📊 Funciones Agregadas y JOINS

Consultas Avanzadas en MySQL

Python Full Stack - Clase 23

¿Qué aprenderemos hoy?

- 📈 Usar funciones agregadas (COUNT, SUM, AVG, MAX, MIN)
- 🔗 Unir tablas con JOIN e INNER JOIN
- ⬅️ Comprender LEFT JOIN y cuándo usarlo
- 🔐 Configurar restricciones de llaves foráneas
- 💾 Exportar bases de datos desde MySQL Workbench
- 🎯 Agrupar resultados con GROUP BY

📋 Preparación: Base de Datos

Antes de comenzar, ejecuta el archivo SQL:

- `ejercicios/00-e-commerce.sql`

Tablas principales:

- `usuarios` - Clientes de la tienda
- `productos` - Productos disponibles
- `pedidos` - Pedidos realizados
- `categorias` - Categorías de productos
- `resenas` - Reseñas de productos
- `direcciones` - Direcciones de usuarios

📈 Funciones Agregadas

¿Qué son las Funciones Agregadas?

Las **funciones agregadas** son herramientas especiales de SQL que nos permiten hacer **cálculos sobre grupos de datos**.

Imagina que tienes una tabla de ventas y quieres saber:

- ¿Cuántas ventas se hicieron? → **COUNT**
- ¿Cuál fue el total de dinero? → **SUM**
- ¿Cuál fue el promedio? → **AVG**
- ¿Cuál fue la venta más alta? → **MAX**
- ¿Cuál fue la venta más baja? → **MIN**

Sintaxis de Funciones Agregadas

```sql
SELECT FUNCION(columna) FROM tabla;
```

Ejemplo básico:

```sql
SELECT COUNT(*) FROM usuarios;
```

Esto cuenta **todas las filas** de la tabla usuarios.

COUNT: Contar Registros

COUNT nos permite contar cuántos registros hay.

```
SELECT COUNT(*) FROM usuarios; -- Contar todos los usuarios

SELECT COUNT(email) FROM usuarios; -- Contar usuarios con email

SELECT COUNT(DISTINCT ciudad) FROM direcciones; -- Contar ciudades únicas en direcciones
```

¿Cuándo usar COUNT?

- Saber cuántos registros hay en total
- Contar registros que cumplen una condición
- Contar valores únicos

SUM: Sumar Valores

SUM suma todos los valores de una columna numérica.

```sql
-- Sumar todos los precios de productos
SELECT SUM(precio) FROM productos;

-- Sumar pedidos de un mes específico
SELECT SUM(total) FROM pedidos
WHERE fecha >= '2024-02-01';
```

⚠️ **Importante:** SUM solo funciona con números (INT, FLOAT, DECIMAL).

AVG: Calcular Promedio

AVG calcula el promedio (media) de los valores.

```sql
-- Promedio de edad de usuarios
SELECT AVG(edad) FROM usuarios;

-- Promedio de precios de productos
SELECT AVG(precio) FROM productos;
```

Resultado: AVG siempre devuelve un número decimal, incluso si los valores son enteros.

MAX y MIN: Valores Extremos

MAX encuentra el valor más grande.

MIN encuentra el valor más pequeño.

```sql
-- Precio más alto
SELECT MAX(precio) FROM productos;

-- Precio más bajo
SELECT MIN(precio) FROM productos;

-- Usuario más joven y más viejo
SELECT MIN(edad), MAX(edad) FROM usuarios;
```

💪 Ejercicio: Funciones Básicas

¡Hora de practicar!

Crea un archivo `ejercicio_funciones.sql` con:

1. ¿Cuántos productos tienen un precio mayor a $100?
2. ¿Cuál es la diferencia entre el precio más caro y el más barato de los productos?
3. ¿Cuántos usuarios tienen más de 25 años?
4. Calcula el valor total del inventario (suma de todos los precios de productos)
5. ¿Cuál es el precio promedio de los productos que cuestan menos de $200?

