precedencia)

Cuando combinas diferentes operadores en una misma consulta, SQL sigue un orden de resolución jerárquico. Si no usas paréntesis, el motor resuelve en este orden:

1. **NOT** (Máxima prioridad)
2. **AND**
3. **OR** (Mínima prioridad)

4.4.6. El uso de Paréntesis (Agrupamiento Logístico)

Los paréntesis son obligatorios para alterar la prioridad natural de SQL y garantizar que la lógica de negocio se cumpla.

- **Ejemplo de error común:** Quieres ver productos caros (> 1000) de las categorías 1 o 2.
  - *Incorrecto:* `WHERE precio > 1000 AND categoria_id = 1 OR categoria_id = 2` (SQL traerá los caros de la cat 1 y TODOS los de la cat 2).
  - *Correcto:* ```sql SELECT * FROM productos WHERE precio > 1000 AND (categoria_id = 1 OR categoria_id = 2);

4.4.7. Lógica de Cortocircuito

Los motores de SQL son eficientes:

- En un `AND`, si la primera condición es falsa, el motor descarta el registro inmediatamente sin leer el resto.
- En un `OR`, si la primera condición es verdadera, el motor incluye el registro sin evaluar las demás.
- **Consejo:** Coloca la condición que filtre más registros al principio para optimizar el rendimiento de la consulta.

4.5. El Operador BETWEEN

El operador `BETWEEN` se utiliza para filtrar registros que se encuentran dentro de un **rango de valores** específico. Es una forma mucho más limpia y legible de escribir una condición que de otro modo requeriría dos comparaciones unidas por un `AND`.

A. Sintaxis y Rango Inclusivo

Es fundamental recordar que `BETWEEN` es **inclusivo**: los valores que definen el límite (el inicial y el final) también se incluyen en el resultado.

- **Con Números:** Ideal para rangos de precios o stock.

SQL
```
-- Productos con precio entre 100 y 500 (incluye ambos)
SELECT nombre, precio
FROM productos
WHERE precio BETWEEN 100 AND 500;
```

- **Con Fechas:** Es su uso más común para reportes temporales.

SQL
```
-- Pedidos realizados en el primer trimestre del año
SELECT pedido_id, fecha_pedido
FROM pedidos
WHERE fecha_pedido BETWEEN '2025-01-01' AND '2025-03-31';
```

B. Negación: NOT BETWEEN

Si necesitas encontrar valores que están **fuera** de un rango, simplemente antepones la palabra `NOT`.

- **Caso de uso:** Buscar productos que están en stock crítico (muy bajo) o en exceso, evitando el rango "normal".

SQL
```
SELECT nombre, stock
FROM productos
WHERE stock NOT BETWEEN 10 AND 100;
```

C. Ventajas frente al uso de >= y <=

Aunque la consulta `WHERE precio >= 100 AND precio <= 500` devuelve exactamente lo mismo, el uso de `BETWEEN`:

1. **Mejora la legibilidad:** El código parece lenguaje natural.
2. **Evita errores:** Reduce la posibilidad de equivocarse de dirección en los signos de comparación (como poner < en lugar de >).

4.6. Búsquedas por Patrones: LIKE e ILIKE

En SQL, cuando no buscamos un valor exacto (como un ID), sino algo que "se le parezca", utilizamos operadores de patrones con los comodines `%` (varios caracteres) y _ (un solo carácter).

A. Operador LIKE (Estándar SQL)

Es el operador universal. Su característica principal es que es **sensible a mayúsculas y minúsculas (Case-Sensitive)**.

- **Comportamiento:** 'Laptop' es distinto de 'laptop'.
- **Uso en TechNova:**

SQL
```
-- Solo encontrará productos que empiecen exactamente con 'S' mayúscula
SELECT nombre FROM productos WHERE nombre LIKE 'S%';
```

B. Operador ILIKE (Exclusivo de PostgreSQL)

Esta es una de las funciones más queridas de PostgreSQL. La "I" al principio significa **"Insensitive"**. Permite realizar búsquedas ignorando si el texto está en mayúsculas o minúsculas.

Nota Técnica: El operador `ILIKE` **no es estándar de SQL**. Es una extensión propia de PostgreSQL. Si utilizas otros motores de base de datos como Oracle o MySQL, este operador no existirá y deberás usar funciones de transformación (como veremos en el Capítulo 5).

- **Uso en TechNova:**

SQL
```
-- Encontrará 'Silla', 'silla' o 'SILLA'
SELECT nombre FROM productos WHERE nombre ILIKE 'silla%';
```

Resumen de Diferencias

Operador	¿Distingue Mayúsculas?	Compatibilidad	Caso de uso
`LIKE`	**SÍ**	Universal (Cualquier DB)	Cuando el formato exacto importa.
`ILIKE`	**NO**	**Solo PostgreSQL**	Buscadores rápidos y amigables.

4.7. Ordenación de Resultados (ORDER BY)

Los datos en la base de datos no tienen un orden garantizado. Para presentarlos, usamos `ORDER BY`.

Sintaxis

Va al final de la consulta.

- `ASC`: Ascendente (A-Z, 0-9). Es el valor por defecto.
- `DESC`: Descendente (Z-A, 9-0).

Ranking de Precios

Queremos ver los productos más caros de TechNova al principio de la lista.

SQL
```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC;
```

Ordenamiento Múltiple

Si queremos ordenar a los empleados por cargo alfabéticamente, y dentro de cada cargo, por el que gana más dinero.

SQL
```
SELECT nombre, cargo, salario
FROM empleados
ORDER BY cargo ASC, salario DESC;
```

4.8. Control de Resultados (LIMIT y OFFSET)

La cláusula `LIMIT` se utiliza para restringir el número de filas que devuelve una consulta. Es una herramienta esencial para manejar grandes volúmenes de datos y optimizar la visualización de reportes.

Sintaxis en PostgreSQL: Ambas cláusulas se ubican al final de la sentencia, siempre después del `ORDER BY`.

SQL
```
SELECT columnas FROM tabla
ORDER BY columna_referencia
LIMIT cantidad OFFSET salto;
```

- **LIMIT:** Define el número máximo de registros que se mostrarán.
- **OFFSET (Específico para PostgreSQL):** Indica cuántas filas debe saltar el motor antes de empezar a entregar los resultados. Mientras que otros motores usan comas para separar estos valores, PostgreSQL requiere el uso explícito de la palabra clave `OFFSET`.

