Capítulo 1

Introducción a MySQL y la sintaxis básica de consultas.

1.1. Creación de base de datos y usuarios.

Es importante evitar usar la base de datos `sys`, ya que esta es una base de datos del sistema y modificarla accidentalmente podría causar problemas graves. Para evitar esto, se recomienda crear bases de datos propias y gestionar usuarios con permisos específicos sobre estas. A continuación, se detalla el procedimiento:

Crear una base de datos: Desde el usuario `root`, se puede crear una nueva base de datos con el siguiente comando:

```
CREATE DATABASE nombre;
```

Para verificar las bases de datos existentes, se puede usar el comando:

```
SHOW DATABASES;
```

Si deseamos trabajar con una base de datos específica, debemos seleccionarla con:

```
USE nombre;
```

Crear un nuevo usuario: Una vez creada la base de datos, es buena práctica crear un usuario dedicado para trabajar con ella. Para crear un nuevo usuario, utiliza el comando:

```
CREATE USER 'nombreusuario'@'localhost' IDENTIFIED BY 'contraseña';
```

Otorgar permisos al usuario: Es necesario otorgar al usuario los permisos adecuados para interactuar con la base de datos. Esto se logra con el siguiente comando:

```
GRANT ALL PRIVILEGES ON nombre.* TO 'nombreusuario'@'localhost';
FLUSH PRIVILEGES;
```

El comando FLUSH PRIVILEGES actualiza los privilegios otorgados sin necesidad de reiniciar el servidor.

Configuración en Workbench: Después de configurar el usuario y la base de datos, puedes crear una nueva conexión en MySQL Workbench. Sigue estos pasos:

1. En el campo **Connection Name**, escribe un nombre descriptivo para la conexión.
2. En **Username**, ingresa el nombre de usuario creado.
3. Al intentar conectarte, se te pedirá la contraseña asignada.

Consideraciones al trabajar con nuevas bases de datos: Cada vez que necesites interactuar con una nueva base de datos, será necesario:

1. Acceder a la conexión `root` en Workbench (Local Instance).
2. Crear la nueva base de datos.
3. Otorgar permisos al usuario sobre la nueva base de datos utilizando los comandos mencionados.

Este enfoque asegura que trabajes de forma segura y evites posibles problemas derivados del uso de la base de datos `sys`.

1.2. Importación de base de datos en formato .sql.

Una de las formas más comunes de importar bases de datos es mediante la consola de comandos de Windows (CMD) o Windows PowerShell. A continuación, se describe el procedimiento paso a paso:

Acceder a MySQL desde la consola

1. Abre CMD o PowerShell.
2. Accede a MySQL con el siguiente comando:

```
mysql -uroot -p;
```

Después de ejecutar este comando, se te pedirá la contraseña de `root`. Al introducirla correctamente, accederás a la consola de MySQL.

3. Una vez dentro, puedes consultar las bases de datos existentes con:

```
SHOW databases;
```

Crear y seleccionar una base de datos

1. Crea una nueva base de datos donde se importarán los datos con el siguiente comando:

   ```
   CREATE DATABASE nombre;
   ```

2. Selecciona la base de datos recién creada:

   ```
   USE nombre;
   ```

Importar un archivo SQL: Para importar un archivo SQL, necesitas conocer la ruta completa del archivo en tu sistema. Por ejemplo, si el archivo `archivo.sql` está en el escritorio del usuario `gabri`, el comando sería:

```
SOURCE C:/Users/gabri/Desktop/archivo.sql;
```

Este comando cargará el contenido del archivo en la base de datos seleccionada. Si no se generan errores, la importación habrá sido exitosa.

Otorgar permisos al usuario: Después de importar la base de datos, otorga los permisos necesarios al usuario con los siguientes comandos:

```
GRANT ALL PRIVILEGES ON nombre.* TO 'usuario'@'localhost';
FLUSH PRIVILEGES;
```

El comando `FLUSH PRIVILEGES` asegura que los cambios de permisos se apliquen de inmediato.

Salir de MySQL: Para salir de la consola de MySQL, utiliza el comando:

```
EXIT;
```

Verificar la base de datos en MySQL Workbench

1. Abre MySQL Workbench y refresca la sección **SCHEMAS**.
2. Verifica que la base de datos importada aparece en la lista y que los datos se han cargado correctamente.

1.3. Importación de base de datos en formato .csv.

El proceso para cargar datos desde un archivo `.csv` en MySQL se realiza creando primero un esquema (schema) en MySQL Workbench y luego utilizando herramientas integradas para la importación. A continuación, se describe el procedimiento paso a paso:

Crear un esquema Un esquema actúa como un contenedor para las tablas y otros objetos de la base de datos. Puedes crear un esquema de dos maneras:

1. **Usando la interfaz gráfica de Workbench**:
   - Haz clic derecho en la sección **SCHEMAS**.
   - Selecciona la opción **Create Schema...**.
   - Asigna un nombre al nuevo esquema y haz clic en **Apply**.
2. **Usando un comando SQL**: Si prefieres utilizar comandos, ejecuta lo siguiente en la consola de MySQL Workbench:

```
CREATE SCHEMA 'nombre_schema';
```

Donde `nombre_schema` es el nombre que deseas asignar al esquema.

Importar un archivo CSV: Una vez que el esquema ha sido creado, puedes cargar los datos del archivo `.csv` utilizando el **Table Data Import Wizard**. Sigue estos pasos:

1. Haz clic derecho sobre el esquema que acabas de crear en la sección **SCHEMAS**.
2. Selecciona la opción **Table Data Import Wizard**.
3. En el asistente de importación:
   - Selecciona el archivo `.csv` que deseas cargar.
   - Configura las opciones de importación, como el delimitador del archivo (por ejemplo, `,` o `;`) y si el archivo incluye encabezados.
   - Especifica el nombre de la tabla donde se importarán los datos o crea una nueva tabla durante el proceso.
4. Revisa las configuraciones y haz clic en **Apply** para iniciar la importación.

Verificar la importació: Una vez completada la importación, verifica que los datos se han cargado correctamente:

1. Expande el esquema en la sección **SCHEMAS**.
2. Haz clic derecho sobre la tabla importada y selecciona **Select Rows - Limit 1000** para visualizar los datos.