Pistas:

- Usa COUNT(*) con WHERE para contar con condiciones
- Puedes hacer operaciones matemáticas: MAX() - MIN()
- WHERE filtra antes de aplicar funciones agregadas
- SUM() suma todos los valores

⏱️ **Tiempo:** 15 minutos

✅ Solución: Funciones Básicas

1. Productos con precio mayor a $100:

```sql
SELECT COUNT(*) AS productos_caros FROM productos WHERE precio > 100;
```

2. Diferencia entre precio más caro y más barato:

```sql
SELECT MAX(precio) - MIN(precio) AS diferencia_precios FROM productos;
```

3. Usuarios con más de 25 años:

```sql
SELECT COUNT(*) AS usuarios_mayores_25 FROM usuarios WHERE edad > 25;
```

4. Valor total del inventario:

```sql
SELECT SUM(precio) AS valor_total_inventario FROM productos;
```

5. Precio promedio de productos menores a $200:

```sql
SELECT AVG(precio) AS promedio_productos_baratos FROM productos WHERE precio < 200;
```

🔗 JOINS: Uniendo Tablas

¿Por qué Necesitamos JOIN?

Hasta ahora hemos trabajado con **una sola tabla**. Pero en la vida real, los datos están **relacionados** entre varias tablas.

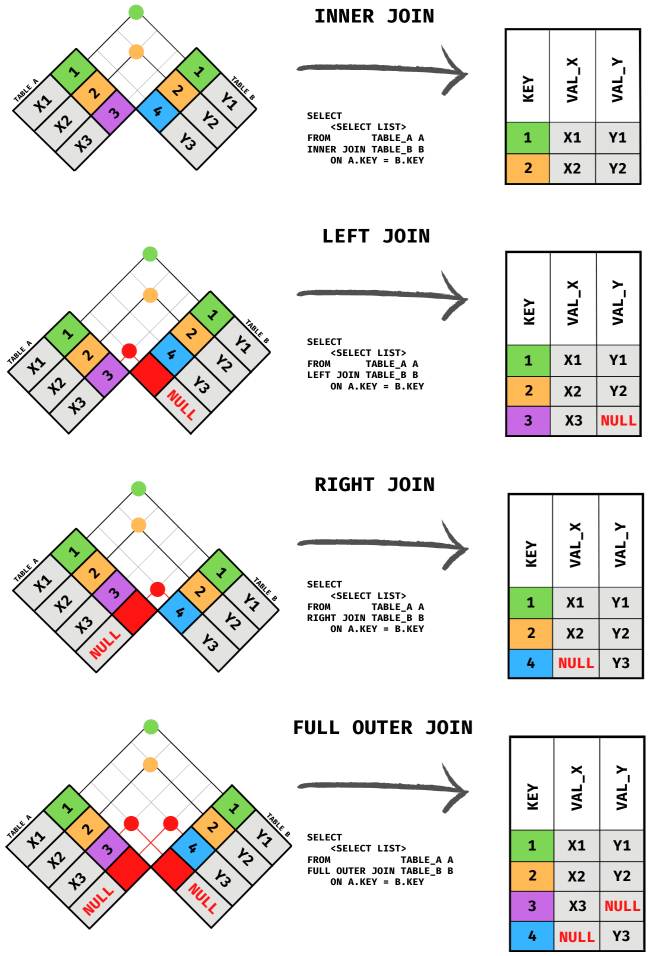
Ejemplo:

- Tabla `usuarios`: información personal
- Tabla `pedidos`: información de compras
- Cada pedido pertenece a un usuario