Caso TechNova 1: Ranking de Precios (Top 3) Para obtener únicamente los tres productos con el precio más alto.

SQL
```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 3;
```

Caso TechNova 2: Paginación de Resultados Para obtener los productos que ocupan los puestos 4, 5 y 6 del ranking (es decir, mostrar 3 registros tras saltar los primeros 3).

SQL

```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 3 OFFSET 3;
```

Caso TechNova 3: Muestreo Rápido Para inspeccionar los datos de los primeros 5 clientes sin procesar la tabla completa.

SQL

```
SELECT * FROM clientes LIMIT 5;
```

4.9. Protocolos de Consulta y Buenas Prácticas

1. **Uso de Alias (`AS`):** Permite renombrar las columnas en el resultado final para que el informe sea más claro para el usuario.
    a. *Ejemplo:* `SELECT nombre AS "Cliente_VIP" FROM clientes;`
2. **Evitar el `SELECT *` en entornos reales:** En tablas con millones de registros, solicitar todas las columnas puede saturar la red y afectar el rendimiento del sistema. Es una práctica fundamental solicitar únicamente los campos estrictamente necesarios para optimizar la transferencia de datos.
3. **Capitalización de palabras reservadas:** Aunque SQL es flexible, escribir los comandos en **MAYÚSCULAS** (`SELECT`, `FROM`, `WHERE`) y los nombres de tablas/columnas en minúsculas ayuda enormemente a distinguir el código del dato.

Capítulo 5: Transformación y Agregación de Datos

Este capítulo es fundamental para convertir datos "crudos" en información útil. Aprenderemos a transformar valores fila por fila y a resumir grandes volúmenes de datos para la toma de decisiones.

5.1. Funciones Escalares (Procesamiento Fila a Fila)

A diferencia de las funciones de agregación que resumen muchas filas en una sola, las **Funciones Escalares** operan de forma individual sobre cada registro. Por cada fila que entra en la consulta, la función devuelve un resultado transformado.

Son herramientas críticas para la **limpieza de datos** y el **formato de reportes**.

A. Funciones de Texto (Limpieza y Búsqueda)

Permiten normalizar y combinar los datos de clientes, empleados y productos.

- **`LOWER(columna)` / `UPPER(columna)`**: Convierten el texto a minúsculas o mayúsculas. Es la técnica recomendada para realizar búsquedas que ignoren si el dato se guardó con errores de capitalización.
    - *Ejemplo (Búsqueda robusta de clientes):*

SQL
```
SELECT * FROM clientes WHERE LOWER(nombre) LIKE '%juan%';
```

- **`CONCAT(texto1, texto2, ...)`**: Une múltiples columnas o textos en una sola cadena. A diferencia de otros métodos, `CONCAT` maneja de forma segura los valores nulos (NULL), evitando que la cadena desaparezca si falta un dato.
    - *Ejemplo (Listado de personal):*

SQL
```
SELECT CONCAT(nombre, ' ', apellido) AS nombre_completo FROM empleados;
```

- **`LENGTH(columna)`**: Devuelve la cantidad de caracteres de un texto. Útil para validar que las identificaciones fiscales o correos cumplan con el formato esperado.

B. Funciones Numéricas (Cálculos y Formato)

Ideales para trabajar con las columnas `salario`, `precio` y `monto_total`.

- **`ROUND(columna, decimales)`**: Redondea un valor numérico a la precisión deseada. Es vital para presentar precios o cálculos de impuestos de forma profesional.
    - *Ejemplo (Cálculo de comisión de pedidos):*

SQL
```
SELECT monto_total, ROUND(monto_total * 0.05, 2) AS comision
FROM pedidos;
```

- **`ABS(valor)`**: Devuelve el valor absoluto, eliminando el signo negativo si lo hubiera. Muy usado para reportar variaciones de stock.

C. Funciones de Fecha (Control Temporal)

PostgreSQL ofrece herramientas potentes para gestionar las columnas `fecha_alta` y `fecha_pedido`.

- **`NOW()`**: Devuelve la fecha y hora exacta del sistema en el momento de la consulta.
- **`CURRENT_DATE`**: Devuelve únicamente la fecha de hoy (formato AAAA-MM-DD).
- **`EXTRACT(unidad FROM columna)`**: Permite extraer una parte específica (Año, Mes o Día) de una fecha para reportes segmentados.
    - *Ejemplo (Ventas por año):*

SQL
```
SELECT pedido_id, EXTRACT(YEAR FROM fecha_pedido) AS
anio_venta FROM pedidos;
```

Aplicación Práctica: Reporte de Operaciones TechNova

En este ejemplo combinamos varias funciones para obtener un reporte profesional de la tabla de empleados:

SQL
```
SELECT
   UPPER(cargo) AS puesto_trabajo,              -- Cargo en
mayúsculas
   CONCAT(nombre, ' ', apellido) AS empleado,    -- Nombre
unido con CONCAT
   ROUND(salario, 1) AS salario_ajustado,        -- Salario
```

```
con un decimal
   EXTRACT(YEAR FROM NOW()) AS anio_actual        -- Año en que
se genera el reporte
FROM empleados;
```

Nota de Diseño: Recuerda que las funciones escalares se pueden usar tanto en el `SELECT` (para mostrar el dato transformado) como en el `WHERE` (para filtrar). Por ejemplo, `WHERE LENGTH(identificacion_fiscal) > 10` te permite encontrar registros con errores de longitud.

5.2. Funciones de Agregado (Estadística Descriptiva)

Las funciones de agregado realizan cálculos sobre un conjunto de valores en una columna y devuelven un **único valor** resultante. Son las herramientas fundamentales para cualquier analista de datos.

- **COUNT()**: Cuenta el número total de registros. Puede usarse como `COUNT(*)` para contar filas totales o `COUNT(columna)` para contar valores no nulos.
- **SUM()**: Suma los valores numéricos de una columna. Es ideal para calcular ingresos o totales de inventario.
- **AVG()**: Calcula el promedio aritmético (la media). Útil para análisis de precios o salarios.
- **MAX() y MIN()**: Identifican el valor más alto y el más bajo en un conjunto de datos.

