



ESCUELA POLITÉCNICA NACIONAL ESCUELA DE FORMACIÓN DE TECNÓLOGOS



BASE DE DATOS

ASIGNATURA:	Base de datos
PROFESOR:	Ing. Yadira Franco R
PERÍODO ACADÉMICO:	2024-B

TÍTULO

PROYECTO FINAL

ESTUDIANTES

Christian David Marquez Yela

Justin Gabriel Imbaquingo Perez

Jairo Sebastián Betancourt Iza

Link repositorio GITHUB: https://github.com/Sebastian-Betancourt/Proyecto_Base_de_Datos.git

ÍNDICE

1. **Introducción**
 - 1.1 Objetivo del Proyecto
2. **Modelado de Base de Datos y Diccionario de Datos**
 - 2.1 Modelo Conceptual
 - 2.2 Modelo Lógico
 - 2.3 Modelo Físico
 - 2.4 Diccionario de Datos
 - 2.5 Restricciones de Integridad
3. **Seguridad, Auditoría y Control de Acceso**
 - 3.1 Políticas de Seguridad y Acceso
 - 3.2 Cifrado de Datos Sensibles
 - 3.3 Auditoría y Registro de Eventos
4. **Respaldo y Recuperación de Datos**
 - 4.1 Respaldo Completo
 - 4.2 Respaldo Incremental
 - 4.3 Respaldo en Caliente
5. **Optimización y Rendimiento de Consultas**
 - 5.1 Creación y Gestión de Índices
 - 5.2 Optimización de Consultas SQL
 - 5.3 Particionamiento de Tablas
6. **Procedimientos Almacenados, Vistas y Triggers**
 - 6.1 Procedimientos Almacenados
 - 6.2 Creación y Uso de Vistas
 - 6.3 Implementación de Triggers
7. **Monitoreo y Optimización de Recursos**
 - 7.1 Pruebas de Carga y Estrés
 - 7.2 Gestión de Índices y Recursos
8. **Git y Control de Versiones**
 - 8.1 Configuración del Repositorio
 - 8.2 Estrategias de Versionado y Colaboración
 - 8.3 Automatización de Pruebas
9. **Conclusiones y Recomendaciones**

1. Introducción

En la actualidad, los cines deben contar con sistemas eficientes para la gestión de sus operaciones, desde la administración de funciones y películas hasta la venta de boletos y el registro de pagos. Un sistema bien diseñado permite mejorar la experiencia del cliente, optimizar la asignación de salas y horarios, y garantizar un control preciso sobre las transacciones realizadas.

Este proyecto, desarrollado en equipo dentro de la asignatura de Bases de Datos, tiene como objetivo diseñar e implementar un sistema de gestión para un cine, utilizando una base de datos relacional. A través del análisis y modelado de datos, se crea una solución que permita almacenar y organizar información clave, como clientes, películas, funciones, boletos y pagos.

1.1 Objetivo del Proyecto

El objetivo de este proyecto es diseñar e implementar una base de datos que optimice la gestión de un cine, permitiendo el almacenamiento y consulta eficiente de la información relacionada con clientes, películas, funciones, boletos y pagos. Con esta solución, se busca mejorar la organización y administración de los datos, asegurando su integridad y disponibilidad.

A través de este proyecto, se aplica conceptos clave de bases de datos, como modelado, normalización y ejecución de consultas, con el fin de crear un sistema estructurado que facilite la toma de decisiones y la automatización de procesos dentro del cine.

2. Modelado de Base de Datos y Diccionario de Datos

Diseñar el modelo conceptual, lógico y físico

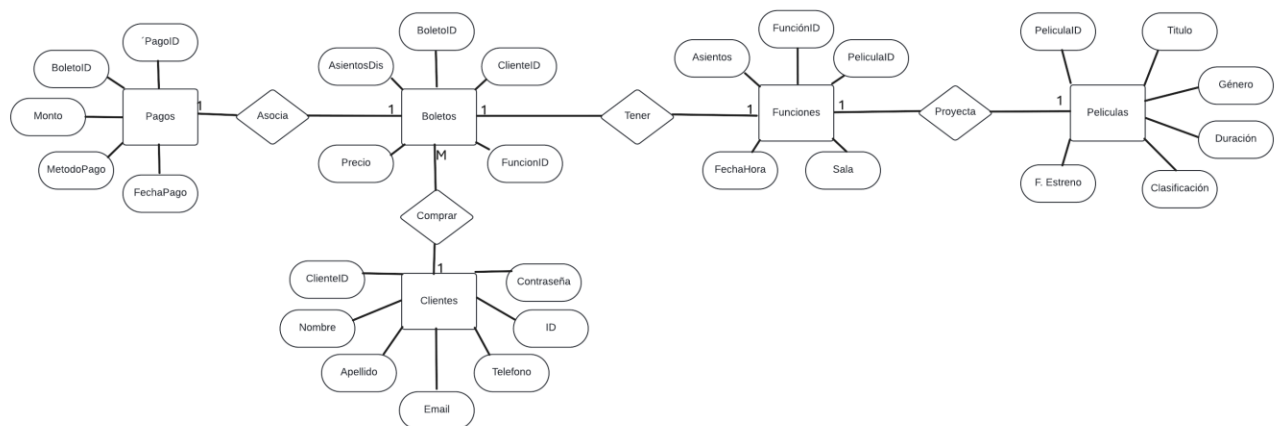
2.1 Modelo Conceptual

- **Entidades principales:**

1. **Cientes:** Información de los usuarios que compran boletos.
2. **Películas:** Detalles de las películas que se proyectan en el cine.
3. **Funciones:** Sesiones específicas de una película en una sala y horario.
4. **Boletos:** Boletos reservados por los clientes para una función.
5. **Pagos:** Transacciones asociadas a la compra de boletos.

- **Relaciones:**

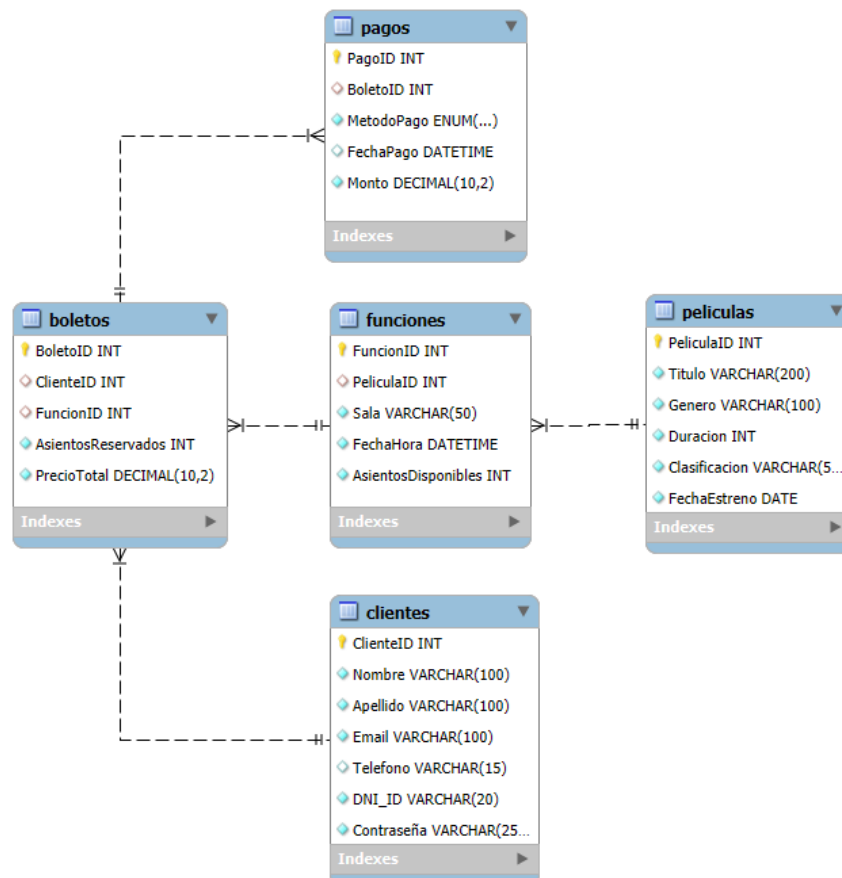
1. Un **Cliente** puede comprar varios **Boletos**.
2. Un **Boleto** está asociado a una **Función**.
3. Una **Función** está asociada a una **Película**.
4. Un **Boleto** puede tener un **Pago**.



2.2 Modelo Lógico

- **Atributos de cada entidad:**

- **Clientes:** ClienteID, Nombre, Apellido, Email, Teléfono, DNI/ID, Contraseña.
- **Películas:** PeliculaID, Título, Género, Duración, Clasificación, Fecha Estreno.
- **Funciones:** FuncionID, PeliculaID, Sala, FechaHora, AsientosDisponibles.
- **Boletos:** BoletoID, ClienteID, FuncionID, AsientosReservados, PrecioTotal.
- **Pagos:** PagoID, BoletoID, MétodoPago, FechaPago, Monto.



2.3 Modelo Físico

- Implementación en MySQL Workbench:
 - Crear las tablas con sus respectivos campos y tipos de datos.

```
4  -- Tabla Clientes
5  ● CREATE TABLE Clientes (
6      ClienteID INT AUTO_INCREMENT PRIMARY KEY,
7      Nombre VARCHAR(100) NOT NULL,
8      Apellido VARCHAR(100) NOT NULL,
9      Email VARCHAR(100) UNIQUE NOT NULL,
10     Telefono VARCHAR(15),
11     DNI_ID VARCHAR(20) UNIQUE NOT NULL,
12     Contraseña VARCHAR(255) NOT NULL
13 );

16 ● CREATE TABLE Peliculas (
17     PeliculaID INT AUTO_INCREMENT PRIMARY KEY,
18     Titulo VARCHAR(200) NOT NULL,
19     Genero VARCHAR(100) NOT NULL,
20     Duracion INT NOT NULL,
21     Clasificacion VARCHAR(50) NOT NULL,
22     FechaEstreno DATE NOT NULL
23 );

25  -- Tabla Funciones
26 ● CREATE TABLE Funciones (
27     FuncionID INT AUTO_INCREMENT PRIMARY KEY,
28     PeliculaID INT,
29     Sala VARCHAR(50) NOT NULL,
30     FechaHora DATETIME NOT NULL,
31     AsientosDisponibles INT NOT NULL,
32     FOREIGN KEY (PeliculaID) REFERENCES Peliculas(PeliculaID) ON DELETE CASCADE
33 );

35  -- Tabla Boletos
36 ● CREATE TABLE Boletos (
37     BoletoID INT AUTO_INCREMENT PRIMARY KEY,
38     ClienteID INT,
39     FuncionID INT,
40     AsientosReservados INT NOT NULL,
41     PrecioTotal DECIMAL(10, 2) NOT NULL,
42     FOREIGN KEY (ClienteID) REFERENCES Clientes(ClienteID) ON DELETE CASCADE,
43     FOREIGN KEY (FuncionID) REFERENCES Funciones(FuncionID) ON DELETE CASCADE
44 );

46  -- Tabla Pagos
47 ● CREATE TABLE Pagos (
48     PagoID INT AUTO_INCREMENT PRIMARY KEY,
49     BoletoID INT,
50     MetodoPago ENUM('Tarjeta', 'PayPal', 'Efectivo') NOT NULL,
51     FechaPago DATETIME DEFAULT CURRENT_TIMESTAMP,
52     Monto DECIMAL(10, 2) NOT NULL,
53     FOREIGN KEY (BoletoID) REFERENCES Boletos(BoletoID) ON DELETE CASCADE
54 );
```

Script:

- Establecer las relaciones entre las tablas usando claves primarias y foráneas.

```
CREATE DATABASE ProyectoBDD;
```

```
USE ProyectoBDD;
```

```
-- Tabla Clientes
```

```
CREATE TABLE Clientes (  
    ClienteID INT AUTO_INCREMENT PRIMARY KEY,  
    Nombre VARCHAR(100) NOT NULL,  
    Apellido VARCHAR(100) NOT NULL,  
    Email VARCHAR(100) UNIQUE NOT NULL,  
    Telefono VARCHAR(15),  
    DNI_ID VARCHAR(20) UNIQUE NOT NULL,  
    Contraseña VARCHAR(255) NOT NULL  
);
```

```
-- Tabla Películas
```

```
CREATE TABLE Peliculas (  
    PeliculaID INT AUTO_INCREMENT PRIMARY KEY,  
    Titulo VARCHAR(200) NOT NULL,  
    Genero VARCHAR(100) NOT NULL,  
    Duracion INT NOT NULL,  
    Clasificacion VARCHAR(50) NOT NULL,  
    FechaEstreno DATE NOT NULL  
);
```

```
-- Tabla Funciones
```

```
CREATE TABLE Funciones (  
    FuncionID INT AUTO_INCREMENT PRIMARY KEY,  
    PeliculaID INT,  
    Sala VARCHAR(50) NOT NULL,  
    FechaHora DATETIME NOT NULL,  
    AsientosDisponibles INT NOT NULL,
```

```

        FOREIGN KEY (PeliculaID) REFERENCES
        Peliculas(PeliculaID) ON DELETE CASCADE
    );

-- Tabla Boletos
CREATE TABLE Boletos (
    BoletoID INT AUTO_INCREMENT PRIMARY KEY,
    ClienteID INT,
    FuncionID INT,
    AsientosReservados INT NOT NULL,
    PrecioTotal DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (ClienteID) REFERENCES Clientes(ClienteID)
    ON DELETE CASCADE,
    FOREIGN KEY (FuncionID) REFERENCES
    Funciones(FuncionID) ON DELETE CASCADE
);

-- Tabla Pagos
CREATE TABLE Pagos (
    PagoID INT AUTO_INCREMENT PRIMARY KEY,
    BoletoID INT,
    MetodoPago ENUM('Tarjeta', 'PayPal', 'Efectivo') NOT NULL,
    FechaPago DATETIME DEFAULT CURRENT_TIMESTAMP,
    Monto DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (BoletoID) REFERENCES Boletos(BoletoID)
    ON DELETE CASCADE
);

