Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2413 - Bases de Datos

Resumen y ejercicios

Fecha: Mayo de 2024

1º semestre 2024 - Profesores: N. Alvarado - A. Reyes - E. Bustos - C.Álvarez

---

Álgebra Relacional

El álgebra relacional se basa en operaciones sobre conjuntos y proporciona un conjunto de operadores para manipular relaciones (tablas) en bases de datos. Aquí te detallo algunas operaciones comunes:

I **Selección (σ)**: La operación de selección se utiliza para filtrar filas de una tabla que cumplen ciertas condiciones. La notación para la selección es $\sigma_{\text{condición}}(R)$, donde R es la relación (tabla) y *condición* es la condición que debe cumplir cada fila.

$$\sigma_{\text{edad}>30}(\text{Personas})$$

II **Proyección (π)**: La operación de proyección se utiliza para seleccionar ciertas columnas de una tabla mientras se eliminan las demás. La notación para la proyección es $\pi_{\text{columnas}}(R)$, donde R es la relación (tabla) y *columnas* son las columnas que deseas mantener.

$$\pi_{\text{nombre, edad}}(\text{Personas})$$

III **Unión ($\cup$)**: La operación de unión combina dos tablas que tienen la misma estructura (mismas columnas) y elimina duplicados. La notación para la unión es $R \cup S$, donde R y S son relaciones (tablas) que deseas unir.

$$R \cup S$$

IV **Intersección ($\cap$)**: La operación de intersección devuelve filas que están presentes en ambas tablas que estás comparando. La notación para la intersección es $R \cap S$, donde R y S son relaciones (tablas) que deseas intersectar.

$$R \cap S$$

V **Diferencia ($-$)**: La operación de diferencia devuelve filas que están en una tabla pero no en la otra. La notación para la diferencia es $R - S$, donde R y S son relaciones (tablas) que deseas comparar.

$$R - S$$

Supongamos que tenemos la siguiente tabla **Personas** con columnas *nombre*, *edad* y *ciudad*:

nombre	edad	ciudad
Juan	25	Madrid
María	35	Barcelona
Carlos	28	Sevilla
Elena	40	Valencia

I **Selección (σ)**:

$\sigma_{\text{edad}>30}(\text{Personas}) =$

nombre	edad	ciudad
María	35	Barcelona
Elena	40	Valencia

II **Proyección (π)**:

$\pi_{\text{nombre, edad}}(\text{Personas}) =$

nombre	edad
Juan	25
María	35
Carlos	28
Elena	40

III **Unión ($\cup$)**: Supongamos otra tabla **Empleados** con la misma estructura:

nombre	edad	ciudad
Laura	30	Madrid
María	35	Barcelona

Luego, al calcular la unión obtenemos:

nombre	edad	ciudad
Juan	25	Madrid
María	35	Barcelona
Carlos	28	Sevilla
Elena	40	Valencia
Laura	30	Madrid

IV **Intersección ($\cap$):**

nombre	edad	ciudad
María	35	Barcelona

V **Diferencia ($-$):**

nombre	edad	ciudad
Juan	25	Madrid
Carlos	28	Sevilla
Elena	40	Valencia

Ejercicios propuestos

I Dada la relación **Empleados** con las columnas *nombre*, *edad* y *departamento*, selecciona los nombres de los empleados que trabajan en el departamento 'Ventas'.

II Teniendo la relación **Productos** con las columnas *nombre*, *precio* y *stock*, encuentra los nombres y precios de los productos que tienen un precio mayor a $50.

III Supongamos dos relaciones: **Clientes_A** y **Clientes_B**, cada una con las columnas *nombre* y *edad*. Encuentra los nombres de clientes que están en la lista de **Clientes_A** pero no en la lista de **Clientes_B**.

IV Dada una relación **Estudiantes** con las columnas *nombre*, *carrera* y *año*, encuentra los nombres de estudiantes que estudian Ingeniería y están en su tercer año.