Aplicación Práctica en TechNova:

SQL

```
-- 1. Contabilidad de Clientes: ¿Cuántos registros tenemos en
nuestra cartera?
SELECT COUNT(*) AS total_clientes FROM clientes;

-- 2. Valorización de Inventario: ¿Cuánto dinero representa
nuestro stock actual?
SELECT SUM(precio * stock) AS inversion_total FROM productos;

-- 3. Análisis Salarial: ¿Cuál es el sueldo promedio de
nuestro equipo?
SELECT AVG(salario) AS sueldo_promedio FROM empleados;
```

5.3. El Motor de los Informes: GROUP BY

La cláusula `GROUP BY` permite agrupar filas que tienen los mismos valores en columnas específicas para aplicar funciones de agregado a cada grupo de forma independiente.

La Regla de Oro del Agrupamiento: Cualquier columna que selecciones en el `SELECT` que **no** esté dentro de una función de agregado (como `SUM`, `AVG`, etc.), debe aparecer obligatoriamente en la cláusula `GROUP BY`.

Ejemplo de Segmentación:

SQL

```
-- Queremos ver el monto total vendido (pagado) por cada empleado vendedor
SELECT empleado_id, SUM(monto_total) AS total_ventas_empleado
FROM pedidos
WHERE pagado = TRUE
GROUP BY empleado_id;
```

5.4. Filtrado Avanzado: La Cláusula HAVING

En el análisis de datos, a menudo no queremos ver todos los grupos resultantes, sino solo aquellos que cumplen con una métrica específica. Es aquí donde entra **HAVING**.

5.4.1. La gran diferencia: WHERE vs. HAVING

Para dominar SQL, es crucial entender el orden de ejecución del motor de la base de datos:

1. **WHERE (El filtro de origen):** Actúa sobre las **filas individuales** antes de que se realice cualquier cálculo. Si quieres eliminar del reporte los productos que no tienen stock antes de sumar el valor total, usas `WHERE`.

2. **HAVING (El filtro de resumen):** Actúa sobre los **resultados agregados** después de que el `GROUP BY` ha terminado su trabajo. Si quieres ver qué categorías tienen un valor de inventario superior a $10,000, debes usar `HAVING`.

Regla nemotécnica: `WHERE` filtra datos crudos; `HAVING` filtra cálculos.

5.4.2. Casos de uso prácticos en TechNova Solutions

A. Control de Calidad en Pedidos: Supongamos que queremos identificar a los clientes que han realizado compras importantes, definidos como aquellos cuyo gasto total acumulado en pedidos pagados supera los $5,000.

SQL
```
SELECT cliente_id, SUM(monto_total) AS gasto_total
FROM pedidos
WHERE pagado = TRUE              -- Paso 1: Filtramos solo los pedidos cobrados
GROUP BY cliente_id                 -- Paso 2: Agrupamos por cliente
HAVING SUM(monto_total) > 5000;    -- Paso 3: Filtramos el resultado del cálculo
```

B. Gestión de Inventario Crítico: Queremos saber qué categorías de productos están en riesgo por tener muy poca variedad (menos de 3 productos distintos registrados).

SQL
```
SELECT categoria_id, COUNT(producto_id) AS variedad_productos
FROM productos
GROUP BY categoria_id
HAVING COUNT(producto_id) < 3;
```

5.4.3. ¿Por qué no puedo usar el Alias en HAVING?

Un error común de principiante es intentar usar el nombre que le dimos a la columna (el Alias) dentro del `HAVING`.

- **Incorrecto:** `HAVING gasto_total > 5000`
- **Correcto:** `HAVING SUM(monto_total) > 5000`

Explicación técnica: El motor de SQL evalúa el `HAVING` antes de generar el nombre final de la columna definido en el `SELECT`. Por lo tanto, debemos repetir la función de agregado dentro de la cláusula.

5.4.4. Combinando todo el poder

Podemos usar ambos filtros en una misma consulta para obtener reportes extremadamente precisos:

SQL

```
-- Queremos el sueldo promedio por cargo, pero solo de los cargos
-- que tienen más de 2 empleados y excluyendo al cargo 'Gerente'
SELECT cargo, AVG(salario) AS sueldo_medio
FROM empleados
WHERE cargo != 'Gerente'                -- Excluimos filas individuales
GROUP BY cargo
HAVING COUNT(empleado_id) > 2;      -- Filtramos grupos por volumen de empleados
```

5.5. Limpieza Visual: DISTINCT

En el análisis de datos, a menudo necesitamos conocer la variedad de nuestros registros sin que se repitan. `DISTINCT` elimina duplicados visuales para mostrarnos los valores únicos presentes.

SQL

```
-- ¿Qué tipos de cargos existen actualmente en la nómina de TechNova?
SELECT DISTINCT cargo FROM empleados;
```

5.5. Protocolos de Análisis y Buenas Prácticas

1. **Nombramiento Profesional (Alias):** Las columnas resultantes de un cálculo suelen tener nombres poco claros (ej. `count`). Use `AS` para otorgar un nombre descriptivo al KPI en su reporte.
2. **Tratamiento de Nulos:** Tenga en cuenta que `AVG`, `SUM`, `MAX` y `MIN` **ignoran** los valores `NULL`. Si un producto no tiene precio asignado, no se incluirá en el cálculo del promedio.
3. **Orden Lógico de la Sentencia:** SQL requiere un orden posicional estricto. Si altera el orden, la consulta fallará:
    a. `SELECT` (¿Qué quiero ver?)
    b. `FROM` (¿De dónde viene?)
    c. `WHERE` (Filtro de filas)
    d. `GROUP BY` (Agrupamiento)
    e. `HAVING` (Filtro de grupos)
    f. `ORDER BY` (Orden final)

Capítulo 6: El Poder de las Relaciones (JOINS)

En el diseño de bases de datos relacionales, la información se distribuye en múltiples tablas para garantizar la integridad y evitar la duplicación de datos (proceso conocido como normalización). Sin embargo, para que los datos tengan sentido en un reporte, es necesario reconstruir esas relaciones.

El **JOIN** (unión) es la operación fundamental de SQL que permite combinar registros de dos o más tablas basándose en una columna común, usualmente una Llave Primaria y una Llave Foránea. Sin esta capacidad, las bases de datos serían simples depósitos de archivos aislados.

6.1. El Rey de las Consultas: INNER JOIN

El **INNER JOIN** es el corazón de las bases de datos relacionales y representa más del 90% de la actividad en un entorno profesional. Su función es extraer la **intersección** de datos: solo muestra registros que tienen una correspondencia exacta en ambas tablas.