```


2.4 Desarrollar un diccionario de datos detallado

Tabla: Clientes			
Nombre del Campo	Tipo de Dato	Restricciones	Descripción
CienteID	INT	PRIMARY KEY, AUTO_INCREMENT	
Nombre	VARCHAR(100)	NOT NULL	Nombre del cliente.
Apellido	VARCHAR(100)	NOT NULL	Apellido del cliente.
Email	VARCHAR(100)	UNIQUE, NOT NULL	Correo electrónico del cliente.
Telefono	VARCHAR(15)		Número de teléfono del cliente.
DNI_ID	VARCHAR(20)	UNIQUE, NOT NULL	Documento de identidad del cliente.
Contraseña	VARCHAR(255)	NOT NULL	Contraseña cifrada del cliente.
Relacionada con la tabla Boletos a través de CienteID.			
Tabla: Películas			
Nombre del Campo	Tipo de Dato	Restricciones	Descripción
PelículaID	INT	PRIMARY KEY, AUTO_INCREMENT	Identificador único de la película.
Título	VARCHAR(200)	NOT NULL	Título de la película.
Genero	VARCHAR(100)	NOT NULL	Género de la película.
Duracion	INT	NOT NULL	Duración de la película en minutos.
Clasificacion	VARCHAR(50)	NOT NULL	Clasificación por edad (ej: PG-13).
FechaEstreno	DATE	NOT NULL	Fecha de estreno de la película.
Relacionada con la tabla Funciones a través de PelículaID.			
Tabla: Funciones			
Nombre del Campo	Tipo de Dato	Restricciones	Descripción
FuncionID	INT	PRIMARY KEY, AUTO_INCREMENT	Identificador único de la función.
PelículaID	INT	FOREIGN KEY, NOT NULL	Identificador de la película proyectada.
Sala	VARCHAR(50)	NOT NULL	Sala donde se proyecta la función.
FechaHora	DATETIME	NOT NULL	Fecha y hora de la función.
AsientosDisponibles	INT	NOT NULL	Número de asientos disponibles.
Relacionada con la tabla Películas a través de PelículaID.			
Relacionada con la tabla Boletos a través de FuncionID.			
Tabla: Boletos			
Nombre del Campo	Tipo de Dato	Restricciones	Descripción
BoletoID	INT	PRIMARY KEY, AUTO_INCREMENT	Identificador único del boleto.
CienteID	INT	FOREIGN KEY, NOT NULL	Identificador del cliente que compró el boleto.
FuncionID	INT	FOREIGN KEY, NOT NULL	Identificador de la función.
AsientosReservados	INT	NOT NULL	Número de asientos reservados.
PrecioTotal	DECIMAL(10, 2)	NOT NULL	Precio total del boleto
Relacionada con la tabla Clientes a través de CienteID.			
Relacionada con la tabla Funciones a través de FuncionID.			
Relacionada con la tabla Pagos a través de BoletoID.			
Tabla: Pagos			
Nombre del Campo	Tipo de Dato	Restricciones	Descripción
PagoID	INT	PRIMARY KEY, AUTO_INCREMENT	Identificador único del pago.
BoletoID	INT	FOREIGN KEY, NOT NULL	Identificador del boleto asociado.
MetodoPago	ENUM	NOT NULL	Método de pago utilizado.
FechaPago	DATETIME	DEFAULT CURRENT_TIMESTAMP	Fecha y hora del pago.
Monto	DECIMAL(10, 2)	NOT NULL	Monto del pago.
Relacionada con la tabla Boletos a través de BoletoID.			

2.5 Definir las restricciones de integridad referencial

```
56  -- Restricciones de integridad referencial
57  • ALTER TABLE Funciones
58  ADD CONSTRAINT fk_pelicula
59  FOREIGN KEY (PeliculaID) REFERENCES Peliculas(PeliculaID)
60  ON DELETE CASCADE
61  ON UPDATE CASCADE;
62
63  • ALTER TABLE Boletos
64  ADD CONSTRAINT fk_cliente
65  FOREIGN KEY (ClienteID) REFERENCES Clientes(ClienteID)
66  ON DELETE CASCADE
67  ON UPDATE CASCADE;
68
69  • ALTER TABLE Boletos
70  ADD CONSTRAINT fk_funcion
71  FOREIGN KEY (FuncionID) REFERENCES Funciones(FuncionID)
72  ON DELETE CASCADE
73  ON UPDATE CASCADE;
74
75  • ALTER TABLE Pagos
76  ADD CONSTRAINT fk_boleto
77  FOREIGN KEY (BoletoID) REFERENCES Boletos(BoletoID)
78  ON DELETE CASCADE
79  ON UPDATE CASCADE;
```

Script:

ALTER TABLE Funciones

ADD CONSTRAINT fk_pelicula

FOREIGN KEY (PeliculaID) REFERENCES Peliculas(PeliculaID)

ON DELETE CASCADE

ON UPDATE CASCADE;

ALTER TABLE Boletos

ADD CONSTRAINT fk_cliente

FOREIGN KEY (ClienteID) REFERENCES Clientes(ClienteID)

ON DELETE CASCADE

ON UPDATE CASCADE;

ALTER TABLE Boletos

ADD CONSTRAINT fk_funcion

FOREIGN KEY (FuncionID) REFERENCES Funciones(FuncionID)

ON DELETE CASCADE

ON UPDATE CASCADE;

ALTER TABLE Pagos

ADD CONSTRAINT fk_boleto

FOREIGN KEY (BoletoID) REFERENCES Boletos(BoletoID)

ON DELETE CASCADE

ON UPDATE CASCADE;

3. Seguridad, Auditoría y Control de Acceso

Objetivo: Proteger los datos sensibles y controlar el acceso a la base de datos.

Actividades:

3.1 Implementar políticas de acceso y seguridad.

Práctica: Crear roles y permisos de usuario (por ejemplo, roles de Administrador, Usuario, Auditor) para controlar el acceso a las tablas y vistas.

```
82  -- Crear roles
83  • CREATE ROLE 'AdminCine', 'Usuario', 'Auditor';
84  -- Asignar permisos al rol Administrador
85  • GRANT ALL PRIVILEGES ON cine.* TO 'AdminCine';
86  -- Asignar permisos al rol Usuario
87  • GRANT SELECT, INSERT, UPDATE ON cine.Boletos TO 'Usuario';
88  • GRANT SELECT ON cine.Funciones TO 'Usuario';
89  -- Asignar permisos al rol Auditor
90  • GRANT SELECT ON cine.* TO 'Auditor';
```

Script:

```
-- Crear roles
CREATE ROLE 'AdminCine', 'Usuario', 'Auditor';
-- Asignar permisos al rol Administrador
GRANT ALL PRIVILEGES ON cine.* TO 'AdminCine';
-- Asignar permisos al rol Usuario
GRANT SELECT, INSERT, UPDATE ON cine.Boletos TO
'Usuario';
GRANT SELECT ON cine.Funciones TO 'Usuario';
-- Asignar permisos al rol Auditor
GRANT SELECT ON cine.* TO 'Auditor';
```

Importancia del Conocimiento: El control adecuado de acceso previene fugas de información y mejora la seguridad general.

3.2 Cifrado de datos sensibles.

Práctica: Cifrar información sensible como contraseñas y detalles de pago (por ejemplo, usando AES_ENCRYPT en MySQL).

```
92 -- Cifrar contraseñas usando AES_ENCRYPT
93 • SET SQL_SAFE_UPDATES = 0; -- Desactivar Modo Seguro
94
95 • UPDATE Clientes
96 SET Contraseña = AES_ENCRYPT('contraseña_segura', 'clave_secreta');
97
98 • SET SQL_SAFE_UPDATES = 1; -- Vuelve a activarlo para evitar errores en el futuro.
```

Script

```
SET SQL_SAFE_UPDATES = 0;
```

```
UPDATE Clientes
```

```
SET Contraseña = AES_ENCRYPT('contraseña_segura',
'clave_secreta');
```

```
SET SQL_SAFE_UPDATES = 1;
```

Investigación: Explorar algoritmos de cifrado y su impacto en el rendimiento de la base de datos.

Importancia del Conocimiento: El cifrado es esencial para la protección de la información confidencial de los usuarios.

3.3 Habilitar auditoría y registrar eventos de base de datos.

Práctica: Activar los logs de acceso y auditoría para monitorear las actividades de los usuarios (por ejemplo, registrar quién accedió a qué datos).

```
100 -- Habilitar logs de auditoría
101 • SET GLOBAL log_output = 'FILE';
102 • SET GLOBAL general_log = 'ON';
```

SCRIPT:

```
-- Habilitar logs de auditoría
```

```
SET GLOBAL log_output = 'FILE';
```

```
SET GLOBAL general_log = 'ON';
```

Importancia del Conocimiento: La auditoría permite rastrear cambios en los datos y detectar actividades sospechosas.

4. RespalDOS y Recuperación de Datos

Objetivo: Asegurar la integridad y disponibilidad de los datos mediante técnicas de respaldo confiables.

Actividades:

4.1 Crear respaldos completos (full backups).

Práctica: Utilizar mysqldump o herramientas similares para hacer respaldos completos de la base de datos.

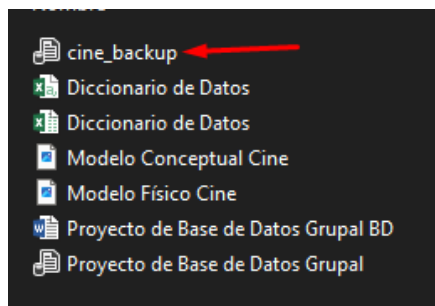
Para la realización del backup de la base de datos se puede abrir la terminal de la computadora y ponemos este comando

```
mysqldump -u [usuario] -p cine > cine_backup.sql
```

En usuario ponemos el nombre de usuario de mysql workbench después el nombre de la base de datos que deseamos hacerle el backup y ponemos el nombre de cómo queremos que se guarde el respaldo.

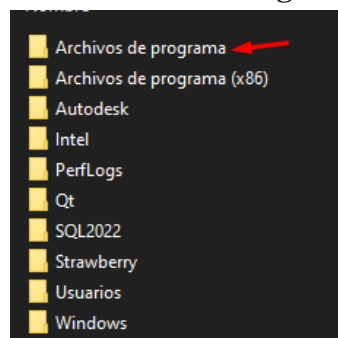
En el caso del proyecto especificamos donde queremos que se guarde el archivo de respaldo y el comando queda así

```
mysqldump -u root -p cine > "C:\Users\Usuario\Documents\3 semestre\Base de datos\Proyecto Final\PYBDD\cine_backup.sql"
```

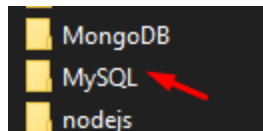


Se guardó el respaldo de la base de datos ahora si en el cmd no reconoce a mysqldump como comando se realiza lo siguiente:

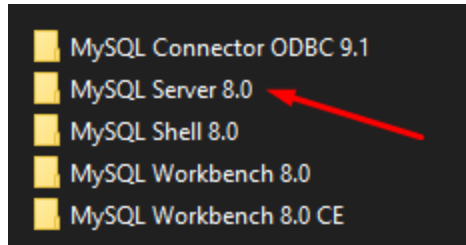
1. Localizar el archivo bin de la instalación de mysql, se encuentra la descarga de mysql en Archivos de programa



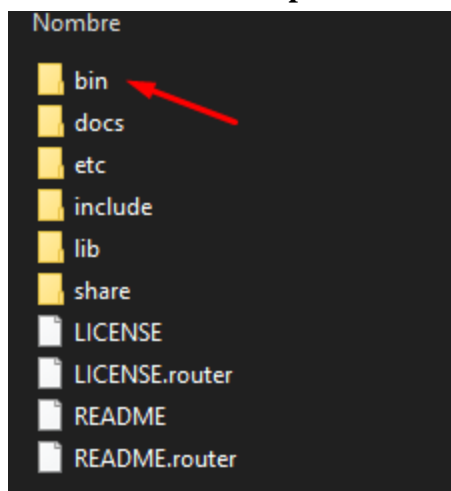
Se busca la carpeta de mysql



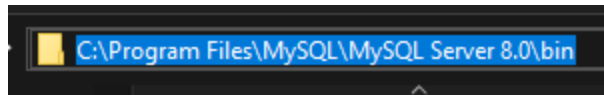
Seleccionamos MySQL Server la versión que se tenga descargada