Capítulo 2

Consultas básicas.

2.1. SELECT con una sola tabla.

En una base de datos, la información se organiza en **tablas**, y cada tabla contiene **columnas** que representan los distintos tipos de datos almacenados. Las tablas suelen estar relacionadas entre sí mediante columnas comunes, lo que permite vincular y cruzar información entre diferentes tablas.

Seleccionar datos de una tabla: Para visualizar el contenido de una tabla, puedes utilizar el comando `SELECT`. Este comando permite extraer información de una o más columnas de la tabla deseada.

Seleccionar todas las columnas: Si deseas extraer todas las columnas de una tabla, usa el siguiente comando:

```
SELECT *
FROM nombre_tabla;
```

Aquí, el símbolo * indica que se seleccionarán todas las columnas disponibles en la tabla.

Seleccionar columnas específicas: Si solo necesitas ciertas columnas de la tabla, debes especificarlas en lugar de usar *. Por ejemplo:

```
SELECT columna1, columna2, ...
FROM nombre_tabla;
```

En este caso:

- Reemplaza `columna1`, `columna2`, etc., con los nombres de las columnas que deseas extraer.
- El orden en que escribas las columnas después de `SELECT` determinará el orden en que aparecerán en el resultado.

Ejemplo práctico Supongamos que tienes una tabla llamada `clientes` con las siguientes columnas:

- `id`
- `nombre`
- `apellido`
- `email`
- `telefono`

1. Para seleccionar toda la información de la tabla:

```
SELECT *
FROM clientes;
```

2. Para seleccionar únicamente los nombres y correos electrónicos:

```
SELECT nombre, email
FROM clientes;
```

El resultado contendrá una tabla con solo las columnas `nombre` y `email` en el orden especificado.

2.2. Alias de columnas.

En ocasiones, es útil asignar un alias a una columna para mejorar la legibilidad de los resultados o para mostrar un nombre más amigable. Un alias permite renombrar temporalmente una columna en el resultado de una consulta sin modificar el esquema de la base de datos.

Sintaxis para asignar un alias: Para dar un alias a una columna en una consulta, utiliza la cláusula `AS`. La sintaxis es la siguiente:

```
SELECT columna1 AS Nombre, columna2 AS "Alias de columna"
FROM nombre_tabla;
```

Alias sin comillas: Si el alias es una sola palabra y no coincide con ninguna palabra reservada de SQL, no es necesario usar comillas. Ejemplo:

```
SELECT columna1 AS Nombre
FROM nombre_tabla;
```

Alias con comillas: Si el alias contiene espacios, caracteres especiales o coincide con una palabra reservada de SQL, debes encerrarlo entre comillas dobles (") o comillas invertidas (“). Ejemplo:

```
SELECT columna2 AS "Alias de columna"
FROM nombre_tabla;
```

2.3. SELECT DISTINCT.

El comando `SELECT DISTINCT` es una variante de `SELECT` que permite eliminar valores duplicados en los resultados de una consulta, mostrando únicamente valores únicos de una o más columnas.

Sintaxis básica: La sintaxis para usar `SELECT DISTINCT` es:

```
SELECT DISTINCT columna1
FROM nombre_tabla;
```

En este caso:

- `columna1`: Es la columna de la cual se desea obtener valores únicos.
- `nombre_tabla`: Es la tabla desde donde se realiza la consulta.

2.4. Operadores aritméticos (+, -, *, /).

En MySQL, los operadores aritméticos permiten realizar cálculos matemáticos de manera similar a una calculadora. Estos operadores pueden

aplicarse tanto a valores literales como a columnas de tablas con datos numéricos. A continuación, se describen los operadores básicos con ejemplos prácticos.

Operadores básicos

- **Suma (+)**

  El operador `+` realiza la suma de dos números. Por ejemplo:

  ```
  SELECT 3+4;
  ```

  arrojará como resultado el valor 7, habiendo realizado la suma de los dos números suministrados.

- **Resta (-)** El operador - calcula la diferencia entre dos números. Por ejemplo:

  ```
  SELECT 3-4;
  ```

  arrojando como resultado -1.

- **Multiplicación (*)** El operador `*` realiza la multiplicación de dos números. Por ejemplo:

  ```
  SELECT 3*4;
  ```

  dando como resultado 12.

- **División (/)** La barra diagonal (`/`) realiza la división de dos números y devuelve un valor con decimales si es necesario. Por ejemplo:

  ```
  SELECT 41/5;
  ```

  Lo que arrojará como resultado el valor 8.2000.

- **División entera (DIV)** El operador `DIV` devuelve únicamente el cociente entero de una división. Por ejemplo:

```
SELECT 41 DIV 5;
```

el resultado será 8, dandonos unicamente el cociente entero de la división.

- **Módulo (MOD)**

  La función `MOD(x, y)` calcula el resto de la división de `x` entre `y`. Por ejemplo:

  ```
  SELECT MOD(41,5);
  ```

  Lo que nos dará como resultado 1, el cual es el resto de la división.

Los operadores aritméticos se vuelven especialmente útiles al aplicarlos a columnas de una tabla. Por ejemplo, si tienes una tabla llamada `ventas` con las columnas `precio` y `cantidad`, puedes calcular el total de cada venta de la siguiente forma:

```
SELECT precio, cantidad, (precio * cantidad) AS "Total Venta"
FROM ventas;
```

Otro ejemplo podría ser realizar divisiones o cálculos más complejos entre columnas:

```
SELECT columna1, (columna2 / columna1) AS "División columnas"
FROM nombre_tabla;
```

Consideraciones adicionales

- **Tipos de datos**: Asegúrate de que las columnas involucradas en operaciones aritméticas contengan valores numéricos para evitar errores.
- **Alias**: Usa alias (`AS`) para dar nombres claros a las columnas resultantes de operaciones, como en el ejemplo `"Total Venta"`.

- **Funciones combinadas**: Los operadores aritméticos pueden combinarse con funciones agregadas como `SUM` o `AVG` para cálculos más avanzados.

Con estos operadores, puedes realizar desde cálculos simples hasta análisis más complejos en tus consultas SQL.