Si una fila de la primera tabla no encuentra su "pareja" en la segunda, el sistema la descarta del reporte. Esto garantiza que la información que visualizas sea coherente y esté completa en ambos lados.

Nota sobre los Alias (`AS`): Para que las consultas sean más fáciles de leer y escribir, asignamos un "apodo" a las tablas utilizando la palabra clave `AS`. Esto nos permite usar el nombre corto (como `p`) en lugar del nombre largo de la tabla (como `productos`) en cada columna.

Ejemplos Prácticos en TechNova

A. Relación de Catálogo (Productos y Categorías)

Queremos listar los productos junto con el nombre de su categoría. Usamos `AS p` para referirnos a productos y `AS c` para categorías.

SQL
```
SELECT
   p.nombre AS producto,
   c.nombre AS categoria,
   p.precio
FROM productos AS p
```

```
INNER JOIN categorias AS c ON p.categoria_id = c.categoria_id;
```

B. Relación de Negocio (Pedidos y Clientes)

Para vincular cada pedido con el nombre del cliente, usamos el alias `AS pe` para la tabla de pedidos y `AS cl` para la tabla de clientes.

SQL
```
SELECT
   pe.pedido_id,
   cl.nombre AS cliente,
   pe.monto_total
FROM pedidos AS pe
INNER JOIN clientes AS cl ON pe.cliente_id = cl.cliente_id;
```

C. El "Salto Doble" (Pedidos, Clientes y Empleados)

El uso de `AS` es fundamental cuando conectamos tres o más tablas, ya que nos permite indicar con claridad a qué tabla pertenece cada columna sin repetir nombres largos.

SQL
```
SELECT
   pe.pedido_id,
   cl.nombre AS cliente,
   CONCAT(em.nombre, ' ', em.apellido) AS vendedor
FROM pedidos AS pe
INNER JOIN clientes AS cl ON pe.cliente_id = cl.cliente_id
INNER JOIN empleados AS em ON pe.empleado_id = em.empleado_id;
```

D. INNER JOIN con Cálculos de Detalle

En este ejemplo, cruzamos la tabla de detalles (`AS dp`) con la de productos (`AS pr`) para calcular el subtotal de cada línea del pedido.

SQL
```
SELECT
   dp.pedido_id,
   pr.nombre AS producto,
   dp.cantidad,
  (dp.cantidad * dp.precio_unitario) AS subtotal
```

```
FROM detalles_pedido AS dp
INNER JOIN productos AS pr ON dp.producto_id = pr.producto_id;
```

Nota Técnica: En PostgreSQL, la palabra `INNER` es opcional. Escribir simplemente `JOIN` realizará la misma operación por defecto. El uso de `AS` ayuda a que el código sea más ordenado y fácil de entender para quien lo lee por primera vez.

6.2. LEFT JOIN (Unión Externa Izquierda)

El **LEFT JOIN** es el segundo tipo de unión más utilizado. A diferencia del `INNER JOIN`, su objetivo no es solo buscar coincidencias, sino **preservar la integridad de la tabla principal** (la que se encuentra a la izquierda del comando, en el `FROM`).

Esta función devuelve todos los registros de la tabla de la izquierda, junto con los datos coincidentes de la tabla de la derecha. Si no existe una correspondencia, el sistema no descarta la fila; en su lugar, rellena los campos de la tabla secundaria con valores **NULL** (nulos).

A. Identificación de registros sin actividad

Es la herramienta ideal para encontrar elementos que no han tenido movimientos. Por ejemplo, listar todos los clientes para identificar quiénes **no han realizado pedidos todavía**.

SQL
```
SELECT
   cl.nombre AS cliente,
   pe.pedido_id AS nro_pedido,
   pe.monto_total
FROM clientes AS cl
LEFT JOIN pedidos AS pe ON cl.cliente_id = pe.cliente_id;
```

En el resultado, los clientes sin compras aparecerán con el campo `nro_pedido` vacío (NULL).

B. Reporte de inventario y categorías

Si deseamos ver todas las categorías del sistema, incluso aquellas que no tienen productos asociados actualmente:

SQL
```
SELECT
   c.nombre AS categoria,
```

```
   p.nombre AS producto
FROM categorias AS c
LEFT JOIN productos AS p ON c.categoria_id = p.categoria_id;
```

C. Control de gestión de empleados

Útil para obtener un listado de toda la plantilla de empleados y verificar cuántos pedidos ha gestionado cada uno, sin excluir a los empleados nuevos o administrativos que no tienen ventas registradas.

SQL

```
SELECT
   CONCAT(em.nombre, ' ', em.apellido) AS empleado,
   pe.pedido_id AS pedido_gestionado
FROM empleados AS em
LEFT JOIN pedidos AS pe ON em.empleado_id = pe.pedido_id;
```

D. Combinación con filtrado de nulos

El `LEFT JOIN` es fundamental para tareas de limpieza de datos. Podemos usarlo para buscar específicamente los registros que **no tienen** pareja, añadiendo una cláusula `WHERE` que busque los valores nulos.

SQL

```
-- Ejemplo: Listar solo los productos que nunca han sido
vendidos
SELECT
   pr.nombre AS producto_sin_venta
FROM productos AS pr
LEFT JOIN detalles_pedido AS dp ON pr.producto_id =
dp.producto_id
WHERE dp.pedido_id IS NULL;
```

6.3. Otros tipos de uniones y técnicas

Aunque el `INNER` y el `LEFT JOIN` cubren la gran mayoría de las necesidades operativas, existen otras variantes diseñadas para casos específicos de análisis de datos, generación de combinaciones o estructuras jerárquicas.

- **RIGHT JOIN:** Es la versión inversa del `LEFT JOIN`. Prioriza la tabla de la derecha, manteniendo todos sus registros y trayendo solo las coincidencias de la tabla de la izquierda. En la práctica se utiliza poco, ya

que el mismo resultado se logra simplemente reordenando las tablas en un `LEFT JOIN`.