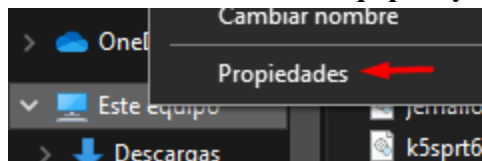
Seleccionamos la carpeta bin



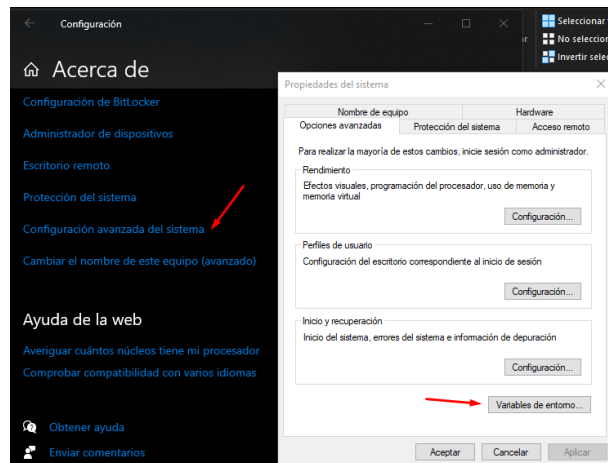
Y copiamos la ruta de la carpeta



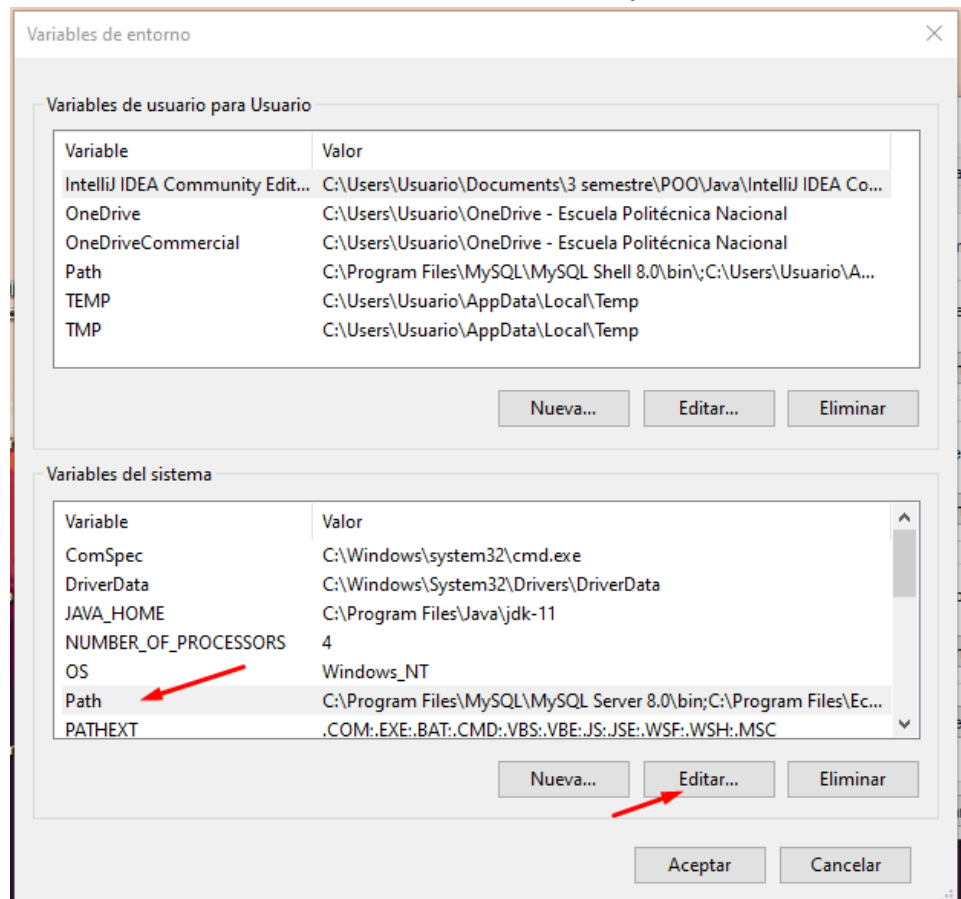
Click derecho en “Este Equipo” y seleccionamos Propiedades



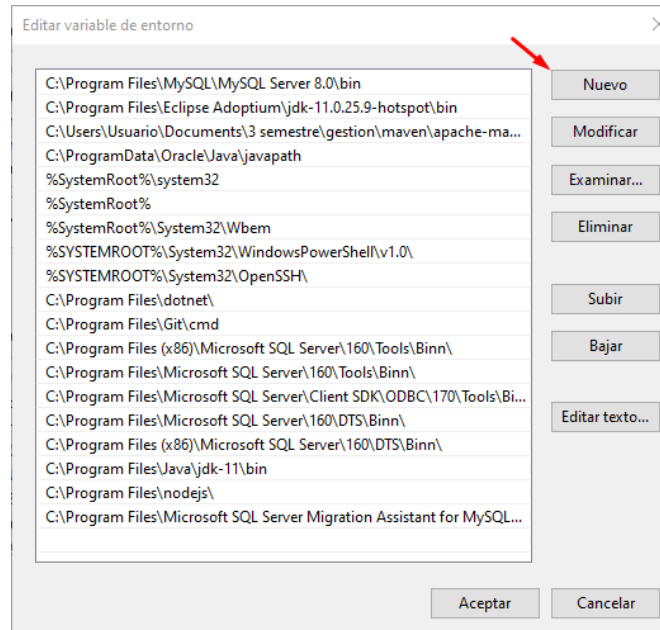
Seleccionar configuración avanzada del sistema, se abre las propiedades del sistema y seleccionamos Variables de entorno



En Variables del sistema seleccionamos Path y Editar



Seleccionamos nuevo y pegamos la dirección donde se encuentra la carpeta bin , seleccionamos subir y aceptamos en todas las ventanas que están abiertas sobre las Variables de entorno, realizado esto se ejecuta el comando para realizar el respaldo de la base de datos



Investigación: Buscar estrategias de respaldo para bases de datos de gran tamaño y la mejor manera de gestionarlas.

Las estrategias más comunes que garantizan la integridad y disponibilidad de los datos sin afectar el rendimiento del sistema son:

Respaldo Incrementales

En lugar de realizar respaldos completos cada vez, se realizan respaldos incrementales que solo contienen los cambios realizados desde el último respaldo completo o incremental. La ventaja de esto es la reducción en el tiempo de respaldo y en el espacio de almacenamiento requerido.

Para realizar estos respaldos incrementales existen herramientas y soluciones como Percona XtraBackup o MySQL Enterprise Backup que permiten respaldos incrementales de manera más eficiente

Respallos en Caliente (Hot Backups)

Permiten respaldar la base de datos mientras está en uso y sin interrumpir el servicio. Esto es ideal para sistemas que requieren alta disponibilidad. Las herramientas que se pueden utilizar son Percona XtraBackup o el propio MySQL Enterprise Backup, que permiten realizar copias de seguridad sin detener el servicio.

4.2 Configurar respaldos incrementales.

Práctica: Realizar respaldos incrementales para reducir el tiempo y espacio de almacenamiento.

Para la realización de un respaldo incremental, añadimos una tabla en la base de datos llamada RegistroCambios

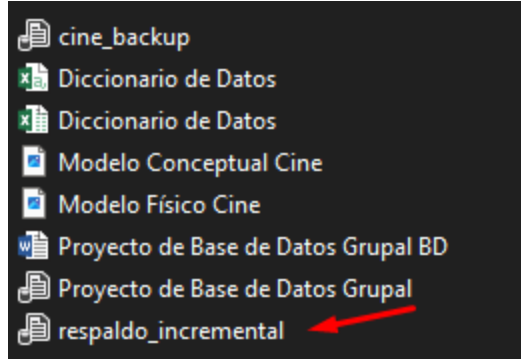
```
> CREATE TABLE RegistroCambios (  
    CambioID INT AUTO_INCREMENT PRIMARY KEY,  
    TablaAfectada VARCHAR(50) NOT NULL,  
    IDRegistro INT NOT NULL,  
    TipoCambio ENUM('INSERT', 'UPDATE', 'DELETE') NOT NULL,  
    FechaCambio TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
);
```

Creamos Trigger los cuales guarden la información en la tabla RegistroCambios cuando se realice algo en cada tabla de la base de datos

Ahora para guardar el respaldo incremental ejecutamos este comando

```
mysqldump -u root -p --where="FechaCambio >= '2024-02-01'" cine  
RegistroCambios > "C:/Users/Usuario/Documents/3 semestre/Base  
de datos/Proyecto Final/PYBDD/respaldo_incremental.sql"
```

Especificamos la fecha que queremos hacer el respaldo incremental y la ruta en la que queremos que se guarde



Investigación: Investigar cómo realizar respaldos incrementales y cuándo es más conveniente utilizarlos.

Un respaldo incremental es una técnica de copia de seguridad en la que solo se respaldan los archivos que han cambiado desde el último respaldo, ya sea completo o incremental.

¿Cómo funciona un respaldo incremental?

Respaldo completo: Se realiza un primer respaldo completo, que incluye todos los archivos y datos.

Respaldo incremental: Posteriormente, se realizan respaldos incrementales, que solo incluyen los archivos que han cambiado desde el último respaldo.

¿Cuándo es más conveniente utilizar respaldos incrementales?

Frecuencia de cambios en la base de datos: Si se tiene una base de datos donde se realizan muchas actualizaciones o inserciones de datos, los respaldos incrementales son una buena opción, ya que solo respaldarás los cambios más recientes.

Limitación de espacio de almacenamiento: Si el espacio de almacenamiento es limitado y no se puede hacer respaldos completos frecuentes, los respaldos incrementales permiten hacer copias de seguridad sin ocupar tanto espacio.

Bases de datos grandes: Para bases de datos muy grandes, los respaldos completos pueden ser poco prácticos, y los incrementales ofrecen una alternativa más manejable.

Ambientes con alta disponibilidad: En entornos donde el sistema debe estar siempre disponible (por ejemplo, sitios web o aplicaciones de misión crítica), los respaldos incrementales permiten realizar copias de seguridad sin interrumpir el servicio.

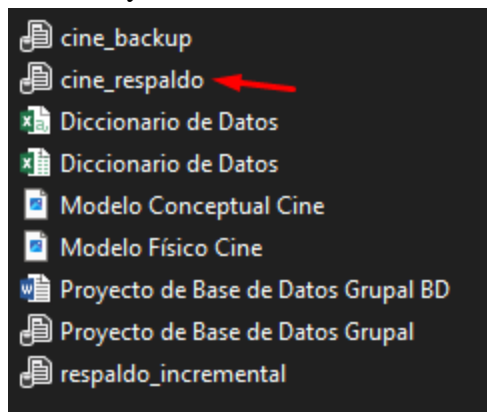
Importancia del Conocimiento: Los respaldos incrementales permiten optimizar los recursos y acelerar los tiempos de recuperación.

4.3 Implementar respaldos en caliente (Hot Backups).

Práctica: Hacer respaldos sin interrumpir el servicio (por ejemplo, usando Percona XtraBackup).

Para realizar el respaldo en caliente ejecutamos el siguiente comando

```
mysqldump --single-transaction --routines --triggers --databases cine  
-u root -p> "C:/Users/Usuario/Documents/3 semestre/Base de  
datos/Proyecto Final/PYBDD/cine_respaldo.sql"
```



Investigación: Investigar cómo hacer respaldos sin detener la base de datos.

Hacer un respaldo sin detener la base de datos (respaldos en caliente) es fundamental para entornos de producción donde la disponibilidad debe mantenerse sin interrupciones.

Uso de mysqldump con opción --single-transaction

mysqldump es una de las herramientas más comunes para realizar respaldos en MySQL. Cuando se utiliza correctamente con la opción --single-transaction, puedes realizar un respaldo coherente sin bloquear las tablas. Esto es ideal para bases de datos con motores de almacenamiento que admiten transacciones, como InnoDB.

Las ventajas de hacer esto es que no interrumpe la operación de la base de datos y permite realizar respaldos consistentes sin necesidad de bloquear tablas.

Percona XtraBackup (Respaldo en caliente)

Es una herramienta especializada para hacer respaldos físicos de bases de datos MySQL y MariaDB sin interrumpir el servicio, lo que se conoce como respaldo en caliente.

Las ventajas es Respaldo físico que no bloquea la base de datos y es adecuado para bases de datos grandes, ya que el respaldo no se ve afectado por la carga en el servidor.

Respaldo Lógico con mydumper

mydumper es una herramienta alternativa a mysqldump que realiza respaldos de bases de datos MySQL y MariaDB de forma paralela y eficiente. Está diseñada para realizar respaldos en caliente de manera rápida y no bloqueante.

Ventajas:

Mucho más rápido que mysqldump.

Permite respaldos en caliente sin bloquear la base de datos.

Importancia del Conocimiento: Los respaldos en caliente son esenciales para bases de datos de producción que no pueden permitirse inactividad.

5. Optimización y Rendimiento de Consultas

Objetivo: Mejorar la eficiencia en la recuperación de datos mediante la optimización de consultas y el uso adecuado de índices.

Actividades:

5.1 Crear y gestionar índices.

Práctica: Implementar índices en las columnas más consultadas, como VueloID, ClienteID, etc.

Crear dos índices importantes

En el caso de este proyecto, se deben implementar índices en las columnas más consultadas para mejorar el rendimiento y la eficiencia de las consultas. A continuación, se detallan los índices que se deben crear:

Índice en la columna ClienteID de la tabla Boletos

Índice en la columna PeliculaID de la tabla Funciones

```
-- Crear índice en la columna ClienteID de la tabla Boletos
CREATE INDEX idx_clienteid ON Boletos(ClienteID);
```

```
-- Crear índice en la columna PeliculaID de la tabla Funciones
CREATE INDEX idx_peliculaid ON Funciones(PeliculaID);
```

Investigación: Investigar sobre los tipos de índices más adecuados para bases de datos transaccionales y cómo afectan el rendimiento.

Las bases de datos transaccionales suelen tener un alto volumen de operaciones de lectura y escritura, por lo que la elección de índices adecuados es crucial para optimizar tanto el rendimiento de las consultas como la eficiencia en las operaciones de actualización. Los índices correctos pueden acelerar significativamente las consultas y, al mismo tiempo, reducir la carga en las escrituras.

Índice B-Árbol (Árbol-B):

Es el tipo de índice más común y el predeterminado en la mayoría de los sistemas de bases de datos, como MySQL y PostgreSQL. Un índice B-Tree organiza los datos en una estructura jerárquica de árbol balanceado, lo que permite realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico ($O(\log n)$).