Capítulo 3

Filtrando resultados de la consulta.

3.1. Cláusula WHERE.

La cláusula `WHERE` se utiliza para filtrar los resultados de una consulta, mostrando únicamente aquellas filas que cumplen con una condición específica. Es una de las herramientas más potentes en SQL, ya que permite seleccionar subconjuntos de datos basados en criterios definidos.

Sintaxis básica: La estructura básica para usar `WHERE` es la siguiente:

```
SELECT *
FROM nombre_tabla
WHERE columna1 = condición;
```

3.2. Operadores de comparación (=, <, >, <=, >=, <>).

Los operadores de comparación se utilizan para comparar valores y determinar si se cumple una relación específica entre ellos. En MySQL, los resultados de estas comparaciones se expresan como `1` (TRUE) o `0` (FALSE). Por ejemplo:

```
SELECT 4 < 5;
```

nos arroja como resultado el valor 1, equivalente a TRUE. Pero si usamos

```
SELECT 7 < 2;
```

Operadores disponibles

- **Menor que (`<`)**
- **Menor o igual que (`<=`)**
- **Mayor que (`>`)**
- **Mayor o igual que (`>=`)**
- **Igual a (`=`)**
- **Distinto de (`!=` o `<>`)**

Uso combinado con `WHERE`: Los operadores de comparación son especialmente útiles al combinarse con la cláusula `WHERE` para filtrar datos en las consultas, permitiendo seleccionar filas que cumplan con condiciones numéricas o de texto.

3.3. Operadores lógicos (AND, OR, NOT).

Los operadores lógicos sirven para operar entre valores booleanos (valores que pueden ser únicamente verdadero o falso).

Por ejemplo, si usamos el operador `AND` de la siguiente manera:

```
SELECT TRUE AND TRUE;
```

Obtendremos como valor `1`, significando que es verdadero. En cambio, si usamos:

```
SELECT TRUE AND FALSE;
```

Obtenemos `0` como valor, significando que es falso. El operador `AND` únicamente devolverá `1` si **ambas sentencias son verdaderas**.

En cambio, si usamos el operador `OR` de la siguiente manera:

```
SELECT TRUE OR TRUE;
```

Obtenemos como resultado el valor `1`, indicando que es verdadero. Si escribimos:

```
SELECT FALSE OR TRUE;
```

Obtenemos también el valor `1`, ya que uno de los valores es verdadero. Finalmente, si escribimos:

```
SELECT FALSE OR FALSE;
```

Obtenemos como resultado el valor `0`. Esto implica que el operador `OR`, a diferencia del operador `AND`, nos devolverá falso únicamente cuando **ambas sentencias son falsas**.

En términos matemáticos:

- El operador `AND` es equivalente a la **intersección**, devolviéndonos lo que hay en común entre ambas condiciones.
- El operador `OR` nos devuelve la **unión de condiciones**, ya que se tiene que cumplir una u otra, pero no necesariamente ambas a la vez.

El operador `NOT` se utiliza para negar un operador. Por ejemplo:

```
SELECT NOT TRUE;
```

Se obtiene el valor `0`, ya que estamos negando el `TRUE`. En cambio, al escribir:

```
SELECT NOT FALSE;
```

Obtenemos el valor `1`, ya que al negar el `FALSE` se convierte en `TRUE`.

Aplicación en consultas: Si queremos aplicar todo esto a consultas de bases de datos, un ejemplo típico en el que se impone que se cumplan dos condiciones a la vez sería el siguiente:

```
SELECT *
FROM nombre_tabla
WHERE columna1 = condicion1 AND columna2 = condicion2;
```

En cambio, si colocamos un `OR`:

```
SELECT *
FROM nombre_tabla
WHERE columna1 = condicion1 AND columna2 = condicion2;
```

Se obtendrán tanto los que cumplen la `condicion1` como los que cumplen la `condicion2`, mientras que con `AND` las dos condiciones se tienen que cumplir **al mismo tiempo**.

3.4. Operador NOT y != (diferencias y similitudes).

La diferencia principal entre los operadores `NOT` y `!=` es la siguiente:

- **!=**: Compara dos cantidades o valores y devuelve un resultado booleano (`1` si no son iguales, `0` si son iguales).
- **NOT**: Invierte un valor booleano, es decir, cambia un `TRUE` por `FALSE` o viceversa.

Ejemplo típico: Un ejemplo común de uso de estos operadores se puede aplicar para comparar el género masculino o femenino:

```
SELECT *
FROM nombre_tabla
WHERE gender != 'M';
```

Lo que nos devolvería una tabla con todos los géneros femeninos.

En cambio, si queremos usar `NOT`, se escribiría así:

```
SELECT *
FROM nombre_tabla
WHERE NOT gender = 'M';
```

3.5. Operador BETWEEN.

El operador `BETWEEN` sirve para poder extraer consultas que tengan en la cláusula `WHERE` un rango. La sintaxis es la siguiente:

```
SELECT *
FROM nombre_tabla
WHERE columna1 BETWEEN rango1 AND rango2;
```

El resultado de esta consulta nos daría una tabla con los valores entre `rango1` y `rango2`. Otra forma de escribir lo mismo, pero utilizando otros operadores, sería:

```
SELECT *
FROM nombre_tabla
WHERE columna1 >= rango1 AND columna1 <= rango2;
```

Obteniéndose exactamente el mismo resultado, aunque con una sintaxis menos limpia que la que se obtiene usando `BETWEEN`. Además, podemos usar el operador `NOT` en combinación con `BETWEEN` para obtener los valores fuera del rango indicado:

```
SELECT *
FROM nombre_tabla
WHERE columna1 NOT BETWEEN rango1 AND rango2;
```

Mientras que con otros operadores se escribiría de la siguiente manera:

```
SELECT *
FROM nombre_tabla
WHERE columna1 > rango2;
```

3.6. Operador LIKE.

El operador `LIKE` sirve para encontrar patrones dentro de cadenas. Por ejemplo, si queremos hacer un filtro con `WHERE` para una columna pero obteniendo únicamente los valores con una segunda condición, se emplea `LIKE`. Por ejemplo:

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "XXX";
```

El operador `LIKE` en SQL utiliza dos comodines:

- Si se quiere buscar todos los elementos de una tabla de nombres que empiecen por G, se escribiría:

  ```
  SELECT *
  FROM nombre_tabla
  WHERE columna1 LIKE "G%";
  ```

  Mientras que si queremos buscar todos los elementos que tienen en su segunda letra una E, escribiríamos:

  ```
  SELECT *
  FROM nombre_tabla
  WHERE columna1 LIKE "_e%";
  ```

  Si queremos buscar todos los individuos cuyo nombre termina en O, escribiríamos:

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "%o";
```

Y si quisiéramos todos los que terminan en O y empiezan con P, escribiríamos:

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "p%o";
```

Si quisiéramos buscar en una columna todos los elementos que contengan, por ejemplo, la palabra `New`, podríamos escribir:

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "%New%";
```

Otro ejemplo interesante sería encontrar todas las personas nacidas en 1953 (para una fecha en formato AAAA-MM-DD):

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "1953%";
```

Mientras que si queremos buscar todos los nacidos en el mes 09, se escribiría:

```
SELECT *
FROM nombre_tabla
WHERE columna1 LIKE "%09___";
```

3.7. Operador IN.

El operador `IN` es equivalente a la concatenación de distintas condiciones que podríamos escribir con el operador `OR`. Por ejemplo, si queremos aplicar distintas condiciones con el operador `OR`, escribiríamos:

```
SELECT *
FROM nombre_tabla
WHERE columna1 = condicion1 OR columan1 = condicion2;
```

Esto nos daría una tabla donde la `condicion1` o la `condicion2` son aplicables. En cambio, se puede escribir de forma mucho más limpia el mismo código usando el operador `IN`:

```
SELECT *
FROM nombre_tabla
WHERE columna1 IN (condicion1, condicion2);
```

Obteniéndose el mismo resultado que con el código anterior, pero de forma mucho más limpia y eficiente.

Capítulo 4

Limitando y ordenando resultados.

4.1. Claúsula ORDER BY.

La cláusula `ORDER BY` nos sirve para ordenar los resultados de una consulta. Por ejemplo, si queremos ordenar una columna alfabéticamente o numéricamente, se usaría:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1;
```

Esto nos arroja una columna ordenada numéricamente (de menor a mayor) o alfabéticamente (de A a Z). El comando `ORDER BY` se aplica siempre al final de una consulta para poder ordenar los resultados de dicha consulta.

4.2. Operador IS NULL, IS NOT NULL.

Los operadores `NULL` y `NOT NULL` sirven para filtrar las consultas en determinadas circunstancias. Por ejemplo, si tenemos registros con valores `NULL` en una tabla y no queremos que aparezcan, podemos aplicar lo siguiente:

```
SELECT *
FROM nombre_tabla
WHERE columna1 IS NOT NULL;
```

Esto nos arrojaría una tabla con resultados con valores distintos a `NULL`. Es importante tener en cuenta que escribir lo siguiente nos arrojaría el mismo resultado:

```
SELECT *
FROM nombre_tabla
WHERE NOT columna1 IS NULL;
```

En cambio, si usamos `IS NULL`, se nos mostrará una tabla con los valores `NULL`:

```
SELECT *
FROM nombre_tabla
WHERE columna1 IS NULL;
```

4.3. Ordenamiento ascendente y descendente.

Se ha visto en la sección anterior que cuando se emplea el comando `ORDER BY`, los resultados aparecen ordenados alfabéticamente (de la A a la Z) o numéricamente (de menor a mayor). Sin embargo, si quisiéramos ordenarlos de forma **descendente**, se puede usar el comando `DESC` Por ejemplo:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1 DESC;
```

Por otro lado, existe el comando `ASC`, que nos arrojaría el mismo resultado que obtenemos cuando no ponemos `DESC`. Esto significa que el código `ASC` viene por defecto al usar `ORDER BY`.

Además, es posible **desordenar los valores de una columna**, y para ello se utiliza el comando `RAND()` de la siguiente forma:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1 RAND();
```

4.4. Clausula LIMIT.

La cláusula `LIMIT` nos permite limitar la cantidad de resultados que devuelve una consulta. Por ejemplo, para obtener los **5 primeros valores más altos de la tabla**, se escribiría de la siguiente forma:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1 DESC
LIMIT 5;
```

Por otro lado, si quisiéramos obtener **5 valores de forma aleatoria**, se escribiría así:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1 RAND()
LIMIT 5;
```

A veces, el comando `LIMIT` puede aparecer con **dos valores**, lo que indica el punto de inicio y la cantidad de filas a recuperar. Por ejemplo:

```
SELECT *
FROM nombre_tabla
LIMIT 0,5;
```

En este caso, el primer valor `0` indica la posición de inicio (la primera fila), y el segundo valor `5` la cantidad de filas a recuperar. Esto es equivalente a usar simplemente `LIMIT 5`. Sin embargo, se puede especificar un inicio diferente para obtener un subconjunto de resultados. Por ejemplo:

```
SELECT *
FROM nombre_tabla
LIMIT 5,5;
```

En este caso, se obtendrán **5 resultados a partir de la quinta fila**. Finalmente, si se desea obtener los últimos **X valores de una columna**, se puede escribir de la siguiente manera:

```
SELECT *
FROM nombre_tabla
ORDER BY columna1 DESC
LIMIT X;
```

Esto devuelve una tabla con los últimos **X valores de la tabla**, ya que se ha ordenado de forma descendente utilizando el comando `DESC`.

Capítulo 5

Trabajando con funciones de agregación.

5.1. Funciones de agregación: COUNT, SUM, AVG, MAX, MIN.

Las funciones de agregación permiten realizar cálculos en una columna específica y devuelven un único valor como resultado, lo cual es útil para analizar datos. Por ejemplo, para obtener la **suma de todos los valores de una columna**, se utiliza la función `SUM`:

```
SELECT SUM(columna1)
FROM nombre_tabla;
```

Otra función importante es la de **promedio**, representada por `AVG`. Se aplica de la siguiente forma:

```
SELECT AVG(columna1)
FROM nombre_tabla;
```

Además, la función **COUNT** se emplea para contar la cantidad de elementos en una tabla o columna. Por ejemplo, para contar el total de filas en una tabla, se usa:

```
SELECT COUNT(*)
FROM nombre_tabla;
```

Finalmente, las funciones `MIN` y `MAX` permiten encontrar el valor mínimo y máximo en una columna respectivamente. Se utilizan de la siguiente manera:

```
SELECT MIN(columna1) AS "Valor mínimo columna1",
MAX(columna1) AS "Valor máximo columna1"
FROM nombre_tabla;
```

Estas funciones son esenciales para realizar análisis estadísticos sobre los datos de una base de datos.