- **FULL OUTER JOIN:** Devuelve absolutamente todos los registros de ambas tablas, coincidan o no. Donde no hay una correspondencia, el sistema rellena los espacios con valores `NULL`. Es la herramienta ideal para auditorías de integridad donde se busca detectar datos huérfanos en ambos lados simultáneamente.
- **CROSS JOIN:** Produce un producto cartesiano. No requiere una columna de unión (cláusula `ON`), ya que combina cada fila de la primera tabla con todas las filas de la segunda. Se utiliza para generar matrices de todas las combinaciones posibles entre dos conjuntos de datos.
- **SELF JOIN (Auto-unión):** Es una técnica donde una tabla se une consigo misma. Es fundamental para manejar jerarquías dentro de una misma entidad; por ejemplo, para asociar a un empleado con su supervisor cuando ambos se encuentran registrados en la misma tabla de empleados.

6.4. Buenas Prácticas en el uso de JOINS

1. **Usa Alias siempre:** Utiliza abreviaturas para las tablas (ej: `FROM productos p`). Esto hace la consulta legible y evita errores de ambigüedad.
2. **Califica tus columnas:** Antepón siempre el alias a la columna (ej: `p.nombre`). Así, si añades una columna igual en otra tabla, el código no se romperá.
3. **Une por Claves (IDs):** Realiza siempre los vínculos mediante `PRIMARY KEY` y `FOREIGN KEY`. Unir por campos de texto (como nombres) es lento y propenso a errores.
4. **No olvides el ON:** Asegúrate de definir siempre la condición de unión. Omitir el `ON` genera un "Producto Cartesiano", combinando todas las filas con todas, lo que puede colapsar el servidor.
5. **Filtra con inteligencia:** Usa `INNER JOIN` para registros que deben existir en ambas tablas y `LEFT JOIN` solo cuando necesites incluir registros que podrían no tener relación (como un cliente sin pedidos).

Capítulo 7: Subconsultas

En los capítulos anteriores aprendimos a extraer datos de tablas relacionadas mediante `JOIN`. Sin embargo, a veces necesitamos realizar una consulta cuyo filtro dependa del resultado de otra consulta previa. Para esto utilizamos las **Subconsultas**.

Una subconsulta es una sentencia `SELECT` anidada dentro de otra instrucción (`SELECT`, `INSERT`, `UPDATE` o `DELETE`).

7.1. Subconsultas Escalares (Un solo valor)

Se utilizan cuando la consulta interna devuelve un único dato (una fila y una columna). Son ideales para comparaciones matemáticas.

Caso de uso: Queremos listar todos los productos cuyo precio sea mayor al promedio de todos los productos de la tienda.

SQL
```
SELECT nombre, precio
FROM productos
WHERE precio > (SELECT AVG(precio) FROM productos);
```

Aquí, la subconsulta calcula el promedio primero, y luego la consulta principal lo usa como filtro.

7.2. Subconsultas con Operadores de Conjunto (`IN` y `NOT IN`)

Cuando la subconsulta devuelve una lista de valores, utilizamos el operador `IN`.

Caso de uso: Obtener los nombres de los clientes que han realizado pedidos por un monto total superior a 500.

SQL
```
SELECT nombre, email
FROM clientes
WHERE cliente_id IN (
   SELECT cliente_id
   FROM pedidos
   WHERE monto_total > 500
);
```

7.3. Subconsultas de Correlación con `EXISTS`

A diferencia de las anteriores, estas se ejecutan por cada fila de la consulta principal. El operador `EXISTS` devuelve *verdadero* si la subconsulta encuentra al menos un registro que coincida.

Caso de uso: Listar los empleados que han procesado al menos un pedido (es decir, empleados activos en ventas).

SQL
```
SELECT nombre, apellido, cargo
FROM empleados e
WHERE EXISTS (
   SELECT 1
   FROM pedidos p
   WHERE p.empleado_id = e.empleado_id
);
```

7.4. Subconsultas en la cláusula `FROM` (Tablas Derivadas)

Podemos tratar el resultado de una subconsulta como si fuera una tabla temporal para realizar cálculos sobre ella.

Caso de uso: Obtener el promedio de productos vendidos por pedido.

SQL
```
SELECT AVG(total_articulos) AS promedio_por_pedido
FROM (
   SELECT pedido_id, SUM(cantidad) AS total_articulos
   FROM detalles_pedido
   GROUP BY pedido_id
) AS resumen_cantidades;
```

7.5. Diferencias clave: ¿Subconsulta o JOIN?

Aunque muchas subconsultas pueden escribirse como `JOIN`, existen reglas generales:

1. **Legibilidad:** Las subconsultas suelen ser más fáciles de leer cuando se busca "filtrar" datos basados en otra tabla sin querer mostrar columnas de esa segunda tabla.

2. **Rendimiento:** Los `JOIN` suelen ser más eficientes en grandes volúmenes de datos porque el motor de la base de datos puede optimizar mejor la ruta de búsqueda.
3. **Necesidad:** Para cálculos agregados (como el ejemplo del promedio en el punto 7.1), la subconsulta es obligatoria.

7.6. Protocolo de Buenas Prácticas

- **Priorizar JOINs:** Utiliza un `JOIN` en lugar de una subconsulta si necesitas procesar grandes volúmenes de datos, ya que el motor de la base de datos suele optimizarlos mejor.
- **Limitar la profundidad:** Evita anidar demasiadas subconsultas (una dentro de otra); esto dificulta la lectura y el mantenimiento del código.
- **Uso de Alias:** Asigna siempre un alias claro a las subconsultas en la cláusula `FROM` para evitar errores de ambigüedad en las columnas.
- **Precisión en el filtrado:** Asegúrate de que las subconsultas de comparación (usando `=` o `IN`) devuelvan únicamente los valores necesarios para no sobrecargar la memoria.
- **Validar Nulos:** Ten precaución al usar `NOT IN` con subconsultas que puedan devolver valores `NULL`, ya que esto puede causar que la consulta principal no devuelva ningún resultado.

Capítulo 8: Estructuras Virtuales (Vistas)

A medida que las consultas se vuelven más complejas (especialmente con múltiples `JOIN` y subconsultas), escribirlas una y otra vez resulta ineficiente. Las **Vistas** nos permiten guardar una consulta en la base de datos y consultarla como si fuera una tabla real.

8.1. Concepto de Tabla Virtual

Una vista es una **ventana** a los datos. No almacena registros físicamente (a excepción de las vistas materializadas); lo que guarda es la instrucción `SELECT`.

- **Abstracción:** El usuario no necesita saber qué tablas o uniones hay detrás; solo consulta la vista.
- **Consistencia:** Si la lógica de un cálculo cambia, solo modificas la vista y todas las aplicaciones conectadas reciben el cambio.