Índice Hash:

Utilice una función de hash para asignar claves a ubicaciones específicas en una tabla de índice. Los índices Hash son ideales para búsquedas exactas, es decir, cuando se busca un valor específico en una columna. Es más adecuado para consultas que utilizan la cláusula = en lugar de rangos (por ejemplo, búsquedas por ID). No es adecuado para rangos de valores o consultas que involucren un orden de los datos.

Índice de Texto Completo (Índice de texto completo):

Este tipo de índice está diseñado para facilitar las búsquedas en grandes volúmenes de texto, como las columnas VARCHAR o TEXT. Utiliza algoritmos de búsqueda de texto que permiten consultas más complejas como MATCH y CONTRA.

Índice compuesto:

Un índice compuesto utiliza múltiples columnas para formar un único índice. Esto es útil cuando las consultas implican más de una columna en las condiciones de búsqueda.

Se utiliza cuando las consultas más comunes incluyen condiciones que involucren varias columnas (por ejemplo, búsquedas por ClienteID y Fecha).

Importancia del Conocimiento: Los índices son cruciales para acelerar las consultas y mejorar el rendimiento general de la base de datos.

5.2 Optimizar consultas SQL.

Práctica: Utilizar herramientas como EXPLAIN para identificar cuellos de botella en las consultas y optimizarlas.

La herramienta EXPLAIN permite analizar el plan de ejecución de una consulta SQL. Proporciona detalles sobre cómo la base de datos ejecutará una consulta, los índices que usará y la cantidad estimada de registros procesados en cada paso. Utilizar EXPLAIN ayuda a identificar cuellos de botella y áreas de mejora en una consulta.

Ejemplo

```
219 • EXPLAIN
220 SELECT Clientes.Nombre, Funciones.PeliculaID, Boletos.AsientosReservados
221 FROM Clientes
222 JOIN Boletos ON Clientes.ClienteID = Boletos.ClienteID
223 JOIN Funciones ON Boletos.FuncionID = Funciones.FuncionID
224 WHERE Boletos.PrecioTotal > 100.00;
225
```

	id	select_type	table	partitions	type	possible_keys	key	
▶	1	SIMPLE	Clientes	NULL	ALL	PRIMARY	NULL	1
	1	SIMPLE	Boletos	NULL	ref	fk_funcion,idx_clienteid	idx_clienteid	5
	1	SIMPLE	Funciones	NULL	eq_ref	PRIMARY	PRIMARY	4

Practica: Aplicación de 3 join

```
227 • EXPLAIN
228 SELECT Clientes.Nombre, Funciones.PeliculaID, Peliculas.Titulo, Boletos.
229 FROM Clientes
230 JOIN Boletos ON Clientes.ClienteID = Boletos.ClienteID
231 JOIN Funciones ON Boletos.FuncionID = Funciones.FuncionID
232 JOIN Peliculas ON Funciones.PeliculaID = Peliculas.PeliculaID
233 WHERE Boletos.PrecioTotal > 100.00;
234
```

	id	select_type	table	partitions	type	possible_keys	key	
▶	1	SIMPLE	Clientes	NULL	ALL	PRIMARY	NULL	1
	1	SIMPLE	Boletos	NULL	ref	fk_funcion,idx_clienteid	idx_clienteid	5
	1	SIMPLE	Funciones	NULL	eq_ref	PRIMARY,idx_peliculaid	PRIMARY	4
	1	SIMPLE	Peliculas	NULL	eq_ref	PRIMARY	PRIMARY	4

Investigación: Investigar cómo hacer uso eficiente de las uniones (JOIN), subconsultas, y optimizar las consultas complejas.

Cuando trabajamos con bases de datos complejas, las uniones (JOIN), las subconsultas y la optimización de consultas juegan un papel crucial en el rendimiento y la eficiencia.

Uso Eficiente de JOIN

El uso de JOIN permite combinar varias tablas basadas en relaciones entre ellas. Sin embargo, es fundamental utilizar JOIN correctamente para evitar impactos negativos en el rendimiento.

Consejos para optimizar el uso de JOIN

Índices adecuados: Se debe de asegurar de tener índices en las columnas que se usan para unir las tablas. Esto mejora el rendimiento de las uniones, especialmente en tablas grandes.

Limitar las columnas: Selecciona solo las columnas necesarias. No utilices SELECT * cuando solo necesitas algunas columnas.

Filtrado antes de unir: Siempre que sea posible, se debe filtrar los datos antes de hacer el JOIN. Esto reduce la cantidad de datos que se deben unir.

Evitar uniones innecesarias: Se debe de asegurar de que las uniones que realizas sean necesarias para obtener los resultados deseados.

Uniones innecesarias pueden reducir el rendimiento.

Uso de Subconsultas

Las subconsultas son consultas anidadas dentro de otra consulta. Son útiles para obtener resultados intermedios o para calcular valores que no son directamente accesibles.

Tipos de subconsultas:

Subconsulta escalar: Devuelve un solo valor y se usa en una cláusula como SELECT, WHERE o HAVING.

Subconsulta de fila: Devuelve una sola fila de múltiples columnas.

Subconsulta de tabla: Devuelve múltiples filas y columnas, y generalmente se utiliza en una cláusula FROM.

Consejos para optimizar subconsultas:

Evitar subconsultas en SELECT si no son necesarias: Si la subconsulta está en la cláusula SELECT y no se necesita para cada fila, considera reescribir la consulta utilizando JOIN.

Usar EXISTS en lugar de IN: En casos donde una subconsulta devuelve un conjunto grande de resultados, EXISTS puede ser más eficiente que IN, ya que EXISTS termina cuando encuentra el primer valor, mientras que IN debe evaluar toda la subconsulta.

Usar subconsultas en la cláusula WHERE: Si se necesita un valor calculado para filtrar las filas, las subconsultas en WHERE son

eficaces. Sin embargo, deben evitarse si la subconsulta devuelve muchos registros innecesarios.

Optimización de Consultas Complejas

Las consultas complejas, especialmente aquellas que involucran varias uniones y subconsultas, pueden ser lentas si no están bien optimizadas.

Estrategias de optimización:

Uso de índices: Se debe de asegurar de que las columnas utilizadas para unir tablas (en las cláusulas ON y WHERE) tengan índices.

Limitar las filas procesadas: Se debe de usar LIMIT para evitar procesar demasiadas filas si solo necesitas una muestra o los primeros resultados.

Evitar SELECT *: Seleccionar solo las columnas necesarias en lugar de usar SELECT *. Esto reduce la cantidad de datos que el motor de la base de datos debe procesar.

Importancia del Conocimiento: Las consultas optimizadas aseguran un sistema rápido y eficiente, especialmente en sistemas con alta demanda.

5.3 Utilizar particionamiento de tablas.

Práctica: Dividir tablas grandes, como Reservas, en particiones según una clave (por ejemplo, por fecha).

En el caso de nuestra base de datos se va a realizar la partición de la tabla funciones

```
-- Particionamiento de la tabla
-- 1 Eliminar restricciones de claves foráneas
ALTER TABLE Funciones DROP FOREIGN KEY fk_pelicula;
-- 2 Crear el índice único
CREATE UNIQUE INDEX idx_fechaHora ON Funciones (FechaHora);
-- 3 Particionar la tabla Funciones
ALTER TABLE Funciones
PARTITION BY RANGE (YEAR(FechaHora)) (
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025)
);
```



```
-- 4 Restaurar la clave foránea
ALTER TABLE Funciones
ADD CONSTRAINT fk_película FOREIGN KEY (PelículaID)
REFERENCES Peliculas(PelículaID) ON DELETE CASCADE;
-- 5 Verificar la partición
SELECT *
FROM information_schema.partitions
WHERE table_name = 'Funciones';
-- Consultar las filas de una partición específica
SELECT *
FROM Funciones
WHERE YEAR(FechaHora) = 2023;
```

Result Grid

Filter Rows:

Edit:

	FuncionID	PelículaID	Sala	FechaHora	AsientosDisponibles
*	NULL	NULL	NULL	NULL	NULL

Investigación: Investigar sobre los beneficios del particionamiento y cómo implementarlo en sistemas de bases de datos grandes.

El particionamiento de bases de datos es una técnica que implica dividir una tabla grande en varias tablas más pequeñas (particiones), pero a efectos de consultas y administración, se sigue tratando como una sola. Este enfoque puede mejorar el rendimiento y la eficiencia en la administración de bases de datos grandes.

Beneficios

Mejora del rendimiento de las consultas

Al particionar una tabla, las consultas pueden ser más eficientes porque solo se necesita buscar en una partición específica en lugar de escanear toda la tabla.

Mantenimiento más fácil

Se puede hacer respaldos de particiones específicas en lugar de hacer un respaldo completo de una tabla grande, lo que puede reducir el tiempo de ejecución de los backups y aumentar la eficiencia de la recuperación de datos.

Escalabilidad mejorada

A medida que los datos aumentan, el particionamiento puede ayudar a mantener el rendimiento de las consultas de manera constante, dividiendo el trabajo entre varias particiones. También puede ser más sencillo distribuir las particiones en diferentes servidores o en diferentes ubicaciones geográficas.

Implementación del particionamiento

Implementar particionamiento en bases de datos grandes requiere una planificación cuidadosa. Existen varios enfoques para particionar, dependiendo de las necesidades y características de los datos.

Tipos de particionamiento

Por rango (Range Partitioning): Es uno de los tipos más comunes. Se divide una tabla en particiones basadas en un rango de valores, por ejemplo, fechas o valores numéricos. Es ideal para tablas donde los datos siguen un patrón cronológico, como registros de ventas por año o por mes.

Por lista (List Partitioning): Se divide la tabla en particiones según un conjunto de valores discretos, como categorías o grupos específicos. Es útil cuando los datos pertenecen a categorías predefinidas.

Por hash (Hash Partitioning): La tabla se divide en particiones usando una función hash en una o más columnas. Esto distribuye los datos de manera uniforme entre las particiones. Es útil cuando no existe un patrón claro de distribución para los datos.

Consideraciones para la implementación

No todas las bases de datos soportan particionamiento o tienen limitaciones en las características.

Si bien el particionamiento mejora el rendimiento de muchas consultas, algunas consultas pueden volverse más lentas si no se tienen en cuenta las particiones adecuadas. Siempre es importante probar y ajustar las consultas para asegurarse de que se aproveche correctamente el particionamiento.

Los índices juegan un papel crucial en las tablas particionadas. Se debe asegurar de tener índices apropiados para optimizar el rendimiento de las consultas en las particiones.

Si ya se tiene una tabla con muchos registros y se desea particionarla, primero se debe pensar en cómo migrar los datos a las nuevas particiones sin interrumpir el servicio.

Importancia del Conocimiento: El particionamiento de tablas mejora la escalabilidad y el rendimiento en bases de datos con gran volumen de datos.

6. Procedimientos Almacenados, Vistas y Triggers, Funciones (prácticas de cada uno)

6.1. Crear procedimientos almacenados.

Práctica: Crear un procedimiento para calcular el precio total de una reserva, aplicando descuentos y cargos adicionales, aplicar 2 ejercicios y explicar comprensión al 100%:

El siguiente procedimiento calcula el precio total de una reserva de boletos para el cine, considerando los siguientes aspectos: el precio base del boleto, los descuentos aplicables y los cargos adicionales, si los hubiera. Este procedimiento debe realizarse cada vez que un cliente realice una compra, y puede involucrar distintas situaciones, como descuentos por promociones o cargos extra por asientos premium, etc.

Pasos:

Obtener el precio del boleto: El precio base del boleto se obtiene a partir del número de asientos reservados y el precio estándar por asiento.

Aplicar descuentos: Dependiendo de la situación del cliente (por ejemplo, si tiene una membresía o si hay una promoción vigente), se puede aplicar un descuento sobre el precio base. Los descuentos se pueden expresar como un porcentaje o también como un valor fijo.

Aplicar cargos adicionales: Si el cliente opta por servicios adicionales como asientos VIP o premium, se añaden cargos adicionales al precio total.

Calcular el precio total: El precio total será la suma del precio base, los descuentos y los cargos adicionales. Este es el valor final que se utilizará para completar la reserva y generar el boleto.

```
252 CREATE PROCEDURE calcularPrecioReserva(  
253     IN p_clienteID INT,  
254     IN p_funcionID INT,  
255     IN p_asientosReservados INT,  
256     OUT p_precioTotal DECIMAL(10, 2)  
257 )  
258 BEGIN  
259     DECLARE precioBase DECIMAL(10, 2);  
260     DECLARE descuento DECIMAL(10, 2) DEFAULT 0;  
261     DECLARE cargoAdicional DECIMAL(10, 2) DEFAULT 0;  
262  
263     -- Obtenemos el precio base de la función  
264     SELECT 10.00 * (p_asientosReservados * 2.00) INTO precioBase  
265     FROM Funciones  
266     WHERE FuncionID = p_funcionID;  
267  
268     -- Aplicar descuento si el cliente tiene un descuento  
269     -- Así, si el cliente es frecuente (por ejemplo, con un clienteID par), se aplicará un 50% de descuento.  
270     IF MOD(p_clienteID, 2) = 0 THEN  
271         SET descuento = precioBase * 0.50;  
272     END IF;  
273  
274     -- Aplicar cargos adicionales si el cliente opta por asientos VIP  
275     -- Por ejemplo, si el número de asientos reservados es mayor a 3, se aplican cargos adicionales.  
276     IF p_asientosReservados > 3 THEN  
277         SET cargoAdicional = 5.00;  
278     END IF;  
279  
280     -- Calculamos el precio total final  
281     SET p_precioTotal = (precioBase - descuento) + cargoAdicional;  
282  
283 END
```

Ejercicio 1: Calcular el precio total para una reserva de 3 asientos con un descuento y un cargo adicional.