5.2. Cláusula GROUP BY.

La cláusula `GROUP BY` en MySQL se utiliza para **agrupar filas que comparten un valor común en una columna específica**. Esto es muy útil cuando se combinan con funciones de agregación como `SUM`, `AVG`, `COUNT`, `MIN`, o `MAX`, ya que permiten aplicar estos cálculos a cada grupo de manera individual.

Ejemplo 1: Agrupar por continentes para calcular la superficie total: Si queremos calcular la superficie total (`SUM`) de cada continente en una base de datos, agruparemos los datos por la columna `Contiene`:

```
SELECT Contiente, SUM(AreaSuperficial)
FROM nombre_tabla
GROUP BY Continente;
```

Este código agrupará las filas de la tabla según el campo `Continente` y calculará la suma de la columna `AreaSuperficial` para cada grupo.

Ejemplo 2: Agrupar por continentes para encontrar la esperanza de vida media: De manera similar, si queremos encontrar el promedio de esperanza de vida (`AVG`) para cada continente, podemos agrupar por el campo `Continente`:

```
SELECT Contiente, AVG(EsperanzaVida)
FROM nombre_tabla
GROUP BY Continente;
```

Este código calculará la esperanza de vida promedio para cada continente en la base de datos.

La clave es que `GROUP BY` clasifica los datos en subconjuntos basados en valores comunes en una columna, y luego las funciones de agregación calculan valores específicos para cada subconjunto.

5.3. Cláusula HAVING.

La cláusula `HAVING` en MySQL es similar a la cláusula `WHERE`, pero con una diferencia clave: mientras que `WHERE` filtra filas **antes de agrupar los datos**, `HAVING` filtra los resultados **después de aplicar el GROUP BY y las funciones de agregación**. Es decir, se utiliza para establecer condiciones sobre los grupos generados por `GROUP BY`.

Ejemplo: Filtrar grupos después de agrupar: Si queremos encontrar los continentes cuya **esperanza de vida promedio** es **mayor a 70 años**, la consulta sería:

```
SELECT Contiente, AVG(EsperanzaVida)
FROM nombre_tabla
GROUP BY Continente
HAVING AVG(EsperanzaVida) > 70;
```

Capítulo 6

Unión de tablas.

6.1. Conceptos de llaves primarias y foráneas.

Las **llaves primarias** y **llaves foráneas** son conceptos fundamentales en bases de datos relacionales que permiten establecer la integridad de los datos y definir relaciones entre tablas:

1. **Llave primaria (Primary Key):**
   - Es un campo (o combinación de campos) que se utiliza para identificar de manera **única cada fila en una tabla**.
   - No puede contener valores nulos y debe contener valores únicos.
   - Sirve como identificador único para cada registro en una tabla.
2. **Llave foránea (Foreign Key):**
   - Es una restricción que establece una relación entre dos tablas.
   - Actúa como un **enlace o referencia** entre un campo en una tabla (llamada tabla hija o referendo) y una clave primaria en otra tabla (llamada tabla padre o referencia).
   - Permite garantizar que los datos en las columnas estén referenciados de forma válida en la base de datos.

Ingeniería inversa en MySQL Workbench:

La ingeniería inversa es una herramienta clave para comprender las relaciones entre tablas en una base de datos existente. A través de esta función, se pueden generar Diagramas de Entidad-Relación (EER) que visualizan las conexiones y relaciones de llaves entre las tablas.

Pasos para aplicar ingeniería inversa:

1. En la barra de herramientas de MySQL Workbench, seleccionamos: **Database >Reverse Engineer...**
2. Elegimos la base de datos que queremos analizar y conectar al proceso.
3. Se genera automáticamente el **Diagrama de Entidad Relación (EER)** con las tablas y sus relaciones reflejadas visualmente.

¿Por qué es importante la ingeniería inversa?

- **Comprender relaciones:** Permite visualizar cómo las tablas están relacionadas entre sí.
- **Optimización de consultas:** Facilita el análisis para la creación de consultas SQL complejas.
- **Detección de errores:** Ayuda a identificar problemas en las relaciones o diseño de la base de datos.
- **Documentación:** Proporciona una representación gráfica clara para documentar la arquitectura de una base de datos.

Esta herramienta es muy útil tanto para el diseño de bases de datos como para el análisis de bases de datos existentes.

6.2. Introducción a JOINS.

Los comandos **JOIN** se utilizan en SQL para combinar información de dos o más tablas basándose en una relación establecida entre ellas.

Esto es esencial para trabajar con bases de datos relacionales, donde la información se almacena en múltiples tablas relacionadas.

Por defecto, el tipo de **JOIN** utilizado es el **INNER JOIN**, que devuelve solo las filas que tienen coincidencias en ambas tablas. Sin embargo, existen otros tipos de JOINS, como **LEFT JOIN** y **RIGHT JOIN**, cada uno con su comportamiento específico:

6.3. INNER JOIN.

Mediante **INNER JOIN**, podemos combinar dos tablas distintas obteniendo solo las filas que tienen elementos comunes en las columnas especificadas. Esto es útil para trabajar con datos relacionados que están almacenados en tablas diferentes. Para realizar un **INNER JOIN**, se puede usar la siguiente sintaxis:

```
SELECT *
FROM nombre_tabla1 t1
JOIN nombre_tabla2 t2
ON t1.columna1 = t2.columna1;
```

Explicación del Código

1. **Alias para las Tablas**:
   - Se asignan nombres cortos (alias) a las tablas (`t1` y `t2`) para hacer la consulta más legible y facilitar su escritura.
   - `t1` es el alias de `nombre_tabla1`, y `t2` es el alias de `nombre_tabla2`.
2. **Cláusula ON**:
   - Especifica las condiciones de unión entre las tablas. En este caso, `t1.columna1 = t2.columna1`.
   - Estas columnas deben tener una relación lógica, definida a partir del análisis previo, por ejemplo, consultando el **EER Diagram** para identificar las claves primarias y foráneas.