8.2. Creación de Vistas de Reporte

Utilizamos `CREATE VIEW` para consolidar información de varias tablas (como `clientes`, `pedidos` y `empleados`) en una sola estructura fácil de leer.

Caso de uso: Reporte de Ventas por Vendedor

SQL

```
CREATE VIEW vista_ventas_empleados AS
SELECT
   e.nombre || ' ' || e.apellido AS vendedor,
   COUNT(p.pedido_id) AS total_pedidos,
   SUM(p.monto_total) AS facturacion_total
FROM empleados e
LEFT JOIN pedidos p ON e.empleado_id = p.empleado_id
GROUP BY e.empleado_id, e.nombre, e.apellido;
```

8.3. Seguridad y Privacidad de Datos

Las vistas permiten ocultar columnas sensibles de las tablas maestras. Es una de las mejores herramientas de control de acceso.

Caso de uso: Directorio público de empleados En tu tabla `empleados` tienes el campo `salario`. No quieres que todos lo vean, así que creas una vista sin esa columna:

SQL

```
CREATE VIEW directorio_empleados AS
SELECT nombre, apellido, cargo
FROM empleados;
```

Si otorgas permisos solo a la vista, los usuarios nunca podrán consultar el salario directamente.

8.4. Vistas Actualizables y `WITH CHECK OPTION`

Puedes usar `INSERT` o `UPDATE` sobre una vista para que afecte a la tabla original, siempre que la vista sea simple (venga de una sola tabla). Para evitar que se inserten datos que no cumplen con el filtro de la vista, usamos `WITH CHECK OPTION`.

SQL

```
CREATE VIEW productos_activos AS
SELECT producto_id, nombre, precio, en_stock
FROM productos
WHERE en_stock = TRUE
WITH CHECK OPTION;
```

Si intentas insertar un producto con `en_stock = FALSE` a través de esta vista, la base de datos lo rechazará.

8.5. Vistas Materializadas: Persistencia y Big Data

A diferencia de una vista estándar, que es una consulta que se ejecuta "en vivo" cada vez que la llamas, una **Vista Materializada** guarda físicamente el resultado en el disco duro. Es, en esencia, una tabla real que se crea a partir de una consulta, pero que mantiene un vínculo con sus tablas de origen.

A. El desafío de los Grandes Volúmenes (Big Data)

En un entorno donde la tabla `detalles_pedido` crece hasta tener millones de registros, realizar cálculos de ventas por categoría en tiempo real se vuelve costoso para el servidor.

- **El Problema:** Cada vez que un gerente pide un reporte, la base de datos debe leer millones de filas, realizar múltiples `JOINs` y sumar precios. Esto

consume CPU y memoria, ralentizando la base de datos para otros usuarios.

- **La Solución:** La Vista Materializada realiza este trabajo pesado **una sola vez** (por ejemplo, en la madrugada) y guarda el resultado final.

B. Implementación Técnica

Utilizando tu estructura de tablas, supongamos que necesitamos un reporte de rendimiento por categoría que sea instantáneo.

SQL
```
-- Creamos la vista materializada para análisis de Big Data
CREATE MATERIALIZED VIEW reporte_rendimiento_categorias AS
SELECT
   c.nombre AS categoria,
   COUNT(DISTINCT p.pedido_id) AS volumen_pedidos,
   SUM(dp.cantidad) AS unidades_vendidas,
   SUM(dp.cantidad * dp.precio_unitario) AS facturacion_total
FROM categorias c
JOIN productos prod ON c.categoria_id = prod.categoria_id
JOIN detalles_pedido dp ON prod.producto_id = dp.producto_id
JOIN pedidos p ON dp.pedido_id = p.pedido_id
GROUP BY c.nombre;
```

C. Gestión del Ciclo de Vida: El "Refresco"

Como los datos están guardados físicamente, si entra un nuevo pedido en la tabla `pedidos`, este **no aparecerá** automáticamente en la vista materializada. Esto se conoce como "datos estancados" (stale data).

Para actualizarla, debemos ejecutar un comando de refresco. En sistemas como PostgreSQL, se utiliza:

SQL
```
-- Actualiza la "foto" de los datos con la información más reciente
REFRESH MATERIALIZED VIEW reporte_rendimiento_categorias;
```

D. ¿Cuándo elegir una Vista Materializada?

1. **Reportes de Inteligencia de Negocios (BI):** Cuando los datos se analizan por tendencias y no importa si falta la venta del último minuto.

2. **Dashboards ejecutivos:** Donde la velocidad de carga de la página es más importante que la precisión al segundo.
3. **Consultas repetitivas pesadas:** Cuando la misma consulta compleja es ejecutada cientos de veces al día por diferentes usuarios.

8.6. Encapsulamiento y Mantenimiento

Las vistas funcionan como una "capa de seguridad" o un "aislante" entre la estructura de las tablas y las aplicaciones (o usuarios) que consumen los datos.

1. **Independencia Lógica**: Si decides cambiar el nombre de la tabla `empleados` a `staff_tecnico`, todas tus consultas fallarían. Sin embargo, si tienes una Vista llamada `v_empleados_activos`, solo cambias la definición interna de la vista y nadie notará que la tabla original cambió de nombre.
2. **Simplificación de Fórmulas**: Si la empresa decide que el `monto_total` de los pedidos ahora debe incluir un impuesto del 15% automáticamente, puedes definir ese cálculo dentro de una vista. Así, todos los reportes usarán la misma fórmula oficial y no habrá discrepancias.

8.7. Gestión y Eliminación

Para administrar tus vistas de forma profesional, utiliza estos comandos esenciales:

- **Actualizar**: Si quieres añadir una columna a una vista existente sin borrarla, usa `CREATE OR REPLACE VIEW`.
- **Eliminar**: Se usa `DROP VIEW nombre_vista;`.
- **Borrado en Cascada**: Si creaste una vista basada en otra vista anterior, no podrás borrar la primera fácilmente. Debes usar `DROP VIEW nombre_vista CASCADE;` para eliminar también las estructuras que dependen de ella.
- **Metadatos**: Puedes consultar qué vistas tienes creadas revisando las tablas del sistema (como `pg_views` en PostgreSQL), lo que permite auditar quién tiene acceso a qué información.