Contexto:

Un cliente reserva 3 asientos para una película. El precio base del boleto por asiento es de \$10. Además, el cliente tiene un 10% de descuento por ser miembro y ha optado por asientos premium con un cargo adicional de \$5 por cada asiento. Calculamos el precio total de esta reserva:

```
297 ● -- Ejercicio 1 --
298 # Cliente frecuente (ClienteID=2) reserva 3 asientos:
299 CALL CalcularPrecioReserva(1, 2, 3, @precioFinal);
300 ● SELECT @precioFinal;
```

Result Grid

	@precioFinal
▶	41.00

Ejercicio 2: Calcular el precio total para una reserva sin descuento y sin cargo adicional.

Contexto:

Ahora supongamos que un cliente ha reservado 5 asientos para una película sin aplicar ningún descuento y sin optar por asientos premium. El precio base del boleto es de \$12 por asiento.

```
302 -- Ejercicio 2 --
303 -- Calcular el precio total para una reserva sin descuento y sin cargo adicional:
304 ● CALL CalcularPrecioReserva(2, 3, 5, @precioFinal);
305 ● SELECT @precioFinal;
```

Result Grid

	@precioFinal
▶	61.00

Investigación: Explorar cómo los procedimientos almacenados pueden mejorar la reutilización de código y la eficiencia.

Los procedimientos almacenados son bloques de código SQL que se almacenan y ejecutan directamente en el servidor de bases de datos. Su uso ofrece múltiples ventajas en términos de eficiencia, seguridad y mantenimiento del sistema. A continuación, se detallan algunas de las principales ventajas:

1. Reutilización de código: Al encapsular la lógica de negocio en procedimientos almacenados, se facilita su reutilización en diferentes partes de una aplicación o en diversas tareas de administración de la base de datos. Esto evita la duplicación de código y simplifica el mantenimiento, ya que cualquier cambio en la lógica se realiza en un único lugar, garantizando consistencia en toda la aplicación.

2. Mejora del rendimiento: Los procedimientos almacenados se ejecutan en el servidor de bases de datos, lo que reduce la necesidad de enviar múltiples consultas desde la aplicación cliente. Esto disminuye el tráfico de red y permite que las operaciones se realicen de manera más eficiente. Además, al estar precompilados, su ejecución es más rápida en comparación con consultas ad hoc enviadas desde el cliente.

3. Seguridad mejorada: Al utilizar procedimientos almacenados, se puede restringir el acceso directo a las tablas, permitiendo que los usuarios interactúen con los datos únicamente a través de estos procedimientos. Esto añade una capa adicional de seguridad, ya que se pueden implementar controles y validaciones dentro de los procedimientos para garantizar la integridad y confidencialidad de los datos.

4. Mantenimiento simplificado: Centralizar la lógica de negocio en procedimientos almacenados facilita el mantenimiento del sistema. Cualquier modificación o actualización en la lógica se realiza en el procedimiento correspondiente, sin necesidad de cambiar múltiples aplicaciones o interfaces que dependan de esa lógica. Esto reduce el riesgo de errores y asegura una implementación coherente de las reglas de negocio.

5. Reducción del tráfico de red: Al ejecutar múltiples instrucciones SQL en una sola llamada al procedimiento almacenado, se minimiza la comunicación entre el cliente y el servidor. Esto disminuye el tráfico de red y mejora el rendimiento de la aplicación, especialmente en entornos donde la latencia de red es un factor crítico.

Importancia del Conocimiento: Los procedimientos almacenados centralizan la lógica y pueden mejorar el rendimiento al ejecutarse directamente en el servidor.

6.2. Crear vistas para simplificar consultas complejas.

Práctica: Crear vistas que presenten información de varias tablas de manera unificada (por ejemplo, una vista que combine datos de Vuelos, Clientes y Reservas).

Vista películas funciones: Esta vista muestra una lista de las películas disponibles junto con sus funciones, mostrando detalles como el título de la película, su género, duración, clasificación, fecha de estreno, y las funciones programadas con su respectiva sala y disponibilidad de asientos.

```

308 -- 5.2 Crear vistas para simplificar consultas complejas.
309
310 -- Vista para obtener la lista de películas disponibles con sus funciones
311 ● CREATE VIEW Vista_Peliculas_Funciones AS
312 SELECT
313     p.PeliculaID,
314     p.Titulo,
315     p.Genero,
316     p.Duracion,
317     p.Clasificacion,
318     p.FechaEstreno,
319     f.FuncionID,
320     f.Sala,
321     f.FechaHora,
322     f.AsientosDisponibles
323 FROM Peliculas p
324 JOIN Funciones f ON p.PeliculaID = f.PeliculaID;
325
326 ● SELECT * FROM Vista_Peliculas_Funciones LIMIT 10;
327
328 -- Vista para obtener los boletos comprados por cada cliente
329 ● CREATE VIEW Vista_Boletos_Clientes AS
330 SELECT

```

PeliculaID	Titulo	Genero	Duracion	Clasificacion	FechaEstreno	FuncionID	Sala	FechaHora	AsientosDisponibles
1	Mi Pelicula	Acción	120	A	2025-02-02	1	Sala 1	2025-02-02 18:00:00	50
2	Avengers: Endgame	Acción	181	Apta para mayores de 13	2019-04-26	2	Sala 2	2025-02-02 20:00:00	40

Vista boletos clientes: Esta vista combina la información de los boletos adquiridos por los clientes, incluyendo el nombre del cliente, detalles del boleto, la película asociada, la función, la sala, los asientos reservados, y el precio total pagado por cada cliente.

```

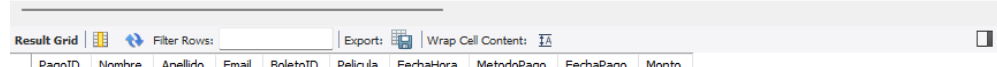
328 -- Vista para obtener los boletos comprados por cada cliente
329 ● CREATE VIEW Vista_Boletos_Clientes AS
330 SELECT
331     c.ClienteID,
332     c.Nombre,
333     c.Apellido,
334     c.Email,
335     b.BoletoID,
336     f.FuncionID,
337     p.Titulo AS Pelicula,
338     f.Sala,
339     f.FechaHora,
340     b.AsientosReservados,
341     b.PrecioTotal
342 FROM Clientes c
343 JOIN Boletos b ON c.ClienteID = b.ClienteID
344 JOIN Funciones f ON b.FuncionID = f.FuncionID
345 JOIN Peliculas p ON f.PeliculaID = p.PeliculaID;
346
347 ● SELECT * FROM Vista_Boletos_Clientes LIMIT 10;
348

```

ClienteID	Nombre	Apellido	Email	BoletoID	FuncionID	Pelicula	Sala	FechaHora	AsientosReservados
1	Sebastian	Betancourt	sebas.betan@gmail.com	4	1	Mi Pelicula	Sala 1	2025-02-02 18:00:00	2
2	Justin	Perez	jus.perez@gmail.com	5	2	Avengers: Endgame	Sala 2	2025-02-02 20:00:00	3

Vista historial pagos: Esta vista muestra el historial de pagos realizados por los clientes, incluyendo el nombre del cliente, la película, la función, el método de pago utilizado, la fecha de pago, y el monto pagado por cada boleto comprado.

```
350 -- Vista para obtener el historial de pagos realizados por los clientes
351 • CREATE VIEW Vista_Historial_Pagos AS
352 SELECT
353     p.PagoID,
354     c.Nombre,
355     c.Apellido,
356     c.Email,
357     b.BoletoID,
358     pel.Titulo AS Pelicula,
359     f.FechaHora,
360     p.MetodoPago,
361     p.FechaPago,
362     p.Monto
363 FROM Pagos p
364 JOIN Boleto b ON p.BoletoID = b.BoletoID
365 JOIN Clientes c ON b.ClienteID = c.ClienteID
366 JOIN Funciones f ON b.FuncionID = f.FuncionID
367 JOIN Peliculas pel ON f.PeliculaID = pel.PeliculaID;
368
369 • SELECT * FROM Vista_Historial_Pagos LIMIT 10;
```



The screenshot shows a database management tool interface. At the top, there's a SQL editor with a query to create a view named 'Vista_Historial_Pagos' and a query to select all data from it, limited to 10 rows. Below the editor, there's a 'Result Grid' section. The grid has columns for 'PagoID', 'Nombre', 'Apellido', 'Email', 'BoletoID', 'Pelicula', 'FechaHora', 'MetodoPago', 'FechaPago', and 'Monto'. The grid is currently empty, showing only the column headers.

Investigación: Investigar las ventajas de usar vistas en lugar de consultas complejas repetitivas.

Las vistas en bases de datos son estructuras que permiten simplificar y optimizar el manejo de consultas complejas. A continuación, se detallan las ventajas de utilizar vistas en lugar de repetir consultas complejas:

1. Simplificación de consultas complejas: Las vistas permiten encapsular consultas complejas, facilitando su reutilización sin necesidad de reescribirlas. Esto no solo ahorra tiempo, sino que también reduce la posibilidad de errores al evitar la duplicación de código. Al utilizar vistas, los desarrolladores pueden referirse a ellas como si fueran tablas, lo que simplifica la sintaxis de las consultas y mejora la legibilidad del código.

2. Mejora de la seguridad: Las vistas pueden restringir el acceso a datos sensibles al mostrar solo la información necesaria a los usuarios. Al definir una vista que excluye columnas confidenciales, se asegura que los usuarios solo puedan acceder a los datos permitidos, añadiendo una capa adicional de seguridad a la base de datos.

3. Mantenimiento y consistencia: Al centralizar consultas complejas en vistas, cualquier modificación en la lógica de la consulta se realiza en un solo lugar. Esto asegura que todas las partes de la aplicación que dependen de esa lógica reflejen automáticamente los cambios, garantizando la consistencia y facilitando el mantenimiento del sistema.

4. Mejora del rendimiento: Aunque las vistas no almacenan datos por sí mismas, algunas bases de datos permiten la creación de vistas materializadas que almacenan

los resultados de la consulta. Esto puede mejorar significativamente el rendimiento al reducir el tiempo de ejecución de consultas complejas, ya que se accede directamente a los datos precomputados.

5. Abstracción de datos: Las vistas proporcionan una capa de abstracción que oculta la complejidad de las tablas subyacentes. Esto permite a los usuarios interactuar con datos de manera más intuitiva, sin preocuparse por las relaciones y estructuras internas de la base de datos.

6.3. Implementar triggers para auditoría y control de cambios.

Práctica: Crear triggers que registren cambios en las tablas de Reservas y Pagos cada vez que un registro se actualiza o elimina., 2 ejercicios conocimiento al 100%

```
377 -- 5.3 Crear triggers que registren cambios en las tablas de Reservas y Pagos:
378 -- Crear la tabla para registrar cambios en Boletos
379
380 ● ○ CREATE TABLE Registro_Boletos (
381     id SERIAL PRIMARY KEY,
382     id_boleto INT,
383     accion VARCHAR(10),
384     fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
385     usuario VARCHAR(50)
386 );
387
388 -- Trigger para registrar actualizaciones en Boletos
389 ● CREATE TRIGGER trigger_actualizar_boleto
390 AFTER UPDATE ON Boletos
391 FOR EACH ROW
392 INSERT INTO Registro_Boletos (id_boleto, accion, usuario)
393 VALUES (OLD.BoletoID, 'UPDATE', CURRENT_USER);
394 ● DESC Boletos;
395
396 -- Trigger para registrar eliminaciones en Boletos
397 ● CREATE TRIGGER trigger_eliminar_boleto
398 AFTER DELETE ON Boletos
399 FOR EACH ROW
400 INSERT INTO Registro_Boletos (id_boleto, accion, usuario)
401 VALUES (OLD.BoletoID, 'DELETE', CURRENT_USER);
402
403 -- Crear la tabla para registrar cambios en Pagos
404 ● ○ CREATE TABLE Registro_Pagos (
405     id SERIAL PRIMARY KEY,
406     id_pago INT,
407     accion VARCHAR(10),
408     fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
409     usuario VARCHAR(50)
410 );
411
412 ● DESC Pagos;
```



```

414      -- Trigger para registrar actualizaciones en Pagos
415 ●    CREATE TRIGGER trigger_actualizar_pago
416      AFTER UPDATE ON Pagos
417      FOR EACH ROW
418      INSERT INTO Registro_Pagos (id_pago, accion, usuario)
419      VALUES (OLD.PagoID, 'UPDATE', CURRENT_USER);

421      -- Trigger para registrar eliminaciones en Pagos
422 ●    CREATE TRIGGER trigger_eliminar_pago
423      AFTER DELETE ON Pagos
424      FOR EACH ROW
425      INSERT INTO Registro_Pagos (id_pago, accion, usuario)
426      VALUES (OLD.PagoID, 'DELETE', CURRENT_USER);

```

Investigación: Investigar cómo utilizar triggers para mantener un historial de cambios en la base de datos.

Los triggers, o disparadores, son procedimientos almacenados en bases de datos que se ejecutan automáticamente en respuesta a eventos específicos, como inserciones, actualizaciones o eliminaciones de registros. Su uso es fundamental para mantener un historial detallado de los cambios en una base de datos, lo que facilita la auditoría y el seguimiento de modificaciones. Al implementar triggers para auditar cambios, se puede registrar información clave como el tipo de operación realizada, el usuario que efectuó el cambio, la fecha y hora del evento, y los valores anteriores y nuevos de los datos modificados. Esta práctica es esencial para garantizar la integridad y seguridad de la información almacenada.