¿Cómo obtenemos el nombre del usuario junto con sus pedidos? 🤔

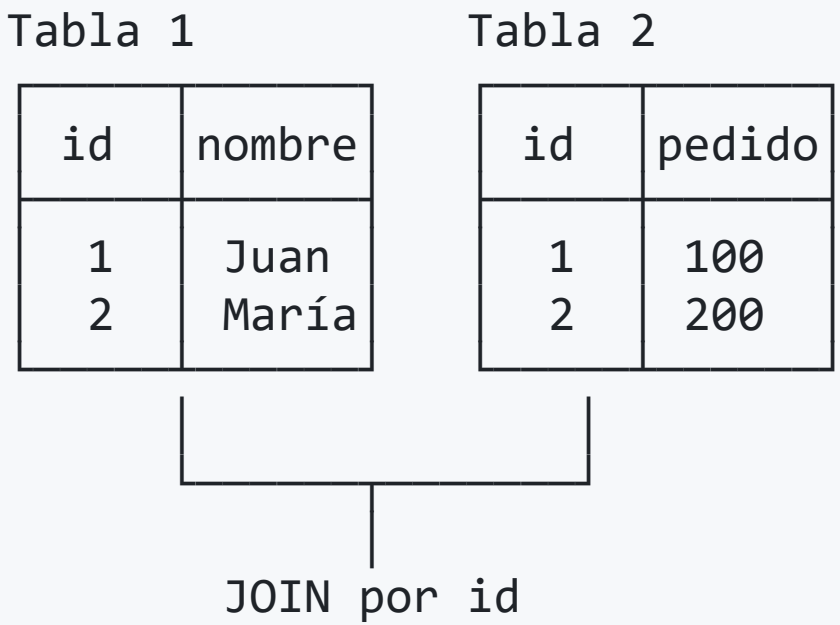
Cheat Sheet de JOINS

Cheat Sheet



¿Qué es un JOIN?

JOIN (o **INNER JOIN**) nos permite **combinar filas** de dos o más tablas basándonos en una relación entre ellas.



Sintaxis de JOIN

```sql
SELECT *
FROM tabla1
JOIN tabla2 ON tabla1.llave_foranea = tabla2.id;
```

Componentes importantes:

- `tabla1` : Tabla principal (izquierda)
- `tabla2` : Tabla que unimos (derecha)
- `ON` : Condición de unión
- `tabla1.llave_foranea` : La llave foránea en tabla1
- `tabla2.id` : El ID en tabla2

¿Por qué Usamos tabla.columna?

Cuando hacemos JOIN, **ambas tablas pueden tener columnas con el mismo nombre** (como `id`, `nombre`, `created_at`).

Solución: Usar `tabla.columna` para especificar de qué tabla viene cada columna.

```
SELECT usuarios.nombre, pedidos.total
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

`usuarios.nombre`: El nombre viene de la tabla usuarios

`pedidos.total`: El total viene de la tabla pedidos

JOIN: Relación Uno a Uno

Ejemplo: Un usuario tiene una dirección.

```
SELECT *
FROM usuarios
JOIN direcciones ON usuarios.id = direcciones.usuario_id;
```

¿Qué hace esto?

- Toma cada usuario
- Busca su dirección usando `usuario_id` en la tabla direcciones
- Combina ambas filas en el resultado

JOIN: Relación Uno a Muchos

Ejemplo: Un usuario tiene muchos pedidos.

```
SELECT *
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

Resultado: Si un usuario tiene 3 pedidos, aparecerá **3 veces** en el resultado (una vez por cada pedido).

JOIN: Relación Muchos a Muchos

Ejemplo: Usuarios y productos (a través de una tabla intermedia).

```
SELECT *
FROM pedidos
JOIN pedidos_has_productos ON pedidos.id = pedidos_has_productos.pedido_id
JOIN productos ON productos.id = pedidos_has_productos.producto_id;
```

Pasos:

1. Unimos `pedidos` con `pedidos_has_productos`
2. Luego unimos con `productos`
3. Obtenemos todos los productos de cada pedido

Seleccionando Columnas Específicas

En lugar de `SELECT *`, podemos elegir qué columnas mostrar:

```
SELECT
    usuarios.nombre,
    usuarios.email,
    pedidos.total,
    pedidos.fecha
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

Ventaja: Solo obtenemos los datos que necesitamos.

Usando Alias (AS) para Simplificar

Podemos usar **alias** para hacer el código más legible:

```sql
SELECT
    u.nombre,
    u.email,
    p.total,
    p.fecha
FROM usuarios AS u
JOIN pedidos AS p ON u.id = p.usuario_id;
```

`AS u`: Crea un alias "u" para usuarios

`AS p`: Crea un alias "p" para pedidos

💪 Ejercicio: JOIN Básico

¡Hora de practicar!

Usando las tablas `usuarios`, `pedidos` y `direcciones`:

1. Muestra el nombre del usuario, su email y la ciudad donde vive
2. Muestra todos los pedidos con el nombre completo del usuario y la fecha del pedido, ordenados por fecha (más recientes primero)
3. Encuentra usuarios que viven en "Madrid" y muestra sus pedidos
4. Muestra el nombre del usuario, su dirección completa (calle, ciudad) y el total de sus pedidos, solo para pedidos realizados en enero de 2024

Pistas:

- Necesitarás múltiples JOINs para unir usuarios → direcciones y usuarios → pedidos
- Usa ORDER BY para ordenar resultados
- Usa CONCAT() para combinar calle y ciudad
- Usa WHERE con condiciones de fecha (fecha >= '2024-01-01' AND fecha < '2024-02-01')

⏱️ **Tiempo:** 20 minutos

✅ Solución: JOIN Básico

1. Usuario con email y ciudad:

```sql
SELECT u.nombre, u.email, d.ciudad
FROM usuarios u
JOIN direcciones d ON u.id = d.usuario_id;
```

2. Pedidos con nombre completo ordenados por fecha:

```sql
SELECT CONCAT(u.nombre, ' ', u.apellido) AS nombre_completo, p.total, p.fecha
FROM usuarios u
JOIN pedidos p ON u.id = p.usuario_id
ORDER BY p.fecha DESC;
```

3. Usuarios de Madrid con sus pedidos:

```sql
SELECT u.nombre, u.apellido, p.total, p.fecha
FROM usuarios u
JOIN direcciones d ON u.id = d.usuario_id
JOIN pedidos p ON u.id = p.usuario_id
WHERE d.ciudad = 'Madrid';
```

4. Usuario, dirección completa y total de pedidos de enero 2024:

```sql
SELECT
    CONCAT(u.nombre, ' ', u.apellido) AS nombre_completo,
    CONCAT(d.calle, ', ', d.ciudad) AS direccion_completa,
    p.total
FROM usuarios u
JOIN direcciones d ON u.id = d.usuario_id
JOIN pedidos p ON u.id = p.usuario_id
WHERE p.fecha >= '2024-01-01' AND p.fecha < '2024-02-01';
```

LEFT JOIN

¿Cuál es la Diferencia?

JOIN (INNER JOIN):

- Solo muestra registros que **coinciden** en ambas tablas
- Si un usuario no tiene pedidos, **no aparece**

LEFT JOIN:

- Muestra **todos los registros** de la tabla izquierda
- Incluye registros que **no tienen coincidencias**
- Los valores faltantes aparecen como `NULL`

Visualizando la Diferencia

Tabla usuarios:

```
id | nombre
---|--------
1  | Juan
2  | María
3  | Pedro
```

Tabla pedidos:

```
id | usuario_id | total
---|------------|------
1  | 1          | 899.99
2  | 1          | 49.99
```

JOIN vs LEFT JOIN

Con JOIN:

```sql
SELECT usuarios.nombre, pedidos.total
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

Resultado:

```
nombre | total
-------|------
Juan   | 899.99
Juan   | 49.99
```

(María y Pedro no aparecen)

JOIN vs LEFT JOIN (cont.)

Con LEFT JOIN:

```sql
SELECT usuarios.nombre, pedidos.total
FROM usuarios
LEFT JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

Resultado:

```
nombre | total
-------|------
Juan   | 899.99
Juan   | 49.99
María  | NULL
Pedro  | NULL
```

(Todos los usuarios aparecen)

¿Cuándo Usar LEFT JOIN?

Usa LEFT JOIN cuando:

- ✅ Quieres ver **todos** los registros de la tabla principal
- ✅ Necesitas encontrar registros **sin relaciones**
- ✅ Quieres incluir información opcional

Ejemplo: "Muéstrame todos los usuarios, incluso si no tienen pedidos"

Ejemplo Práctico: Reseñas

```sql
SELECT usuarios.nombre, productos.nombre AS producto, resenas.comentario
FROM usuarios
LEFT JOIN resenas ON usuarios.id = resenas.usuario_id
LEFT JOIN productos ON resenas.producto_id = productos.id
WHERE usuarios.id = 2;
```

¿Qué hace?

- Muestra el usuario con id 2
- Incluye todas sus resenas de productos
- Si no tiene resenas, muestra NULL

Múltiples LEFT JOINs

Podemos unir **varias tablas** con LEFT JOIN:

```
SELECT
    usuarios.nombre,
    productos.nombre AS producto,
    resenas.calificacion,
    resenas.comentario
FROM usuarios
LEFT JOIN resenas ON usuarios.id = resenas.usuario_id
LEFT JOIN productos ON resenas.producto_id = productos.id
WHERE usuarios.id = 1;
```

Pasos:

1. Unimos usuarios con resenas
2. Luego unimos resenas con productos
3. Obtenemos productos que el usuario ha reseñado

Autounión: JOIN con la Misma Tabla

Ejemplo: Productos relacionados (un producto puede estar relacionado con otros productos).

```sql
-- Nota: Este ejemplo requiere una tabla adicional de relaciones entre productos
-- Para este esquema, podemos mostrar productos de la misma categoría:
SELECT
    p1.nombre AS producto1,
    p2.nombre AS producto2,
    c.nombre AS categoria
FROM productos p1
JOIN productos p2 ON p1.categoria_id = p2.categoria_id AND p1.id != p2.id
JOIN categorias c ON p1.categoria_id = c.id
WHERE p1.id = 1;
```

`AS p1, p2` : Creamos alias para usar la misma tabla dos veces y encontrar productos relacionados.

💪 Ejercicio: LEFT JOIN

¡Hora de practicar!

Usando las tablas `usuarios`, `resenas`, `productos` y `pedidos`:

1. Muestra todos los productos con sus resenas (incluyendo productos sin resenas), mostrando el nombre del producto y la calificación
2. Encuentra productos que NO tienen ninguna reseña
3. Muestra todos los usuarios con la cantidad de resenas que han hecho (incluyendo usuarios con 0 resenas)
4. BONUS: Encuentra usuarios que tienen pedidos pero NO han hecho ninguna reseña

Pistas:

- LEFT JOIN incluye todos los registros de la izquierda
- WHERE puede filtrar por NULL para encontrar registros sin relaciones
- COUNT() con LEFT JOIN puede contar 0 para registros sin coincidencias
- Puedes combinar LEFT JOIN con JOIN normal

⏱️ **Tiempo:** 20 minutos

✅ Solución: LEFT JOIN

1. Todos los productos con sus resenas (incluyendo sin resenas):

```
SELECT p.nombre AS producto, r.calificacion, r.comentario
FROM productos p
LEFT JOIN resenas r ON p.id = r.producto_id;
```

2. Productos sin resenas:

```
SELECT p.nombre AS producto
FROM productos p
LEFT JOIN resenas r ON p.id = r.producto_id
WHERE r.id IS NULL;
```

3. Usuarios con cantidad de resenas (incluyendo 0):

```sql
SELECT
    u.nombre,
    u.apellido,
    COUNT(r.id) AS total_reseñas
FROM usuarios u
LEFT JOIN resenas r ON u.id = r.usuario_id
GROUP BY u.id, u.nombre, u.apellido;
```

4. BONUS: Usuarios con pedidos pero sin resenas:

```sql
SELECT DISTINCT u.nombre, u.apellido
FROM usuarios u
JOIN pedidos p ON u.id = p.usuario_id
LEFT JOIN resenas r ON u.id = r.usuario_id
WHERE r.id IS NULL;
```

GROUP BY: Agrupando Resultados

¿Qué es GROUP BY?

GROUP BY nos permite **agrupar filas** que tienen el mismo valor en una columna y luego aplicar funciones agregadas a cada grupo.

Ejemplo:

- Agrupar pedidos por usuario
- Contar cuántos pedidos tiene cada usuario
- Calcular el total gastado por cada usuario

Sintaxis de GROUP BY

```
SELECT columna, FUNCION_AGREGADA(otra_columna)
FROM tabla
GROUP BY columna;
```

Regla importante:

- Las columnas en SELECT deben estar en GROUP BY
- O deben ser funciones agregadas (COUNT, SUM, etc.)

Ejemplo: Contar Pedidos por Usuario