8.8. Protocolo de Buenas Prácticas

- **Independencia Lógica:** Utiliza vistas como una capa de seguridad para que, si cambias nombres de tablas originales, las aplicaciones externas no se vean afectadas al consultar la vista.
- **Simplificación de Fórmulas:** Define cálculos complejos (como impuestos o subtotales) dentro de la vista para asegurar que todos los reportes utilicen la misma fórmula oficial y evitar discrepancias.

- **Privacidad de Datos:** Emplea vistas para ocultar columnas con información sensible (como salarios o datos privados) y otorga permisos de acceso solo a estas estructuras filtradas.
- **Gestión de Big Data:** Para tablas con millones de registros, prefiere **Vistas Materializadas**, ya que guardan el resultado físicamente y evitan recalculados costosos en tiempo real.
- **Control de Ciclo de Vida:** Recuerda ejecutar el comando `REFRESH MATERIALIZED VIEW` periódicamente si usas vistas materializadas para evitar trabajar con "datos estancados".
- **Mantenimiento Limpio:** Utiliza `CREATE OR REPLACE VIEW` para actualizar definiciones y `DROP VIEW ... CASCADE` cuando necesites eliminar vistas que tengan otras estructuras dependientes.

Anexo A: El Contrato de Fiabilidad (Propiedades ACID)

Para que una base de datos sea considerada profesional y segura, debe cumplir con las **Propiedades ACID**. Estas no son sugerencias, sino un conjunto de principios fundamentales que garantizan que cada transacción en **TechNova Solutions** sea infalible.

1. Atomicity (Atomicidad)

Concepto: "O todo, o nada". Una transacción es una unidad indivisible.

- **Ejemplo en SQL:**

SQL

```
BEGIN TRANSACTION;
-- Paso 1: Restar stock
UPDATE productos SET stock = stock - 1 WHERE id_producto =
101;
-- Paso 2: Registrar venta (Si aquí falla la luz...)
INSERT INTO ventas (monto, id_cliente) VALUES (1200.00, 5);
-- El motor detecta el fallo y ejecuta:
ROLLBACK;
```

2. Consistency (Consistencia): El Equilibrio de Responsabilidad

Concepto: La base de datos solo pasa de un estado válido a otro. Es un equilibrio entre el **Diseñador** (que crea las leyes) y el **Motor** (que las hace cumplir).

- **PostgreSQL:** El "policía estricto". Bloquea cualquier dato sospechoso de inmediato.
- **MySQL:** Depende de su configuración. Requiere el `STRICT MODE` para no "recortar" datos erróneos.
- **SQLite:** El "permisivo". Delega casi toda la responsabilidad al programador.

Ejemplo de Equilibrio:

SQL

```
-- El DISEÑADOR crea la ley (Restricción):
ALTER TABLE productos ADD CONSTRAINT precio_positivo CHECK
(precio > 0);

-- El MOTOR rechaza la infracción:
UPDATE productos SET precio = -10.50 WHERE id_producto = 101;
-- Resultado: Error de Check Constraint.
```

3. Isolation (Aislamiento)

Concepto: Las transacciones no se "pisan". El motor asegura que si dos personas tocan el mismo dato, no haya caos.

Ejemplo en SQL (El problema del doble cobro):

Imagina dos vendedores intentando actualizar el mismo saldo al mismo tiempo:

Tiempo	Usuario A (Cajero 1)	Usuario B (Cajero 2)
T1	`BEGIN;`	`BEGIN;`
T2	`UPDATE cuenta SET saldo = saldo - 100 WHERE id = 1;`	
T3	*(La fila queda bloqueada por A)*	`UPDATE cuenta SET saldo = saldo - 50 WHERE id = 1;`
T4		*(El motor deja al Usuario B en "espera" o "congelado")*
T5	`COMMIT;`	*(Se libera el bloqueo)*
T6		*(La consulta de B se ejecuta ahora sobre el nuevo saldo)*

Importancia técnica: Sin aislamiento, el Usuario B podría leer un saldo desactualizado antes de que el Usuario A finalice su operación, provocando que uno de los cobros se pierda en el sistema. El aislamiento garantiza que todos los cambios se procesen de manera ordenada y segura.

4. Durability (Durabilidad)

Concepto: Una vez que recibes el mensaje de `COMMIT`, el dato es permanente, pase lo que pase con el hardware.

- **Ejemplo en SQL:**

SQL

```
INSERT INTO clientes (nombre) VALUES ('Laura Soler');
COMMIT;
-- El motor confirma que el dato ya está físicamente en el
disco.
```

Resumen de Responsabilidades

Propiedad	Lo que hace el Diseñador	Lo que hace el Motor (SGBD)
Atomicidad	Agrupar pasos en `BEGIN/COMMIT`.	Deshacer todo si algo falla.
Consistencia	Definir reglas (`CHECK`, `NOT NULL`).	Rechazar datos que rompan esas reglas.
Aislamiento	No dejar transacciones abiertas por horas.	Bloquear filas para evitar choques.
Durabilidad	Asegurarse de ejecutar el `COMMIT`.	Grabar en el disco antes de avisar "OK".

Anexo B: Guía de Instalación y Uso de PostgreSQL (Local)

Para realizar las prácticas del curso, instalaremos **PostgreSQL** como motor de base de datos y **pgAdmin** como interfaz gráfica para gestionar las tablas de forma visual.

1. Descarga e Instalación

1. **Sitio Oficial:** Ve a postgresql.org/download y selecciona tu sistema operativo (Windows, macOS o Linux).
2. **Instalador:** Descarga el instalador de **EDB**. Durante la instalación, asegúrate de que estas cuatro opciones estén marcadas:
    a. PostgreSQL Server (El motor).
    b. pgAdmin 4 (La interfaz gráfica).
    c. Stack Builder (Para complementos).
    d. Command Line Tools (Para usar la terminal).
3. **Contraseña del Superusuario:** El instalador te pedirá una contraseña para el usuario `postgres`.

IMPORTANTE: No olvides esta contraseña. La usaremos para conectarnos siempre.

4. **Puerto:** Por defecto es el **5432**. No lo cambies a menos que sea estrictamente necesario.

2. Primeros pasos con pgAdmin 4

Una vez instalado, abre la aplicación **pgAdmin 4**.

1. **Master Password:** Te pedirá una contraseña para proteger la aplicación; puedes usar la misma de la instalación.
2. **Conexión al Servidor:** En el panel izquierdo, despliega "Servers". Te pedirá la contraseña del usuario `postgres` que definiste al instalar.
3. **Crear la base de datos TechNova:**
    a. Haz clic derecho en "Databases" -> Create -> Database...
    b. Nombre: `technova_solutions`.
    c. Clic en **Save**.