V Utilizando las relaciones **Pedidos** y **Clientes**, donde **Pedidos** tiene las columnas *cliente_id*, *producto* y *cantidad*, y **Clientes** tiene las columnas *cliente_id* y *nombre*, encuentra los nombres de los clientes que han realizado pedidos de productos en más de una ocasión.

Modelo Entidad-Relacion

Llaves

Las llaves desempeñan un papel fundamental en garantizar la integridad y la estructura de las relaciones (tablas).

- **Llave Primaria (Primary Key):**
  - Una **llave primaria** es un atributo (o conjunto de atributos) en una tabla que identifica de forma única a cada fila en la tabla.
  - Cada tabla en una base de datos relacional debe tener una llave primaria definida, lo que significa que no puede contener valores duplicados ni valores nulos en la columna de la llave primaria.

- La llave primaria garantiza la unicidad de las filas y facilita la indexación y la búsqueda eficiente de datos.
- Ejemplo de definición en SQL:

```
CREATE TABLE Empleados (
    empleado_id INT PRIMARY KEY,
    nombre VARCHAR(50),
    departamento VARCHAR(50)
);
```

- **Llave Foránea (Foreign Key):**
  - Una **llave foránea** es un atributo (o conjunto de atributos) en una tabla que establece una relación entre dos tablas.
  - La llave foránea en una tabla hace referencia a la llave primaria de otra tabla, estableciendo así una relación de integridad referencial entre las tablas.
  - La llave foránea asegura que los valores en una columna correspondan a los valores existentes en la columna de la llave primaria de otra tabla.
  - Ejemplo de definición en SQL:

```
CREATE TABLE Pedidos (
    pedido_id INT PRIMARY KEY,
    cliente_id INT,
    producto VARCHAR(50),
    cantidad INT,
    FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
);
```

- **Llave Candidata (Candidate Key):**
  - Una **llave candidata** es un conjunto de uno o más atributos que podrían ser seleccionados como llave primaria de una tabla.
  - Una tabla puede tener múltiples llaves candidatas, pero solo una de ellas se selecciona como llave primaria.
  - Todas las llaves candidatas deben ser únicas y no nulas.
  - Ejemplo de identificación de llaves candidatas:

```
CREATE TABLE Estudiantes (
    estudiante_id INT,
    nombre VARCHAR(50),
```

```
    email VARCHAR(50),
    PRIMARY KEY (estudiante_id),
    UNIQUE (nombre),
    UNIQUE (email)
);
```

- **Llave Sustituta (Surrogate Key):**
  - Una **llave sustituta** (surrogate key) es una llave primaria artificial creada específicamente con el propósito de ser la llave principal de una tabla.
  - La llave sustituta no tiene ningún significado real o contexto externo, se utiliza solo para garantizar la unicidad y simplicidad en el diseño de la base de datos.
  - Ejemplo de definición utilizando una llave sustituta en SQL:

```
CREATE TABLE Estudiantes (
    estudiante_id INT PRIMARY KEY,
    nombre VARCHAR(50),
    email VARCHAR(50),
    UNIQUE (email)
);
```

    En este ejemplo, 'estudiante_id' es una llave sustituta que se utiliza como la llave principal, aunque 'nombre' y 'email' también podrían haber sido candidatos para la llave primaria.

ER

El Modelo Entidad-Relación (ER) es un enfoque utilizado en el diseño de bases de datos para describir y visualizar la estructura lógica de una base de datos. En el modelo ER, se utilizan conceptos como entidades, atributos y relaciones para representar la información y las interacciones dentro del sistema de información.