```sql
SELECT
    usuarios.nombre,
    COUNT(pedidos.id) AS total_pedidos
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id
GROUP BY usuarios.id, usuarios.nombre;
```

Resultado:

```
nombre | total_pedidos
-------|---------------
Juan   | 2
María  | 3
Pedro  | 1
```

Ejemplo: Sumar Total por Usuario

```sql
SELECT
    usuarios.nombre,
    SUM(pedidos.total) AS total_gastado
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id
GROUP BY usuarios.id, usuarios.nombre;
```

Resultado:

```
nombre | total_gastado
-------|---------------
Juan   | 949.98
María  | 709.97
Pedro  | 299.99
```

Combinando Funciones Agregadas

Podemos usar **múltiples funciones** en la misma consulta:

```
SELECT
    usuarios.nombre,
    COUNT(pedidos.id) AS total_pedidos,
    SUM(pedidos.total) AS total_gastado,
    AVG(pedidos.total) AS promedio_por_pedido
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id
GROUP BY usuarios.id, usuarios.nombre;
```

GROUP BY con WHERE

Podemos **filtrar antes** de agrupar:

```sql
SELECT
    usuarios.nombre,
    COUNT(pedidos.id) AS total_pedidos
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id
WHERE pedidos.fecha >= '2024-01-01'
GROUP BY usuarios.id, usuarios.nombre;
```

¿Qué hace?

- Filtra pedidos del 2024 en adelante
- Luego agrupa por usuario

GROUP BY con HAVING

HAVING filtra **después** de agrupar (similar a WHERE pero para grupos):

```
SELECT
    usuarios.nombre,
    COUNT(pedidos.id) AS total_pedidos
FROM usuarios
JOIN pedidos ON usuarios.id = pedidos.usuario_id
GROUP BY usuarios.id, usuarios.nombre
HAVING COUNT(pedidos.id) > 2;
```

Resultado: Solo usuarios con más de 2 pedidos.

💪 Ejercicio: GROUP BY

¡Hora de practicar!

Usando las tablas disponibles:

1. Agrupa pedidos por mes (año y mes) y muestra cuántos pedidos se hicieron cada mes
2. Calcula el promedio de calificación de resenas por producto (solo productos con resenas)
3. Encuentra la categoría con más productos y muestra cuántos productos tiene
4. Muestra usuarios que han hecho más de 1 pedido, junto con el total gastado y el promedio por pedido

5. BONUS: Agrupa productos por categoría y muestra el precio promedio, el precio más alto y el más bajo de cada categoría

Pistas:

- DATE_FORMAT(fecha, '%Y-%m') agrupa por mes
- GROUP BY puede agrupar por múltiples columnas
- HAVING filtra después de agrupar
- Puedes usar múltiples funciones agregadas en la misma consulta
- LEFT JOIN puede ser útil para incluir categorías sin productos

⏱️ **Tiempo:** 25 minutos

✅ Solución: GROUP BY

1. Pedidos agrupados por mes:

```sql
SELECT
    DATE_FORMAT(fecha, '%Y-%m') AS mes,
    COUNT(*) AS total_pedidos
FROM pedidos
GROUP BY DATE_FORMAT(fecha, '%Y-%m')
ORDER BY mes;
```

2. Promedio de calificación por producto:

```sql
SELECT
    p.nombre AS producto,
    AVG(r.calificacion) AS promedio_calificacion
FROM productos p
JOIN resenas r ON p.id = r.producto_id
GROUP BY p.id, p.nombre;
```

3. Categoría con más productos:

```sql
SELECT
    c.nombre AS categoria,
    COUNT(pr.id) AS total_productos
FROM categorias c
LEFT JOIN productos pr ON c.id = pr.categoria_id
GROUP BY c.id, c.nombre
ORDER BY total_productos DESC
LIMIT 1;
```

4. Usuarios con más de 1 pedido:

```sql
SELECT
    u.nombre,
    COUNT(p.id) AS total_pedidos,
    SUM(p.total) AS total_gastado,
    AVG(p.total) AS promedio_por_pedido
FROM usuarios u
JOIN pedidos p ON u.id = p.usuario_id
GROUP BY u.id, u.nombre
HAVING COUNT(p.id) > 1;
```

5. BONUS: Estadísticas de precios por categoría:

```sql
SELECT
    c.nombre AS categoria,
    COUNT(pr.id) AS total_productos,
    AVG(pr.precio) AS precio_promedio,
    MAX(pr.precio) AS precio_maximo,
    MIN(pr.precio) AS precio_minimo
FROM categorias c
LEFT JOIN productos pr ON c.id = pr.categoria_id
GROUP BY c.id, c.nombre;
```

🔐 Restricciones de Llaves Foráneas

¿Qué son las Restricciones?

Las **restricciones de llaves foráneas** son reglas que garantizan la **integridad de los datos** cuando eliminamos o actualizamos registros relacionados.

Problema sin restricciones:

- Si eliminas un usuario que tiene pedidos
- Los pedidos quedan "huérfanos" (con un usuario_id que no existe)

Tipos de Restricciones (ON DELETE)

Cuando eliminas un registro padre, ¿qué pasa con los hijos?

Opciones:

- **RESTRICT**: No permite eliminar si hay registros relacionados
- **CASCADE**: Elimina automáticamente los registros relacionados
- **SET NULL**: Establece la llave foránea en NULL
- **NO ACTION**: Igual que RESTRICT

RESTRICT: Prevenir Eliminación

RESTRICT impide eliminar un registro si tiene relaciones:

```
-- Si intentas eliminar un usuario con pedidos:
DELETE FROM usuarios WHERE id = 1;
-- ❌ Error: No se puede eliminar porque tiene pedidos
```

¿Cuándo usar?

- Cuando los datos relacionados son importantes
- Cuando quieres prevenir eliminaciones accidentales

CASCADE: Eliminar en Cascada

CASCADE elimina automáticamente los registros relacionados:

```
-- Si eliminas un usuario con pedidos:
DELETE FROM usuarios WHERE id = 1;
-- ✅ Elimina el usuario Y todos sus pedidos
```

⚠️ **Cuidado:** Esto puede eliminar muchos datos. Úsalo con precaución.

SET NULL: Establecer NULL

SET NULL establece la llave foránea en NULL cuando se elimina el padre:

```
-- Si eliminas un usuario:
DELETE FROM usuarios WHERE id = 1;
-- ✅ El usuario se elimina
-- ✅ Los pedidos.usuario_id se ponen en NULL
```

¿Cuándo usar?

- Cuando quieres conservar los registros hijos
- Cuando la relación es opcional

Configurando Restricciones en MySQL Workbench

1. Abre tu ERD en MySQL Workbench
2. Haz clic en la tabla que tiene la llave foránea
3. Ve a la pestaña **Foreign Keys**
4. Selecciona la llave foránea
5. Configura **On Delete** con la opción deseada
6. Aplica los cambios con **Forward Engineer**

💪 Ejercicio: Restricciones

¡Hora de practicar!

En MySQL Workbench:

1. Abre tu esquema `tienda` con las tablas usuarios y pedidos
2. Configura la restricción ON DELETE en la relación entre usuarios y pedidos
3. Prueba eliminar un usuario con pedidos usando RESTRICT (debe fallar)
4. Cambia a CASCADE y prueba de nuevo (debe eliminar usuario y pedidos)
5. Observa la diferencia en el comportamiento

6. BONUS: Prueba SET NULL en la relación productos-categorias

Pistas:

- Ve a Foreign Keys en las propiedades de la tabla pedidos
- On Delete tiene las opciones disponibles
- Prueba con datos de ejemplo (usuario con id 1 tiene pedidos)

⏱️ **Tiempo:** 15 minutos

✅ Solución: Restricciones

Pasos en MySQL Workbench:

1. **Abrir el esquema** `tienda` en MySQL Workbench
2. **Ir a la tabla** `pedidos` → Click derecho → "Alter Table"
3. **Pestaña "Foreign Keys"** → Seleccionar la relación `pedidos_ibfk_1`
4. **Configurar "On Delete"** con las siguientes opciones:

Con RESTRICT (por defecto):