Para implementar un sistema de auditoría utilizando triggers, es necesario crear una tabla de auditoría que almacene los registros de cambios. Esta tabla debe contener columnas que reflejen los datos mencionados anteriormente. Posteriormente, se definen los triggers asociados a las tablas que se desean auditar. Estos triggers se configuran para activarse antes o después de las operaciones de inserción, actualización o eliminación, según los requerimientos específicos. Por ejemplo, un trigger puede insertarse en la tabla de auditoría cada vez que se actualiza un registro en la tabla principal, registrando los valores anteriores y nuevos de los campos modificados.

Es importante considerar que, aunque los triggers son herramientas poderosas para la auditoría, su uso indebido puede afectar el rendimiento de la base de datos. Por ello, se recomienda diseñar triggers eficientes y específicos, evitando operaciones complejas dentro de ellos. Además, es esencial gestionar adecuadamente el tamaño de la tabla de auditoría, implementando políticas de retención de datos y realizando mantenimientos periódicos para asegurar un rendimiento óptimo. Al seguir estas prácticas, se puede lograr una auditoría efectiva y sostenible de los cambios en la base de datos.

Importancia del Conocimiento: Los triggers permiten automatizar tareas como la auditoría y validación de datos.

7. Monitorear el rendimiento de consultas.

Práctica: Usar herramientas como SHOW PROCESSLIST para detectar consultas lentas y optimizarlas.

Esta instrucción muestra las consultas actualmente en ejecución en el servidor de base de datos, lo que permite monitorear qué consultas están consumiendo recursos y tiempo:

```
433 -- 6. Monitoreo y Optimización de Recursos
434 -- Usar herramientas como SHOW PROCESSLIST para detectar consultas lentas y optimizarlas.
435 • SHOW PROCESSLIST;
436
```

Con el primer código verificaremos si una consulta tarda más que este tiempo, y si es así se registrará en el log de consultas lentas.

En el siguiente código lo vamos a usar para mostrar el número total de consultas lentas que han ocurrido desde que se inició el servidor.

```
438 • SHOW VARIABLES LIKE 'long_query_time'; -- umbral para consultas lentas
439 • SHOW GLOBAL STATUS LIKE 'Slow_queries'; -- consultas lentas
```

Habilitamos el log de consultas lentas y establecemos un umbral de 2 segundos, lo que significa que cualquier consulta que tarde más de 2 segundos será registrada en el log de consultas lentas.

```
441 -- Activamos el log de consultas lentas
442 • SET GLOBAL slow_query_log = 'ON';
443 • SET GLOBAL long_query_time = 2; -- Consultas que tarden más de 2 segundos
```

Usamos EXPLAIN para que nos ayude a entender cómo se procesarán las tablas y si se utilizarán índices de manera eficiente.

```
-- Para optimizar una consulta
EXPLAIN
SELECT Clientes.Nombre, Funciones.PeliculaID, Boletos.AsientosReservados
FROM Clientes
JOIN Boletos ON Clientes.ClienteID = Boletos.ClienteID
JOIN Funciones ON Boletos.FuncionID = Funciones.FuncionID
WHERE Boletos.PrecioTotal > 100.00;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref
	1	SIMPLE	Boletos	NULL	ALL	fk_funcion,idx_clienteid	NULL	NULL	NULL
	1	SIMPLE	Cientes	NULL	eq_ref	PRIMARY	PRIMARY	4	cine.Boletos.ClienteID
	1	SIMPLE	Funciones	NULL	eq_ref	PRIMARY	PRIMARY	4	cine.Boletos.FuncionID

Aquí también usamos EXPLAIN para analizar cómo MySQL ejecuta una consulta más compleja con varios JOIN, lo cual es útil para identificar las ineficiencias que hay en la ejecución.

```
-- Revisaremos la ejecución de consultas con múltiples JOIN
EXPLAIN
SELECT Clientes.Nombre, Funciones.PeliculaID, Peliculas.Titulo, Boletos.AsientosReservados, Boletos.PrecioTotal
FROM Clientes
JOIN Boletos ON Clientes.ClienteID = Boletos.ClienteID
JOIN Funciones ON Boletos.FuncionID = Funciones.FuncionID
JOIN Peliculas ON Funciones.PeliculaID = Peliculas.PeliculaID
WHERE Boletos.PrecioTotal > 100.00;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
	1	SIMPLE	Boletos	NULL	ALL	fk_funcion,idx_clienteid	NULL	NULL	NULL	2	50.00	Using where
	1	SIMPLE	Cientes	NULL	eq_ref	PRIMARY	PRIMARY	4	cine.Boletos.ClienteID	1	100.00	NULL
	1	SIMPLE	Funciones	NULL	eq_ref	PRIMARY,idx_pelculaaid,idx_pelcula_fecha	PRIMARY	4	cine.Boletos.FuncionID	1	100.00	Using where
	1	SIMPLE	Peliculas	NULL	eq_ref	PRIMARY	PRIMARY	4	cine.Funciones.PeliculaID	1	100.00	NULL

evaluamos si las columnas más utilizadas en las consultas tienen índices apropiados para mejorar el rendimiento.

```
463 • SHOW INDEXES FROM Boletos;
464 • SHOW INDEXES FROM Funciones;
```

Aquí también se usa EXPLAIN para analizar cómo se ejecuta una consulta que filtra por Columna respectiva en la tabla que escojamos

```
EXPLAIN SELECT * FROM Boletos WHERE ClienteID = 1;
EXPLAIN SELECT * FROM Funciones WHERE PeliculaID = 2;
```

Usamos SHOW ERRORS para que nos muestre los errores que se hayan producido. Luego verificamos el estado de cada tabla.

```
SHOW ERRORS;
SHOW TABLE STATUS WHERE Name = 'Funciones';
SHOW TABLE STATUS WHERE Name = 'Peliculas';
```

Investigación: Investigar las mejores prácticas para monitorear el rendimiento de las consultas en producción.

Utilizar el log de consultas lentas: Configurar y habilitar el log de consultas lentas es una de las mejores formas de identificar problemas de rendimiento. Esto permite registrar todas las consultas que tardan más de un tiempo determinado en ejecutarse, lo que facilita la localización de consultas ineficientes. Se debe ajustar el umbral de long_query_time para que el sistema registre únicamente las consultas que exceden el tiempo establecido, ayudando a optimizar las que están causando problemas

Monitorizar el uso de recursos del sistema: Es fundamental supervisar el uso de CPU, memoria y disco durante las consultas en producción. Herramientas como SHOW PROCESSLIST en MySQL o pg_stat_activity en PostgreSQL permiten ver en tiempo real qué consultas están en ejecución y si están consumiendo demasiados recursos. El monitoreo de estas métricas puede ayudar a identificar cuellos de botella relacionados con el hardware y a ajustar las configuraciones del servidor o la base de datos

Optimizar consultas con índices adecuados: Una práctica clave es asegurarse de que las consultas estén utilizando índices de manera eficiente. El uso de herramientas como EXPLAIN o EXPLAIN ANALYZE ayuda a visualizar los planes de ejecución y detectar consultas que podrían beneficiarse de un índice adicional o un cambio en la estructura de la consulta. El análisis de los planes de ejecución proporciona una visión detallada de cómo se están accediendo a los datos y ayuda a realizar mejoras sustanciales en el rendimiento

Implementar alertas proactivas: Es recomendable configurar sistemas de monitoreo que emitan alertas cuando las consultas superen los umbrales de rendimiento establecidos. Herramientas como Prometheus con Grafana, New Relic o incluso servicios específicos de bases de datos proporcionan métricas sobre el rendimiento de las consultas y pueden alertar a los administradores si se detectan anomalías o consultas lentas que podrían afectar la experiencia del usuario

Realizar análisis de tendencias y carga de trabajo: Monitorizar las consultas en producción a lo largo del tiempo permite identificar patrones y cargas de trabajo inusuales. Es fundamental revisar las métricas de rendimiento durante los picos de tráfico o en momentos de alta demanda para asegurarse de que las consultas siguen siendo eficientes bajo diferentes condiciones. La utilización de herramientas que almacenen el historial de consultas y las comparen en distintas franjas horarias puede ayudar a mejorar la capacidad de respuesta del sistema.

Importancia del Conocimiento: El monitoreo proactivo puede identificar cuellos de botella antes de que afecten el rendimiento del sistema

7.1. Realizar pruebas de carga.

Práctica: Simular múltiples usuarios concurrentes usando herramientas como Apache JMeter para ver cómo responde la base de datos bajo alta carga.

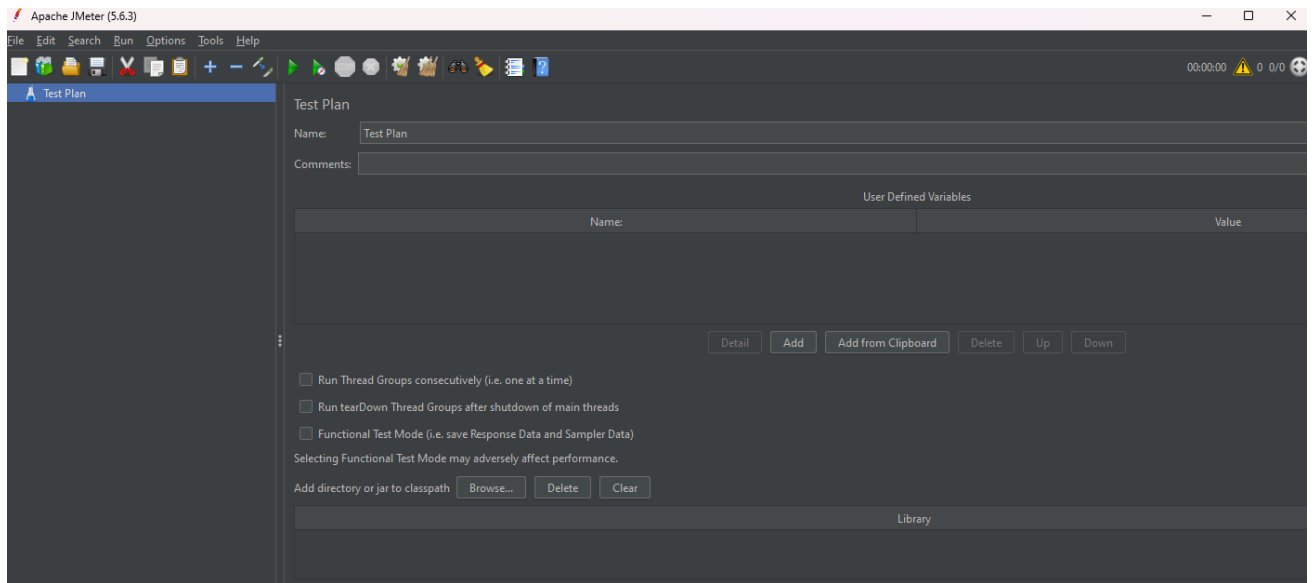
Primero nos dirigimos a la página principal de Apache JMeter para instalarlo

Apache JMeter 5.6.3 (Requires Java 8+)

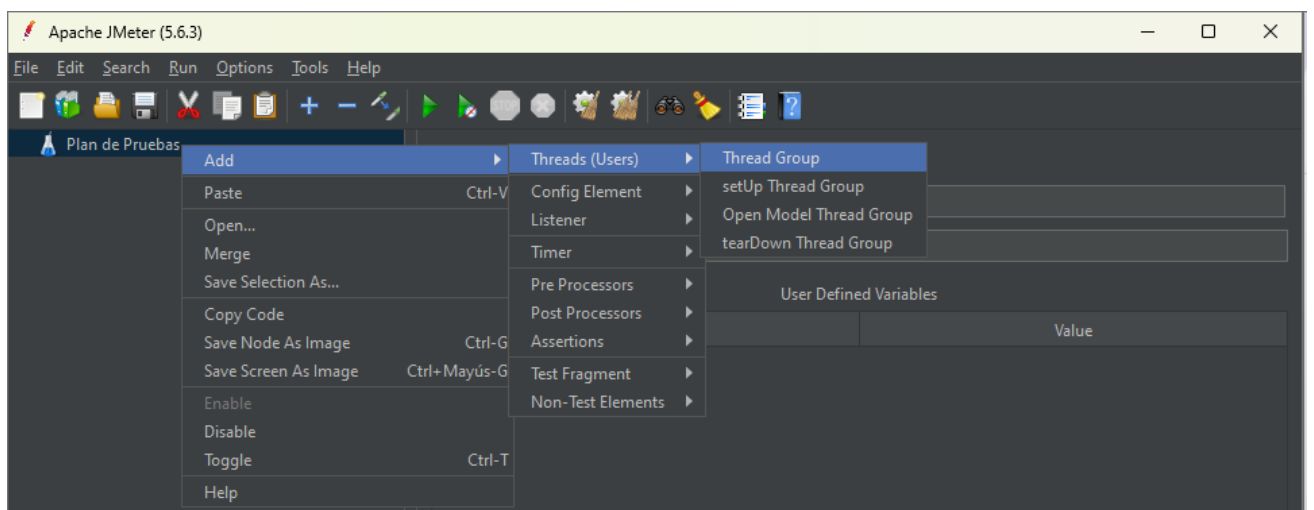
Binaries

[apache-jmeter-5.6.3.tgz](#) [sha512](#) [pgp](#)

El Test Plan es el contenedor principal de la prueba en JMeter. Aquí vamos a definir todos los elementos y configuraciones necesarios para ejecutar la prueba de carga



El Thread Group nos ayudara a simular múltiples usuarios que ejecutarán las peticiones de forma concurrente.



Estos parámetros configuran el número de usuarios simulados, el tiempo que JMeter usará para iniciar todos los hilos (Ramp-Up), y cuántas veces cada hilo ejecutará las acciones (Loop Count).