- **Entidad:** Una **entidad** representa un objeto o concepto del mundo real que puede ser identificado claramente. Por ejemplo, en un sistema de gestión de bibliotecas, las entidades podrían incluir libros, autores y estudiantes.
- **Atributo:** Un **atributo** es una característica o propiedad de una entidad que describe las características de la entidad. Por ejemplo, un libro puede tener atributos como título, ISBN y año de publicación.
- **Relación:** Una **relación** describe la asociación o conexión entre dos o más entidades. Puede ser una relación uno a uno, uno a muchos o muchos a muchos. Por ejemplo,

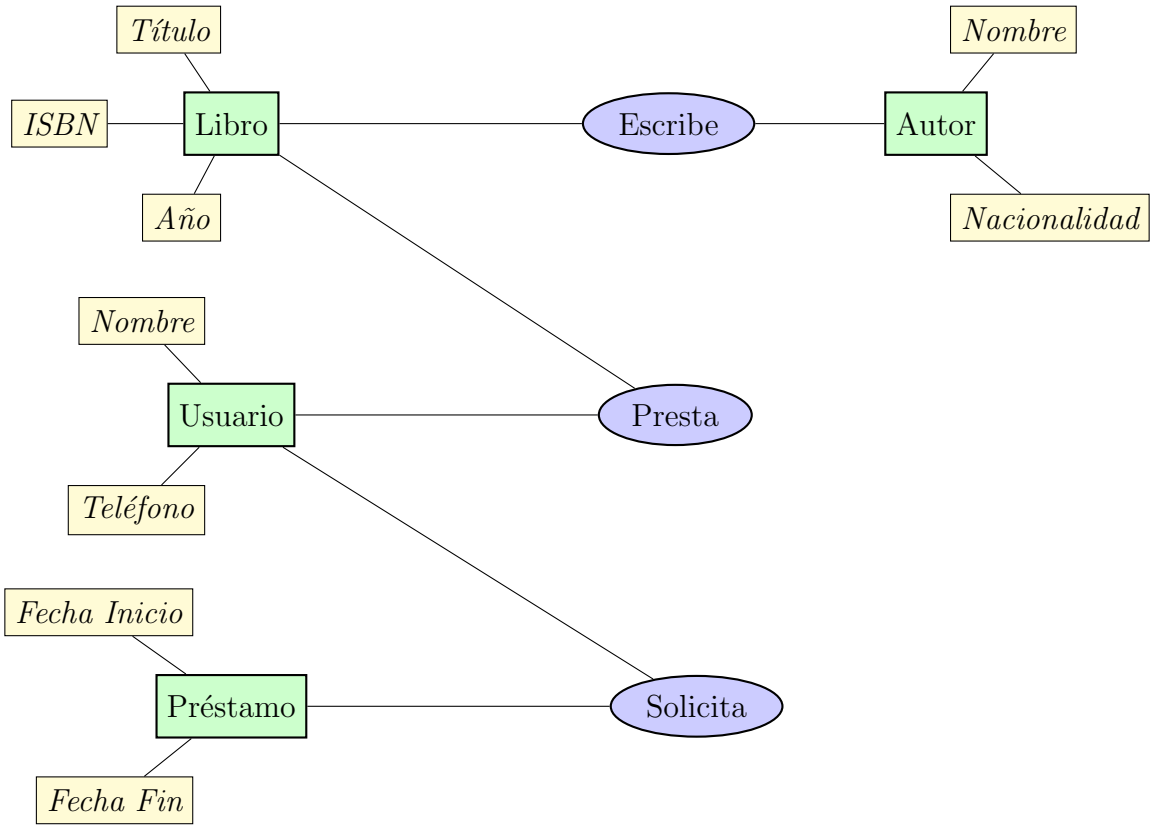
la relación entre un libro y un autor puede ser ün libro tiene un autor"(uno a muchos).

- **Llave Primaria:** La **llave primaria** es un atributo (o conjunto de atributos) que identifica de manera única cada instancia de una entidad. Es crucial para la identificación y la integridad de los datos en la base de datos.
- **Diagrama ER:** Un **diagrama Entidad-Relación (ER)** es una representación gráfica del modelo ER que muestra entidades como rectángulos, atributos como elipses y relaciones como líneas conectando entidades.