3. **Nombres de Columnas**: Las columnas referenciadas en la cláusula `ON` (`t1.columna1` y `t2.columna1`) no tienen que tener el mismo nombre en ambas tablas. La condición de igualdad se basa en la relación lógica entre los campos, no necesariamente en sus nombres.

El resultado de este comando será una nueva tabla que contendrá solo las filas de ambas tablas que cumplen la condición establecida en la cláusula `ON`.

6.4. LEFT JOIN.

La cláusula **LEFT JOIN** se utiliza para combinar dos tablas, devolviendo **todos los registros de la primera tabla (tabla izquierda)** y solo aquellos registros de la segunda tabla (tabla derecha) que cumplan la condición establecida. Si no hay coincidencia en la segunda tabla, se incluyen los datos de la primera tabla con valores `NULL` en las columnas de la segunda tabla. La sintaxis básica para realizar un **LEFT JOIN** es la siguiente:

```
SELECT *
FROM nombre_tabla1 t1
LEFT JOIN nombre_tabla2 t2
ON t1.columna1 = t2.columna1;
```

Explicación del Código

1. **Alias para las Tablas**:
   - `t1` es el alias para `nombre_tabla1`.
   - `t2` es el alias para `nombre_tabla2`.
2. **Cláusula ON**: Define la relación entre las tablas comparando `t1.columna1` con `t2.columna1`.
3. **Comportamiento del LEFT JOIN**:
   - La consulta devuelve **todas las filas de la primera tabla (nombre_tabla1)**.

- Se incluirán las filas de la segunda tabla (nombre_tabla2) que cumplan la condición especificada en la cláusula `ON`.
- Las filas de la segunda tabla que no cumplan la condición tendrán valores `NULL`.

Observaciones

1. **Valores NULL**: En la fila para `columna1 = 2`, como no existe una coincidencia en la segunda tabla, las columnas de `nombre_tabla2` devuelven `NULL`.
2. **Diferencia con INNER JOIN**: Mientras que el `INNER JOIN` devuelve únicamente las filas con coincidencias entre las tablas, el `LEFT JOIN` incluye todas las filas de la primera tabla, independientemente de si tienen coincidencia en la segunda tabla.

6.5. RIGHT JOIN.

La cláusula **RIGHT JOIN** se utiliza para combinar dos tablas, devolviendo **todos los registros de la segunda tabla (tabla derecha)** y solo aquellos registros de la primera tabla (tabla izquierda) que cumplan la condición establecida. Si no hay coincidencia en la primera tabla, se incluyen los datos de la segunda tabla con valores `NULL` en las columnas de la primera tabla. La sintaxis básica para realizar un **RIGHT JOIN** es la siguiente:

```
SELECT *
FROM nombre_tabla1 t1
RIGHT JOIN nombre_tabla2 t2
ON t1.columna1 = t2.columna1;
```

Explicación del Código

1. **Alias para las Tablas**:
   - `t1` es el alias para `nombre_tabla1`.

- `t2` es el alias para `nombre_tabla2`.

2. **Cláusula ON**: Define la relación entre las tablas comparando `t1.columna1` con `t2.columna1`.
3. **Comportamiento del RIGHT JOIN**:
   - La consulta devuelve **todas las filas de la segunda tabla (nombre_tabla2)**.
   - Se incluirán las filas de la primera tabla (nombre_tabla1) que cumplan la condición especificada en la cláusula `ON`.
   - Las filas de la primera tabla que no cumplan la condición tendrán valores `NULL`.

Observaciones

1. **Valores NULL**: En la fila para `columna1 = 4`, como no existe una coincidencia en la primera tabla, las columnas de `nombre_tabla1` devuelven `NULL`.
2. **Diferencia con LEFT JOIN**: Mientras que el `LEFT JOIN` devuelve todas las filas de la primera tabla, el `RIGHT JOIN` devuelve todas las filas de la segunda tabla.

6.6. JOIN con multiples tablas.

Cuando necesitamos unir múltiples tablas utilizando el comando `JOIN`, simplemente encadenamos los `JOIN` con la cláusula `ON` correspondiente para establecer las condiciones de relación entre las columnas de las tablas.

```
SELECT *
FROM nombre_tabla1 t1
JOIN nombre_tabla2 t2
ON t1.columna1 = t2.columna1
JOIN nombre_tabla3 t3
ON t3.columna1 = t2.columna1;
```

Explicación

1. **JOIN entre la primera y segunda tabla**: `JOIN` se aplica a la primera y segunda tabla utilizando la condición `ON t1.columna1 = t2.columna1`. Esto crea una tabla combinada donde las filas coinciden según el valor de `columna1`.
2. **JOIN con la tercera tabla**: Se agrega otro `JOIN`, relacionando la tercera tabla con la segunda a través de la condición `ON t3.columna1 = t2.columna1`.

Observaciones Importantes

- **Orden de las tablas**:
  El orden de los `JOIN` es importante, ya que define cómo se combinan las tablas.
- **Condiciones de relación**:
  Cada `JOIN` debe tener su propia cláusula `ON` para establecer la relación entre las columnas correspondientes.
- **Tipo de JOIN por defecto (INNER JOIN)**:
  Si no se especifica un tipo de `JOIN`, el tipo predeterminado es el `INNER JOIN`. Esto significa que solo se devuelven las filas que tienen coincidencias en todas las tablas relacionadas.

Capítulo 7

Consultas avanzadas.

7.1. Cláusula UNION, UNION ALL.

La cláusula UNION y UNION ALL en SQL se utilizan para combinar los resultados de dos o más consultas que devuelven el mismo número de columnas con nombres compatibles. Estas cláusulas permiten realizar la unión de conjuntos de resultados de manera flexible.