Thread Group

Name:

Comments:

Action to be taken after a Sampler error

☒ Continue
 ☐ Start Next Thread Loop
 ☐ Stop Thread
 ☐ Stop Test
 ☐ Stop Test Now

Thread Properties

Number of Threads (users):

Ramp-up period (seconds):

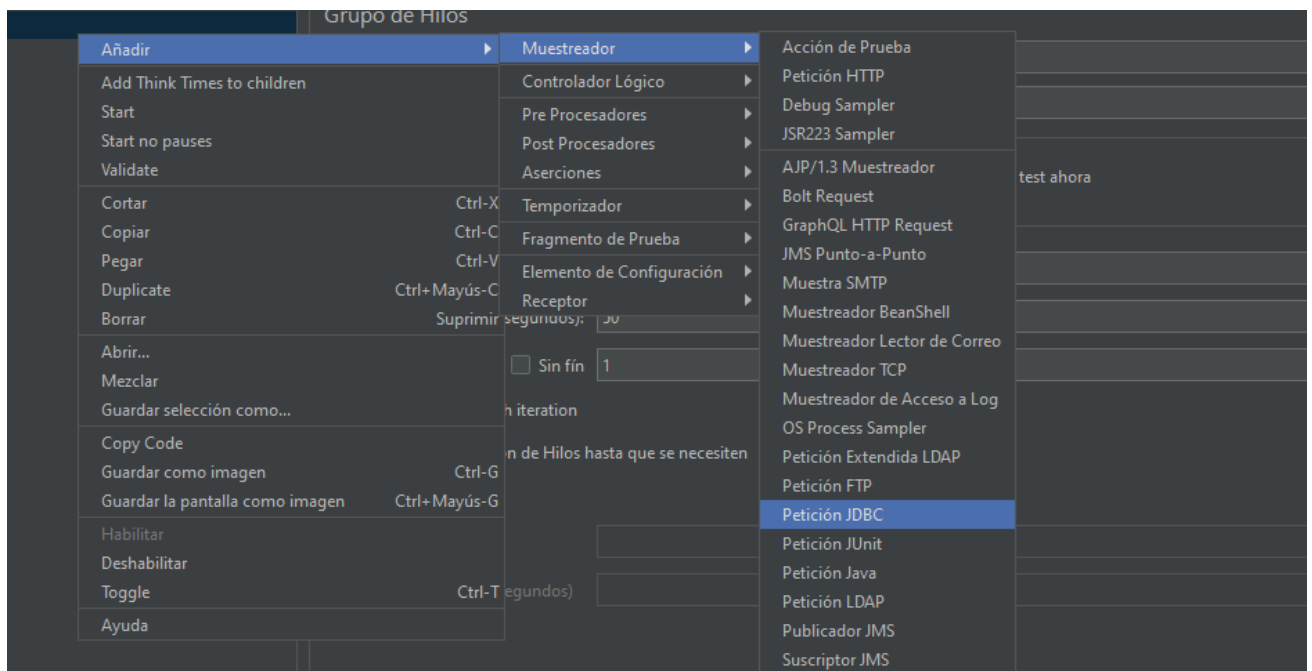
Loop Count: ☐ Infinite

☒ Same user on each iteration
☐ Delay Thread creation until needed
☐ Specify Thread lifetime

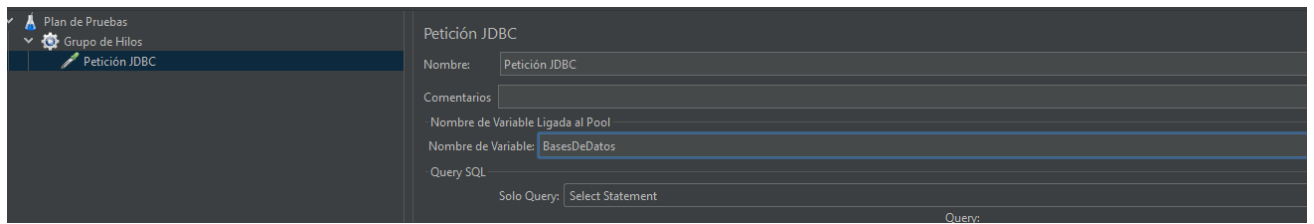
Duration (seconds):

Startup delay (seconds):

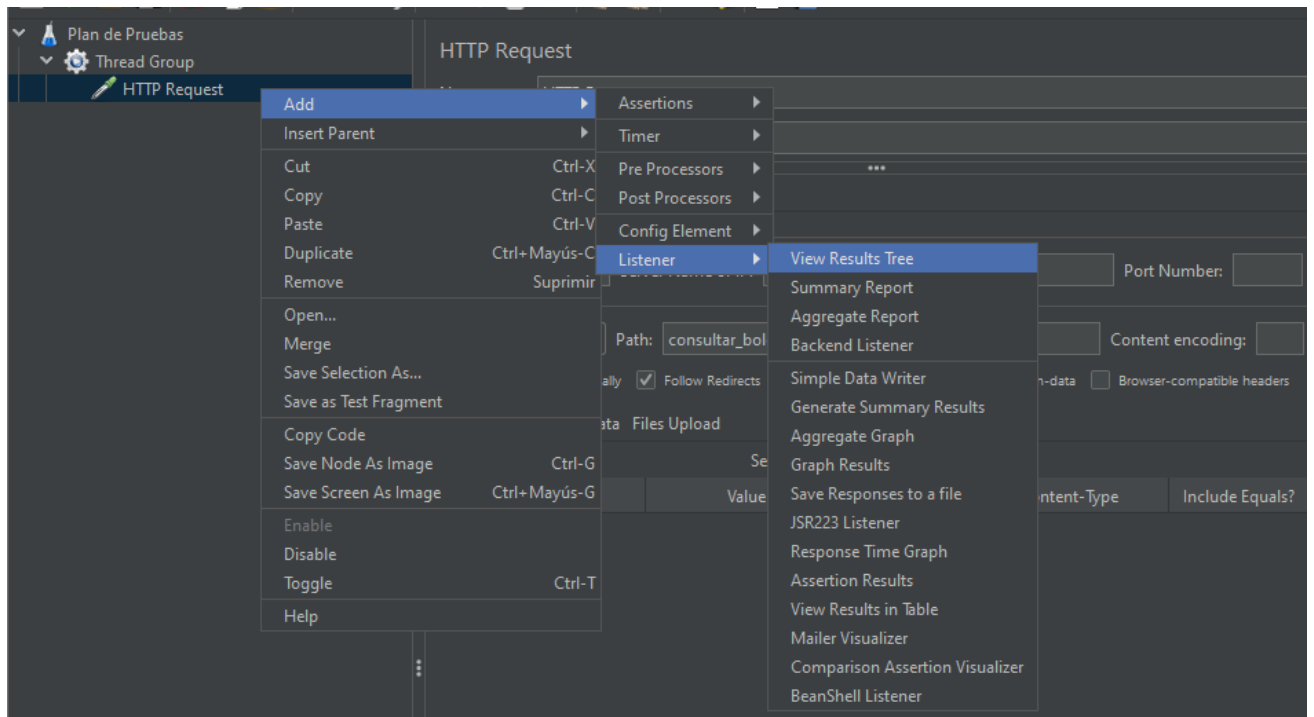
Especificamos el tipo de consulta y la conexión a la base de datos a la que se vamos a acceder para ejecutar las consultas:



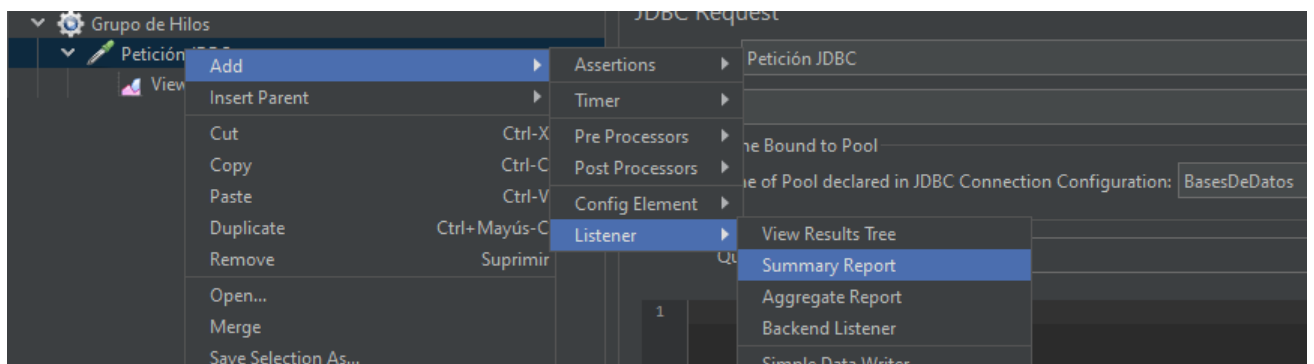
Le agregamos un nombre de Variable y le agregamos el Query ue se ejecutara durante la prueba `SELECT * FROM boletos WHERE precio_total > 100`



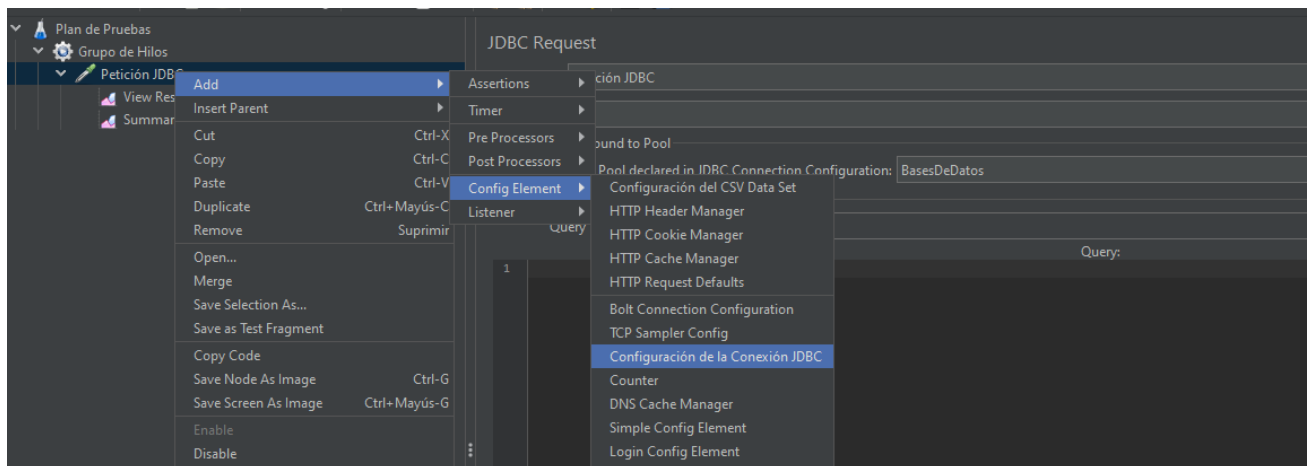
El “View Results Tree” nos va a permitir visualizar el detalle de cada solicitud JDBC realizada durante la prueba, incluyendo la respuesta y los posibles errores.



En cambio el “Summary Report” nos mostrara un resumen de las métricas generales de la prueba, como tiempos de respuesta promedio, número de solicitudes por segundo y tasa de errores



Aquí configuramos la conexión JDBC a la base de datos



Le damos a iniciar



Y este sería nuestro resultado final:

The screenshot shows the 'View Results in Table' window in JMeter. The table displays the results of the JDBC Request test. The columns are: Sample #, Start Time, Thread Name, Label, Sample Time, Status, Bytes, Sent Bytes, Latency, and Connect Time. The table contains 20 rows of data, all with a status of 'Success' and a latency of 0 ms.

Sample #	Start Time	Thread Name	Label	Sample Time	Status	Bytes	Sent Bytes	Latency	Connect Time
1	17:11:41.781	Thread Group ...	JDBC Request	325	Success	887	0	0	267
2	17:11:41.827	Thread Group ...	JDBC Request	381	Success	887	0	0	401
3	17:11:41.827	Thread Group ...	JDBC Request	489	Success	887	0	0	457
4	17:11:42.118	Thread Group ...	JDBC Request	2	Success	887	0	0	0
5	17:11:42.118	Thread Group ...	JDBC Request	2	Success	887	0	0	0
6	17:11:42.118	Thread Group ...	JDBC Request	2	Success	887	0	0	0
7	17:11:42.122	Thread Group ...	JDBC Request	2	Success	887	0	0	0
8	17:11:42.122	Thread Group ...	JDBC Request	2	Success	887	0	0	0
9	17:11:42.122	Thread Group ...	JDBC Request	2	Success	887	0	0	0
10	17:11:42.124	Thread Group ...	JDBC Request	2	Success	887	0	0	0
11	17:11:42.124	Thread Group ...	JDBC Request	2	Success	887	0	0	0
12	17:11:42.124	Thread Group ...	JDBC Request	2	Success	887	0	0	0
13	17:11:42.127	Thread Group ...	JDBC Request	2	Success	887	0	0	0
14	17:11:42.127	Thread Group ...	JDBC Request	2	Success	887	0	0	0
15	17:11:42.127	Thread Group ...	JDBC Request	4	Success	887	0	0	0
16	17:11:42.129	Thread Group ...	JDBC Request	2	Success	887	0	0	0
17	17:11:42.129	Thread Group ...	JDBC Request	2	Success	887	0	0	0
18	17:11:42.131	Thread Group ...	JDBC Request	0	Success	887	0	0	0
19	17:11:42.131	Thread Group ...	JDBC Request	0	Success	887	0	0	0
20	17:11:42.131	Thread Group ...	JDBC Request	0	Success	887	0	0	0

Investigación: Investigar cómo realizar pruebas de estrés y carga en bases de datos de alto rendimiento.