El modelo ER utiliza varios componentes para representar la estructura de la base de datos:

- **Entidades:** Se representan como rectángulos en el diagrama ER, con el nombre de la entidad en la parte superior del rectángulo.
- **Atributos:** Se representan como elipses (óvalos) dentro de los rectángulos de las entidades, describiendo las propiedades de las entidades.
- **Relaciones:** Se representan como líneas que conectan entidades en el diagrama ER, con indicadores de cardinalidad (uno a uno, uno a muchos, muchos a muchos) en las líneas.
- **Llaves Primarias y Extranjeras:** Las llaves primarias se subrayan en el diagrama ER para indicar la identificación única de cada entidad. Las llaves foráneas se muestran como atributos en una entidad que hacen referencia a la llave primaria de otra entidad.

A continuación se muestra un ejemplo simplificado de un diagrama ER de un sistema de gestión de bibliotecas:



El diagrama presentado ilustra el modelo entidad-relación para un sistema de gestión de bibliotecas. Este modelo incluye las siguientes entidades principales:

- **Libro**: Representa los libros disponibles en la biblioteca. Cada libro tiene atributos como *Título*, *ISBN* y *Año*.
- **Autor**: Representa los autores de los libros. Los atributos incluyen *Nombre* y *Nacionalidad*.
- **Usuario**: Representa los usuarios de la biblioteca. Los atributos importantes son *Nombre* y *Teléfono*.
- **Préstamo**: Gestiona los préstamos de libros a los usuarios. Incluye *Fecha de Inicio* y *Fecha de Fin* del préstamo.

Las relaciones entre estas entidades son:

- **Escribe**: Relaciona *Autor* con *Libro*, indicando qué autor escribió cada libro.
- **Presta**: Relaciona *Libro* con *Usuario*, mostrando qué libros han sido prestados a los usuarios.
- **Solicita**: Conecta *Usuario* con *Préstamo*, detallando los préstamos solicitados por cada usuario.

- **Libro**(ISBN, Título, Año)

  Cada instancia representa un libro único en la biblioteca. El ISBN es un identificador único para cada libro.

- **Autor**(ID Autor, Nombre, Nacionalidad)

  Almacena información sobre los autores de los libros. Cada autor tiene un identificador único.

- **Usuario**(ID Usuario, Nombre, Teléfono)

  Contiene detalles sobre los usuarios de la biblioteca, donde cada usuario es identificado de manera única por un ID.

- **Préstamo**(ID Préstamo, Fecha Inicio, Fecha Fin, ID Usuario, ISBN)

  Registra los préstamos de libros a los usuarios. Cada préstamo tiene un identificador único y contiene referencias (llaves foráneas) al ID del Usuario y al ISBN del Libro prestado.

Relaciones

- **Escribe** entre Autor y Libro
  - **Cardinalidad**: n a n.

    Esta relación indica qué autor ha escrito cada libro. Un autor puede escribir varios libros y un libro puede ser coescrito por varios autores.

- **Presta** entre Libro y Usuario
  - **Cardinalidad**: n a n

    Relaciona los libros con los usuarios a los que se les ha prestado. Un usuario puede tomar prestados varios libros y un libro puede ser prestado a varios usuarios a lo largo del tiempo.

Ejercicios propuestos

I Dibuja un diagrama ER para un sistema de una biblioteca que gestiona libros, autores, y préstamos. Asegúrate de incluir entidades, relaciones, y llaves primarias.

II Define el modelo entidad-relación para un sistema de gestión universitaria, incluyendo estudiantes, profesores, cursos y departamentos. Identifica las llaves primarias y foráneas.

III Crea un diagrama ER para un sistema de reservaciones de hotel, que debe incluir clientes, habitaciones, reservaciones y pagos. Explica las relaciones entre cada entidad.

IV Desarrolla un modelo entidad-relación para un sistema de venta de boletos en línea, que debe incluir eventos, boletos, clientes y transacciones de pago. Identifica todas las llaves primarias y foráneas necesarias.