```
-- Intentar eliminar usuario con pedidos:
DELETE FROM usuarios WHERE id = 1;
-- ❌ Error: Cannot delete or update a parent row
```

Con CASCADE:

```
-- Cambiar a CASCADE en MySQL Workbench
-- Luego intentar eliminar:
DELETE FROM usuarios WHERE id = 1;
-- ✅ Elimina el usuario Y todos sus pedidos automáticamente
```

BONUS: SET NULL en productos-categorias:

- Ya está configurado en el esquema
- Si eliminas una categoría, los productos.categoria_id se ponen en NULL

💾 Exportar Base de Datos

¿Por qué Exportar?

Exportar nos permite:

- ✅ Hacer **backups** de nuestra base de datos
- ✅ **Compartir** la estructura y datos con otros
- ✅ **Migrar** datos a otro servidor
- ✅ **Versionar** cambios en la base de datos

Pasos para Exportar en MySQL Workbench

1. Abre MySQL Workbench
2. Conéctate a tu servidor local
3. Ve a **Server** → **Data Export**
4. Selecciona el esquema que quieres exportar
5. Selecciona las tablas (o todas)
6. Elige **Export to Self-Contained File**
7. Selecciona la ubicación y nombre del archivo
8. Marca **Create dump in a single transaction**
9. Marca **Include Create schema**
10. Ve a **Export Progress** y haz clic en **Start Export**

Opciones de Exportación

Export to Self-Contained File:

- Crea un archivo `.sql` completo
- Incluye estructura y datos
- Puede ejecutarse en cualquier servidor MySQL

Create dump in a single transaction:

- Garantiza consistencia de datos
- Recomendado para bases de datos grandes

Include Create schema:

- Incluye el comando CREATE SCHEMA
- Crea la base de datos si no existe

Importar una Base de Datos Exportada

Para importar un archivo `.sql`:

1. Abre MySQL Workbench
2. Conéctate a tu servidor
3. Ve a **Server → Data Import**
4. Selecciona **Import from Self-Contained File**
5. Elige el archivo `.sql`
6. Selecciona el esquema destino
7. Haz clic en **Start Import**

🎯 Resumen de Conceptos Clave

Funciones Agregadas

- **COUNT**: Cuenta registros
- **SUM**: Suma valores numéricos
- **AVG**: Calcula promedios
- **MAX**: Encuentra el valor máximo
- **MIN**: Encuentra el valor mínimo

Sintaxis:

```
SELECT FUNCION(columna) FROM tabla;
```

JOINS

- **JOIN (INNER JOIN)**: Solo registros que coinciden
- **LEFT JOIN**: Todos los registros de la izquierda
- **Múltiples JOINs**: Unir varias tablas
- **Autounión**: JOIN con la misma tabla usando alias

Sintaxis:

```
SELECT * FROM tabla1
JOIN tabla2 ON tabla1.llave = tabla2.id;
```

GROUP BY

- Agrupa filas con el mismo valor
- Permite aplicar funciones agregadas por grupo
- Se usa con COUNT, SUM, AVG, etc.
- HAVING filtra después de agrupar

Sintaxis:

```
SELECT columna, COUNT(*)
FROM tabla
GROUP BY columna;
```

Restricciones de Llaves Foráneas

- **RESTRICT**: Impide eliminar si hay relaciones
- **CASCADE**: Elimina registros relacionados
- **SET NULL**: Establece NULL en la llave foránea
- **NO ACTION**: Igual que RESTRICT

Configuración: En MySQL Workbench → Foreign Keys → On Delete

Exportar Base de Datos

1. Server → Data Export
2. Selecciona esquema y tablas
3. Export to Self-Contained File
4. Marca opciones importantes
5. Start Export

Importar: Server → Data Import → Import from Self-Contained File

🎉 ¡Felicidades!

Ya sabes usar funciones agregadas, JOINs y exportar bases de datos

[Has aprendido herramientas poderosas para trabajar con datos relacionados]

? Preguntas

¿Alguna duda sobre funciones agregadas, JOINs o exportación?

🙌 ¡Excelente Trabajo!

¡Nos vemos en la siguiente clase con más consultas avanzadas!

No olvides completar todos los ejercicios 📝