Las pruebas de carga simulan un número normal o alto de usuarios para ver cómo se comporta la base de datos bajo condiciones normales o de alta demanda. Las pruebas de estrés van más allá y prueban qué sucede cuando la base de datos recibe más consultas o usuarios de los que puede manejar, buscando el punto en el que comienza a fallar.

¿Cómo Se Hacen?: Se usan herramientas como JMeter o Sysbench para simular múltiples usuarios que hacen consultas a la base de datos al mismo tiempo. Estas herramientas nos permiten ver cómo se comporta la base de datos al aumentar el número de usuarios. Durante la prueba, se mide el tiempo que tardan las consultas y si la base de datos puede manejar las peticiones sin caerse.

Monitoreo del Rendimiento: Mientras se hacen las pruebas, es importante monitorear el rendimiento de la base de datos, como el uso de CPU, memoria y tiempos de respuesta de las consultas. Usando herramientas como Grafana o Prometheus, se pueden ver estos datos en tiempo real, lo que ayuda a identificar problemas y ajustar la configuración de la base de datos para mejorar su rendimiento.

Importancia del Conocimiento: Las pruebas de carga aseguran que el sistema sea capaz de manejar tráfico alto y crecimiento de datos.

7.2 Optimizar el uso de recursos y gestionar índices.

Práctica: Identificar índices no utilizados y eliminarlos para liberar recursos y mejorar la velocidad de las operaciones de escritura.

Para optimizar el rendimiento de la base de datos, se identificaron índices innecesarios y se realizaron mejoras en la gestión de índices:

```
475      -- Verificamos los indices de nuestra base de datos
476 •    SHOW INDEX FROM Clientes;
477 •    SHOW INDEX FROM Peliculas;
478 •    SHOW INDEX FROM Boletos;
479 •    SHOW INDEX FROM Funciones;
```

Eliminar índices no utilizados: Se determinó que el índice `idx_peliculaid` en la tabla `Funciones` es redundante, ya que `PeliculaID` es una clave foránea y MySQL ya genera automáticamente un índice para este campo.

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
►	funciones	0	PRIMARY	1	FuncionID	A	2	NULL	NULL		BTREE			YES	NULL
	funciones	0	idx_fechaHora	1	FechaHora	A	2	NULL	NULL		BTREE			YES	NULL
	funciones	1	idx_peliculaid	1	PeliculaID	A	2	NULL	NULL	YES	BTREE			YES	NULL

Optimizar la búsqueda combinada: Se creó un índice compuesto `idx_pelicula_fecha` en la tabla `Funciones` con las columnas `PeliculaID`, `FechaHora`, optimizando consultas que buscan funciones por película y fecha.

```
---
481 •    CREATE INDEX idx_pelicula_fecha ON Funciones (PeliculaID, FechaHora);
```

Optimizar la gestión de boletos: Se creó un índice compuesto `idx_cliente_funcion` en `Boletos` (`ClienteID`, `FuncionID`), mejorando la eficiencia en la búsqueda de boletos de un cliente en una función específica.

```
484 •    CREATE INDEX idx_cliente_funcion ON Boletos (ClienteID, FuncionID);
```

Investigación: Investigar cómo ajustar el número de índices según el tipo de consulta (lectura/escritura).

Ajustar el número de índices en una base de datos según el tipo de consulta (lectura o escritura) es esencial para optimizar el rendimiento. Los índices aceleran las consultas de lectura, pero pueden ralentizar las operaciones de escritura debido al tiempo adicional necesario para mantenerlos actualizados. A continuación, se detallan las consideraciones clave:

1. Índices para Consultas de Lectura (SELECT):

Las consultas de lectura se benefician significativamente de los índices, ya que permiten localizar rápidamente los datos sin necesidad de escanear toda la tabla. Es recomendable crear índices en las columnas que se utilizan con frecuencia en las cláusulas WHERE, JOIN o ORDER BY de las consultas. Sin embargo, es importante no crear demasiados índices, ya que cada índice adicional puede afectar el rendimiento de las operaciones de escritura.

2. Índices para Consultas de Escritura (INSERT, UPDATE, DELETE):

Las operaciones de escritura pueden volverse más lentas si la base de datos tiene demasiados índices, ya que cada operación de escritura requiere actualizar todos los índices asociados a las columnas modificadas. Por lo tanto, es aconsejable ser selectivo con los índices que se crean, enfocándose en aquellos que son esenciales para las consultas de lectura más frecuentes. Además, es importante considerar el tipo de índice; por ejemplo, los índices de árbol B son eficientes para consultas de rango, mientras que los índices hash son más adecuados para búsquedas exactas.

3. Mantenimiento de Índices:

Con el tiempo, los índices pueden fragmentarse, lo que puede afectar negativamente el rendimiento. Es crucial realizar un mantenimiento regular de los índices, como la reconstrucción o reorganización, para garantizar su eficiencia. Además, es recomendable monitorear el rendimiento de las consultas y ajustar los índices según sea necesario, eliminando aquellos que no se utilizan o que tienen un impacto negativo en el rendimiento.

En resumen, la clave es equilibrar la creación de índices según el tipo de consulta predominante en la base de datos, asegurando que las consultas de lectura se beneficien de índices adecuados sin comprometer el rendimiento de las operaciones de escritura.

Importancia del Conocimiento: La optimización de los recursos asegura un uso eficiente del hardware y mejora la escalabilidad.

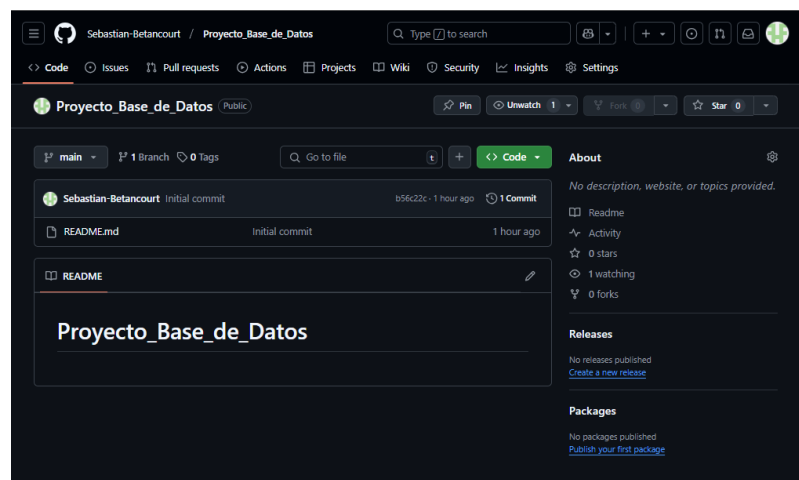
8. Git y Control de Versiones

Objetivo: Asegurar que el código relacionado con la base de datos esté versionado y que el equipo pueda colaborar de manera eficiente.

8.1 Configuración del Repositorio

Práctica: Inicializar un repositorio en Git y subir los archivos de definición de la base de datos, scripts de SQL y procedimientos almacenados.

Link del repositorio: https://github.com/Sebastian-Betancourt/Proyecto_Base_de_Datos.git



Investigación: Investigar buenas prácticas de flujo de trabajo en Git (por ejemplo, uso de ramas, git merge).

1. Uso de Ramas (Branches)

El uso de ramas permite mantener el código limpio y organizado.

Algunos enfoques comunes son:

Rama principal (main o master): Contiene la versión estable del proyecto.

Rama de desarrollo (develop): Contiene el código en desarrollo antes de fusionarlo con la principal.

Ramas de características (feature): Se crean para nuevas funcionalidades y se eliminan tras su integración.

Ramas de corrección de errores (bugfix o hotfix): Para solucionar errores sin afectar la rama principal.

Ejemplo para crear una rama y cambiar a ella:

`git checkout -b feature/nueva-funcionalidad`

2. Commits Pequeños y Descriptivos

Cada commit debe representar un cambio específico con un mensaje claro:

```
git commit -m "Corrige bug en el cálculo de precios"
```

3. Uso de Pull Requests (PRs)

Antes de fusionar cambios, es recomendable hacer un pull request (PR) para revisión del equipo.

4. Mantener el Repositorio Limpio

Eliminar ramas que ya no se usan:

```
git branch -d feature/rama-antigua
```

Ignorar archivos innecesarios con un .gitignore.

Importancia del Conocimiento: Git permite la colaboración y el manejo eficiente de cambios en el código, especialmente cuando se trabaja en equipo.

8.2 Realizar commits frecuentes y con mensajes claros.

Práctica: Hacer commits regularmente, describiendo claramente los cambios realizados en los scripts SQL y la estructura de la base de datos.

Para mantener un flujo de trabajo organizado y asegurar que cada integrante cumpliera con su parte, establecimos un esquema de trabajo basado en la división de tareas y la entrega por etapas. Cada compañero fue responsable de una sección específica del desarrollo del proyecto y debía enviar periódicamente los avances.

Cada vez que un integrante completaba su parte, los archivos eran revisados por el equipo antes de que el siguiente continuara con el desarrollo. Este método nos permitió coordinar el trabajo de manera efectiva, asegurando que cada miembro pudiera avanzar sin contratiempos y manteniendo un registro claro de los progresos a lo largo del proyecto.

Investigación: Investigar cómo utilizar git rebase y git pull para evitar conflictos.

En un flujo de trabajo con Git, es común que los equipos colaboren en el mismo proyecto y realicen cambios en diferentes partes del código o la base de datos. Esto puede generar conflictos cuando se combinan esos cambios. Para evitar estos conflictos y mantener un historial limpio y organizado, existen herramientas y comandos útiles como git rebase y git pull. Aunque en nuestro proyecto no usamos Git

directamente, podemos entender cómo estas herramientas hubieran sido útiles si lo hubiésemos implementado.

Uso de git pull

El comando git pull se utiliza para traer cambios del repositorio remoto al local y fusionarlos con la rama en la que estás trabajando. Sin embargo, si varios miembros del equipo están trabajando en diferentes partes del proyecto y no sincronizan frecuentemente sus cambios, pueden ocurrir conflictos de fusión cuando intentan combinar los cambios.

Uso correcto: Es recomendable hacer un git pull antes de comenzar a trabajar en cualquier nueva funcionalidad para asegurarse de que tu rama local esté actualizada con respecto a la rama remota. De esta manera, puedes minimizar los conflictos al trabajar sobre la versión más reciente del proyecto.

Estrategia para evitar conflictos: Si un compañero ha realizado cambios significativos en una parte del código que afecten a tu área, hacer un git pull antes de empezar a trabajar te ayudará a ver si hay conflictos en las modificaciones. Si hay conflictos, puedes resolverlos antes de continuar con nuevas modificaciones.

Uso de git rebase

El comando git rebase se utiliza para aplicar los cambios de una rama sobre otra, lo que resulta en un historial de commits más limpio y lineal. Es particularmente útil cuando varios desarrolladores trabajan en ramas diferentes y es necesario incorporar los cambios de otros antes de realizar un git merge.

Uso correcto: Usar git rebase permite mantener un historial más limpio al "reaplicar" los commits de una rama sobre la rama base. En lugar de un git merge, que crea un commit de fusión, el rebase aplica los cambios directamente sobre la base sin generar un commit extra.

Estrategia para evitar conflictos: Si varios compañeros están trabajando en diferentes partes del proyecto, el git rebase puede ayudar a evitar conflictos al aplicar los cambios más recientes a tu rama de trabajo antes de subirlos al repositorio. Esto permite resolver conflictos a medida que surgen, en lugar de al final del proceso, lo que puede hacer que la resolución de conflictos sea más sencilla.

Importancia del Conocimiento: Un flujo de trabajo claro en Git mejora la colaboración y la gestión de versiones.

8.3 Automatización de Pruebas Práctica: Crear flujos de trabajo de CI/CD que automaticen las pruebas de las consultas SQL y otros scripts relacionados con la base de datos.

Investigación: Investigar sobre integración continua y cómo aplicarla en bases de datos con GitHub Actions.

La integración continua en bases de datos es un proceso que permite validar y desplegar cambios en la estructura de la base de datos de manera automatizada. Su objetivo es reducir errores y asegurar que todas las modificaciones sean compatibles con el sistema.

Uso de GitHub Actions para CI/CD en bases de datos

GitHub Actions es una herramienta que permite definir flujos de trabajo automatizados dentro de un repositorio de GitHub. Para bases de datos, se pueden configurar acciones que ejecuten pruebas en scripts SQL cada vez que se realicen cambios en el código.

Importancia del Conocimiento: Las pruebas automáticas aseguran que las bases de datos se mantengan consistentes y funcionales a lo largo del tiempo.

9. Conclusiones y Recomendaciones

Conclusión

El desarrollo de este proyecto nos permitió profundizar en la aplicación de bases de datos dentro de un entorno real, en este caso, un sistema de gestión para un cine. A lo largo del proceso, diseñamos una base de datos estructurada con entidades clave como Clientes, Películas, Funciones, Boletos y Pagos, asegurando la correcta organización y gestión de la información.

Al trabajar en equipo, logramos distribuir eficientemente las tareas, aunque la falta de un sistema de control de versiones como Git limitó la sincronización óptima del código y la base de datos. Sin embargo, el uso de herramientas como WhatsApp para compartir avances y evidencias nos permitió coordinar las entregas y el desarrollo progresivo del proyecto.

Recomendación

Implementar un sistema de control de versiones: En futuros proyectos, se recomienda el uso de Git para gestionar los cambios en los scripts SQL y facilitar la colaboración entre los integrantes del equipo.

Optimizar el diseño de la base de datos: Seguir buenas prácticas de normalización y considerar índices para mejorar el rendimiento en consultas de gran volumen de datos.

Automatizar pruebas de consultas SQL: La integración de herramientas de CI/CD como GitHub Actions permitiría detectar errores antes de la implementación, asegurando mayor calidad en los scripts de la base de datos.