V Propón un diagrama ER para un sistema de gestión de empleados para una empresa pequeña, incluyendo departamentos, empleados, proyectos y asignaciones de proyectos.

VI Describe un modelo entidad-relación para un sistema de tienda en línea, que incluya productos, categorías de productos, clientes, órdenes y detalles de órdenes. Determina las llaves primarias y foráneas.

VII Elabora un diagrama ER para un sistema de control de inventarios que gestione productos, proveedores, compras y ventas. Explica las relaciones entre las entidades y sus atributos llave.

VIII Diseña un modelo ER para un sistema de registro de pacientes en un hospital, que debe incluir pacientes, doctores, citas y tratamientos. Define claramente las llaves primarias y las llaves foráneas.

IX Crea un diagrama entidad-relación para un sistema de gestión de aerolíneas que maneje vuelos, pasajeros, boletos y tripulaciones. Identifica y explica la importancia de las llaves en el diagrama.

X Formula un modelo ER para un sistema de gestión de contenido que incluya artículos, autores, categorías y comentarios. Detalla las llaves primarias y cómo se relacionan las entidades entre sí.

SQL

SQL es un lenguaje de programación diseñado para gestionar y manipular sistemas de bases de datos relacionales. SQL es fundamental en el mundo de la tecnología de la información debido a su capacidad para permitir el acceso y la gestión de datos de manera eficiente y efectiva. Las operaciones principales que SQL puede realizar incluyen la inserción, actualización, eliminación y consulta de datos en bases de datos. Además, SQL permite la creación y modificación de esquemas de base de datos y el control de acceso a los datos.

En SQL, cada variable, conocida como **columna** en el contexto de una tabla, tiene un tipo de datos específico que define el tipo de datos que la columna puede almacenar. Los tipos de datos básicos en SQL se pueden clasificar en las siguientes categorías:

- **Tipos Numéricos**
  - `INT`: Para números enteros.
  - `DECIMAL(M, N)`: Para números decimales, donde `M` es la precisión total y `N` es el número de decimales.
  - `FLOAT`: Para números en coma flotante de precisión simple.
  - `DOUBLE`: Para números en coma flotante de doble precisión.
- **Tipos de Cadena de Caracteres**
  - `CHAR(N)`: Para cadenas de caracteres de longitud fija, donde `N` es la longitud.
  - `VARCHAR(N)`: Para cadenas de caracteres de longitud variable, hasta un máximo de `N` caracteres.
  - `TEXT`: Para cadenas de texto largo.
- **Tipos de Fecha y Hora**
  - `DATE`: Para fechas.
  - `TIME`: Para tiempo.
  - `DATETIME`: Para la combinación de fecha y tiempo.
  - `TIMESTAMP`: Para marcas de tiempo que también incluyen información de fecha y hora.
- **Tipos Lógicos**
  - `BOOLEAN`: Para valores verdadero o falso.

Cada tipo de dato en SQL está diseñado para optimizar el almacenamiento de datos y mejorar la eficiencia de las operaciones de base de datos, garantizando al mismo tiempo la integridad y la precisión de los datos.

Los comandos más básicos:

- **SELECT**: Utilizado para seleccionar datos de una base de datos.
  - **Forma de uso**: `SELECT column1, column2 FROM table_name;`
  - **Ejemplo**: `SELECT nombre, apellido FROM Empleados;`
- **INSERT INTO**: Permite insertar nuevas filas en una tabla.
  - **Forma de uso**: `INSERT INTO table_name (column1, column2) VALUES (value1, value2);`
  - **Ejemplo**: `INSERT INTO Empleados (nombre, apellido) VALUES ('Juan', 'Pérez');`