3. Ejecutar comandos SQL (Query Tool)

Para escribir el código de los **Capítulos 2 al 8**:

1. Selecciona la base de datos `technova_solutions` en el árbol de la izquierda.
2. Haz clic en el icono de la herramienta de consultas (**Query Tool**) en la barra superior (parece un pequeño rayo o un tablero con un lápiz).
3. Escribe tu código SQL y presiona **F5** (o el botón de "Play") para ejecutarlo.

4. Configuración del "Auto-commit" en pgAdmin

Como mencionamos en el **Capítulo 3** y el **Anexo A**, para practicar realmente el `ROLLBACK`, es recomendable controlar las transacciones.

- En la ventana del Query Tool, ve a la pestaña **"Query Tool Settings"** o al menú de opciones.
- Busca la opción **"Auto-commit"** y **"Auto-rollback"**.
- Si deseas practicar de forma manual, desactiva el Auto-commit. De esta forma, nada se guardará hasta que tú escribas expresamente el comando `COMMIT;`.

Tips de Resolución de Problemas (Troubleshooting)

- **Servidor no responde:** Asegúrate de que el servicio de PostgreSQL esté iniciado en tu sistema operativo (en Windows: *Servicios -> PostgreSQL -> Iniciar*).
- **Error de Contraseña:** Si olvidas la contraseña, deberás modificar el archivo `pg_hba.conf` para permitir el acceso, pero lo ideal es guardarla en un lugar seguro desde el inicio.

Anexo C: Acceso mediante Consola (Herramienta psql)

En entornos de servidores o situaciones donde no se dispone de una interfaz gráfica, el acceso a PostgreSQL se realiza a través de **psql**, una interfaz basada en terminal para gestionar la base de datos mediante comandos.[1]

1. Conexión Inicial

Para iniciar una sesión desde la terminal o línea de comandos del sistema operativo, se utiliza la siguiente sintaxis:

Bash
```
psql -h localhost -U postgres -d technova_solutions
```

- **-h:** Indica el servidor (host), usualmente `localhost` para pruebas locales.[2]
- **-U:** Define el usuario (por defecto es `postgres`).
- **-d:** Especifica la base de datos a la que se desea conectar.

2. Comandos de Navegación Esenciales (Meta-comandos)

Una vez dentro de la consola, los comandos de PostgreSQL comienzan con una barra invertida (`\`). Los más utilizados son:

Comando	Acción
\l	Listar todas las bases de datos disponibles.
\c nombre_bd	Conectarse a una base de datos específica.
\dt	Listar todas las tablas de la base de datos actual.
\d nombre_tabla	Mostrar la estructura detallada de una tabla (columnas, tipos y constraints).
\q	Salir de la consola psql y volver a la terminal.

3. Ejecución de Consultas

Dentro de psql se pueden escribir las mismas sentencias SQL vistas en los capítulos anteriores. Es indispensable finalizar cada instrucción con un **punto y**

coma (;), de lo contrario, la consola interpretará que la instrucción continúa en la siguiente línea.

SQL
```
SELECT * FROM productos;
```

Nota de Seguridad: Al ejecutar el comando de conexión, el sistema solicitará la contraseña. Por seguridad, los caracteres escritos no se mostrarán en pantalla mientras se introducen.

Anexo D: Portabilidad y Backups en PostgreSQL

Este apartado profundiza en las herramientas de administración que operan fuera del entorno de consultas SQL. Estas habilidades son críticas para asegurar que los datos puedan ser migrados, respaldados o integrados con otras herramientas de análisis.

A.1. Respaldos con `pg_dump`

La utilidad `pg_dump` extrae una base de datos de PostgreSQL hacia un archivo de script o hacia otros formatos de archivo.

- **Exportación en Texto Plano (SQL):** Genera un archivo `.sql` que contiene todas las instrucciones `CREATE TABLE`, `INSERT`, etc. Es ideal para bases de datos pequeñas o medianas que necesitan ser legibles.
    - **Comando:**

Bash
```
pg_dump -U username -d dbname > backup_file.sql
```

- **Exportación en Formato Custom (Binario):** Utiliza el parámetro `-Fc`. Es el método más profesional porque permite restauraciones selectivas y compresión de datos.
    - **Comando:**

Bash
```
pg_dump -U username -F c -d dbname > backup_file.dump
```

A.2. Restauración de Datos

Dependiendo de cómo se exportó el archivo, el método de recuperación varía:

- **Restauración con `psql` (para archivos .sql):** Simplemente "vuelve a escribir" el script en una base de datos nueva o existente.

Bash
```
psql -U username -d new_dbname -f backup_file.sql
```

- **Restauración con `pg_restore` (para archivos .dump):** Ofrece mayor flexibilidad, como la capacidad de usar múltiples hilos para acelerar el proceso (`-j`).

Bash
```
pg_restore -U username -d new_dbname backup_file.dump
```

A.3. Intercambio Masivo de Datos con CSV

Para un profesional de datos, el comando `COPY` es la herramienta más eficiente para mover información entre PostgreSQL y herramientas como Excel, Pandas o Spark.

- **Import (Carga de datos externa):** Útil cuando recibes un dataset externo y necesitas poblar una tabla rápidamente.

SQL
```
COPY table_name
FROM '/ruta/del/archivo.csv'
DELIMITER ','
CSV HEADER;
```

- **Export (Extracción para análisis):** Permite guardar el resultado de una consulta directamente en un archivo CSV.

SQL
```
COPY (SELECT * FROM ventas WHERE fecha > '2023-01-01')
TO '/ruta/del/reporte.csv'
DELIMITER ','
CSV HEADER;
```

A.4. Consideraciones de Seguridad y Entorno

1. **Variables de Entorno:** Para evitar que el sistema pida la contraseña en cada comando (lo cual impide la automatización), se puede configurar la variable `PGPASSWORD` o usar un archivo `.pgpass`.
2. **Rutas de Archivos:** Al usar `COPY`, el usuario de PostgreSQL debe tener permisos de lectura/escritura en la carpeta del sistema operativo donde se encuentra el archivo.

3. **Integridad:** Se recomienda realizar exportaciones siempre con la bandera `--no-owner` si se planea restaurar la base de datos en un servidor con un usuario diferente.