7.1.1. Diferencia entre UNION y UNION ALL

1. **UNION**:
   - Combina los resultados de dos o más consultas.
   - Elimina las filas duplicadas de la tabla resultante (comportamiento por defecto).
   - Más lento en comparación con `UNION ALL` si la operación implica eliminar duplicados.
2. **UNION ALL**:
   - Combina los resultados de dos o más consultas, pero **mantiene duplicados** en los resultados.
   - Más rápido que `UNION` porque no realiza la comprobación para eliminar duplicados.

7.1.2. Ejemplo Completo con UNION y UNION ALL

Supongamos que tenemos una base de datos con una tabla llamada `country`, que contiene información sobre países, sus continentes, población y área superficial:

Consulta 1: Población mayor a 150 millones

```
SELECT Name, Continent
FROM country
WHERE Population > 150000000;
```

Consulta 2: Superficie mayor a 9 millones

```
SELECT Name, Continent
FROM country
WHERE SurfaceArea > 9000000;
```

Ambas consultas devuelven una tabla con dos columnas: `Name` y `Continent`.

Uso de UNION ALL:

Si utilizamos `UNION ALL`, combinamos ambas consultas, manteniendo duplicados:

```
SELECT Name, Continent
FROM country
WHERE Population > 150000000
UNION ALL
SELECT Name, Continent
FROM country
WHERE SurfaceArea > 9000000;
```

¿Qué hace este código?

- Combina los resultados de ambas consultas.
- No elimina duplicados, es decir, los países que cumplen ambas condiciones aparecerán más de una vez en el resultado.

Uso de UNION (sin duplicados):

Si deseamos que no haya duplicados en el resultado final, utilizamos `UNION`:

```
SELECT Name, Continent
FROM country
WHERE Population > 150000000
UNION
SELECT Name, Continent
FROM country
WHERE SurfaceArea > 9000000;
```

¿Qué hace este código?

- Combina los resultados de ambas consultas.
- Elimina automáticamente las filas duplicadas para garantizar que cada fila sea única en la tabla resultante.

7.1.3. Conclusión

- Si necesitas mantener duplicados, usa **UNION ALL**.
- Si necesitas eliminar duplicados, usa **UNION**.

7.2. Introducción a subconsultas.

Las **subconsultas** son una herramienta muy útil en SQL que permiten realizar consultas anidadas dentro de otra consulta. Su propósito es obtener resultados más complejos o específicos utilizando la salida de una consulta como entrada para otra. A continuación se detallan los tipos más comunes de subconsultas:

7.2.1. Subconsultas con cláusula SELECT.

Las subconsultas en una cláusula `SELECT` permiten agregar columnas adicionales a la consulta principal que provienen de otra tabla.

Formato básico:

```
SELECT columna1, (SELECT subconsulta) as alias
FROM nombre_tabla;
```

Explicación:

- `(SELECT subconsulta)` es la consulta anidada que devuelve un valor único.
- Se usa un alias para facilitar la referencia en el resultado.

Ejemplo:

Supongamos que tenemos dos tablas: `empleados` y `departamentos`. Queremos obtener una lista de empleados junto con el nombre de su departamento como información adicional.

```
SELECT nombre_empleado,
       (SELECT nombre_departamento
        FROM departamentos
        WHERE empleados.departamento_id = departamentos.departame
FROM empleados;
```

En este ejemplo:

- La subconsulta selecciona el `nombre_departamento` correspondiente a cada empleado, comparando el campo `departamento_id`.
- El resultado tiene una nueva columna (`departamento`) agregada.

7.2.2. Subconsultas con cláusula WHERE.

Se utilizan para establecer condiciones basadas en la salida de una subconsulta. A menudo se combina con el operador `IN`.

```
SELECT columna1
FROM nombre_tabla
WHERE columna IN (SELECT subconsulta);
```

Explicación: `IN (SELECT subconsulta)` devuelve un conjunto de valores que la consulta principal puede usar para filtrar datos.

Ejemplo:

Supongamos que tenemos una base de datos con las tablas `empleados` y `departamentos`. Queremos obtener los empleados que trabajan en departamentos específicos, definidos por una subconsulta.

```
SELECT nombre_empleado
FROM empleados
WHERE departamento_id IN (
    SELECT departamento_id
    FROM departamentos
    WHERE ubicacion = 'Nueva York'
);
```

En este ejemplo:

- La subconsulta selecciona los `departamento_id` de la tabla `departamentos` cuya `ubicacion` es `'Nueva York'`.
- La consulta principal devuelve solo los empleados que trabajan en esos departamentos.

7.2.3. Subconsultas con cláusula HAVING.

Se utilizan con `GROUP BY` para aplicar filtros a los resultados agrupados. Estas subconsultas permiten realizar condiciones sobre cálculos como promedios, sumas, conteos, etc.

```
SELECT columna1
FROM nombre_tabla
GROUP BY columna1
HAVING columna1 IN (SELECT subconsulta);
```

Explicación:

- La cláusula `HAVING` actúa como un filtro para grupos generados por el `GROUP BY`.
- La subconsulta permite comparar estos grupos con ciertos valores.

Ejemplo:

Supongamos que tenemos la tabla `ventas`, y queremos identificar las ciudades donde la suma de las ventas supera un umbral definido:

```
SELECT ciudad
FROM ventas
GROUP BY ciudad
HAVING SUM(ventas) > (
    SELECT AVG(ventas)
    FROM ventas
);
```

En este ejemplo:

- La subconsulta calcula el promedio de las ventas en toda la tabla `ventas`.
- La consulta principal devuelve las ciudades donde la suma de las ventas es mayor que este promedio.

7.2.4. Subconsultas con cláusula EXIST.

El operador `EXISTS` permite verificar la existencia de ciertos registros en una subconsulta. Devuelve `TRUE` si la subconsulta devuelve filas y `FALSE` si no devuelve ninguna.

Formato básico:

```
SELECT columna1
FROM nombre_tabla
WHERE EXISTS (SELECT subconsulta);
```

Explicación: La subconsulta se ejecuta, y el resultado determina si la condición es verdadera o falsa para filtrar los datos de la consulta principal.

Ejemplo:

```
SELECT nombre_cliente
FROM clientes c
WHERE EXISTS (
    SELECT 1
    FROM pedidos p
    WHERE p.id_cliente = c.id_cliente
);
```

En este ejemplo:

- La subconsulta verifica si hay al menos una fila en la tabla `pedidos` para cada cliente en la tabla `clientes`.
- Si existe un registro correspondiente en la tabla `pedidos`, el cliente es incluido en los resultados.

Supongamos que tenemos las tablas `clientes` y `pedidos`. Queremos obtener la lista de clientes que tienen al menos un pedido registrado:

7.2.5. Conclusión

Tipos de Subconsultas:

1. **Con SELECT:** Permiten agregar columnas adicionales con resultados de otra tabla.
2. **Con WHERE + IN:** Permiten filtrar filas comparando una condición con otra tabla.
3. **Con HAVING + GROUP BY:** Realizan filtros sobre grupos calculados.
4. **Con EXISTS:** Devuelven un valor booleano para filtrar la consulta principal.

Consideraciones importantes:

- Las subconsultas son solo para **leer información**, **no para insertar datos**.
- Pueden afectar el rendimiento si las subconsultas son muy complejas o la base de datos es muy grande.
- Es recomendable usar índices si se planean realizar subconsultas frecuentes para mejorar el rendimiento.

Las subconsultas son herramientas fundamentales para realizar operaciones avanzadas en SQL, permitiendo soluciones más flexibles y personalizadas para consultas complejas.

7.3. Common Table Expression (CTE).

Las Common Table Expressions (CTE) son una característica avanzada en SQL que permite crear consultas intermedias reutilizables dentro de una consulta principal. Se utilizan para simplificar consultas complejas, hacerlas más legibles y estructurar mejor la lógica de las mismas. Las CTE son una alternativa más clara y modular en comparación con las subconsultas tradicionales.

¿Qué es una CTE?

Una CTE es una consulta definida temporalmente que se puede referenciar en la consulta principal. Se define utilizando la cláusula `WITH`, y permite trabajar con datos intermedios de manera similar a una subconsulta, pero con una sintaxis más clara y mejor reutilización.

7.3.1. Formato básico:

Las CTE se crean con la siguiente sintaxis:

```
WITH nombre_cte AS (
    SELECT columna1, columna2
```

```
    FROM nombre_tabla
    WHERE condicion
)
SELECT *
FROM nombre_cte
WHERE columna1 = 'valor';
```

Componentes:

1. **WITH** $nombre_cte$ **AS (xxx)**: Define la consulta temporal que actúa como la CTE.
2. **Subconsulta dentro de la CTE**: Aquí escribimos la lógica que queremos reutilizar.
3. **Consulta principal**: Después de definir la CTE, la utilizamos en la consulta principal con su nombre.

7.3.2. Ejemplo práctico:

Supongamos que tenemos una tabla llamada `empleados` y queremos obtener una lista de empleados con salarios superiores al promedio de su departamento.

Paso 1: Crear la CTE para calcular el promedio por departamento

```
WITH promedio_departamento AS (
    SELECT departamento_id, AVG(salario) AS salario_promedio
    FROM empleados
    GROUP BY departamento_id
)
SELECT e.nombre, e.departamento_id,
e.salario, p.salario_promedio
FROM empleados e
JOIN promedio_departamento p ON e.departamento_id = p.departament
WHERE e.salario > p.salario_promedio;
```

Explicación del ejemplo:

1. **WITH** $promedio_departamento$ **AS (...)**: Aquí calculamos el salario promedio de cada departamento.
2. **Consulta principal**: Nos unimos a la CTE para comparar el salario de cada empleado con el promedio de su departamento.
3. **Filtro (WHERE e.salario** $> p.salario_promedio$**)**: Devuelve solo empleados cuyo salario es superior al salario promedio de su respectivo departamento.

7.3.3. Ventajas de las CTE:

1. **Código más claro y modular**: Permiten dividir consultas complejas en partes lógicas, lo que facilita la comprensión.
2. **Reutilización de la lógica**: Las CTE pueden ser referenciadas múltiples veces en una consulta.
3. **Soporte para consultas recursivas**: Las CTE permiten crear consultas recursivas, lo que es muy útil para trabajar con datos jerárquicos (por ejemplo, estructuras de árbol).

7.3.4. CTE Recursivas

Las CTE pueden ser **recursivas**, lo que significa que se pueden llamar a sí mismas de manera iterativa para resolver problemas jerárquicos.

Ejemplo con una jerarquía de empleados:

Supongamos que tenemos una tabla `empleados` con campos `id_empleado`, `nombre`, y `id_supervisor`. Queremos obtener la cadena completa de supervisores para cada empleado.

```
WITH RECURSIVE jerarquia_empleados AS (
    SELECT id_empleado, nombre, id_supervisor
    FROM empleados
    WHERE id_supervisor IS NULL
```

```
    UNION ALL
    SELECT e.id_empleado, e.nombre, e.id_supervisor
    FROM empleados e
    JOIN jerarquia_empleados j
    ON e.id_supervisor = j.id_empleado
)
SELECT *
FROM jerarquia_empleados;
```

Explicación del ejemplo:

1. **Primera parte (SELECT** $id_empleado$**, nombre,** $id_supervisor$ **FROM empleados WHERE** $id_supervisor$ **IS NULL)**: Obtiene los empleados en el nivel más alto de la jerarquía (sin supervisores).
2. **Segunda parte (UNION ALL)**: Se unen recursivamente los empleados a sus supervisores, construyendo la cadena de supervisión en cada paso.

7.3.5. Conclusión

Las **CTE** son una evolución de las subconsultas que ofrecen mejor claridad, reutilización y flexibilidad en las consultas. Mientras que las subconsultas funcionan bien para operaciones más simples, las CTE son más recomendables para consultas complejas, jerárquicas o que requieren cálculos intermedios reutilizables.

Por lo tanto:

- Usa **subconsultas** para tareas más simples y directas.
- Opta por **CTE** cuando trabajes con lógica compleja, múltiples pasos interdependientes, o requieras consultas recursivas.

Las CTE mejoran la legibilidad y la eficiencia de las consultas, lo que las convierte en una herramienta clave para desarrolladores y analistas de datos que trabajan con bases de datos grandes o con datos complejos.