- **UPDATE**: Modifica los datos existentes en una tabla.
  - **Forma de uso**: `UPDATE table_name SET column1 = value1 WHERE condition;`
  - **Ejemplo**: `UPDATE Empleados SET apellido = 'Sánchez' WHERE id = 1;`
- **DELETE**: Elimina filas de una tabla.
  - **Forma de uso**: `DELETE FROM table_name WHERE condition;`
  - **Ejemplo**: `DELETE FROM Empleados WHERE id = 1;`
- **CREATE TABLE**: Crea una nueva tabla en la base de datos.
  - **Forma de uso**: `CREATE TABLE table_name (column1 datatype, column2 datatype, ...);`
  - **Ejemplo**: `CREATE TABLE Empleados (id INT, nombre VARCHAR(100), apellido VARCHAR(100));`
- **DROP TABLE**: Elimina una tabla de la base de datos.
  - **Forma de uso**: `DROP TABLE table_name;`
  - **Ejemplo**: `DROP TABLE Empleados;`
- **ALTER TABLE**: Utilizado para modificar la estructura de una tabla existente.
  - **Forma de uso**: `ALTER TABLE table_name ADD column_name datatype;`
  - **Ejemplo**: `ALTER TABLE Empleados ADD email VARCHAR(255);`
- **WHERE**: Especifica una condición para filtrar los registros.
  - **Forma de uso**: `SELECT column1 FROM table_name WHERE condition;`
  - **Ejemplo**: `SELECT nombre FROM Empleados WHERE apellido = 'Pérez';`
- **JOIN**: Combina filas de dos o más tablas, basado en una columna relacionada entre ellas.
  - **Forma de uso**: `SELECT table1.column1, table2.column2 FROM table1 JOIN table2 ON table1.common_field = table2.common_field;`
  - **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados JOIN Departamentos ON Empleados.depto_id = Departamentos.id;`

Tipos de Join:

- **INNER JOIN**: Devuelve filas cuando hay una coincidencia en ambas tablas.
  - **Forma de uso**: `SELECT columns FROM table1 INNER JOIN table2 ON table1.common_field = table2.common_field;`

- **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados INNER JOIN Departamentos ON Empleados.depto_id = Departamentos.id;`

- **LEFT JOIN** (o **LEFT OUTER JOIN**): Devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha. Si no hay coincidencia, los resultados son NULL en el lado derecho.
  - **Forma de uso**: `SELECT columns FROM table1 LEFT JOIN table2 ON table1.common_field = table2.common_field;`
  - **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados LEFT JOIN Departamentos ON Empleados.depto_id = Departamentos.id;`
- **RIGHT JOIN** (o **RIGHT OUTER JOIN**): Devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si no hay coincidencia, los resultados son NULL en el lado izquierdo.
  - **Forma de uso**: `SELECT columns FROM table1 RIGHT JOIN table2 ON table1.common_field = table2.common_field;`
  - **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados RIGHT JOIN Departamentos ON Empleados.depto_id = Departamentos.id;`
- **FULL JOIN** (o **FULL OUTER JOIN**): Devuelve filas cuando hay una coincidencia en una de las tablas.
  - **Forma de uso**: `SELECT columns FROM table1 FULL OUTER JOIN table2 ON table1.common_field = table2.common_field;`
  - **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados FULL OUTER JOIN Departamentos ON Empleados.depto_id = Departamentos.id;`
- **CROSS JOIN**: Produce el producto cartesiano de todas las filas de las tablas involucradas.
  - **Forma de uso**: `SELECT columns FROM table1 CROSS JOIN table2;`
  - **Ejemplo**: `SELECT Empleados.nombre, Departamentos.nombre FROM Empleados CROSS JOIN Departamentos;`

Algunos ejemplos de consultas:

- `SELECT nombre FROM Empleados WHERE departamento = 'Ventas';`

$$\sigma_{\text{departamento='Ventas'}}(\text{Empleados})$$

Ambas consultas obtienen los nombres de los empleados que pertenecen al departamento de ventas.

- `SELECT nombre, apellido FROM Empleados;`

$$\pi_{\text{nombre, apellido}}(\text{Empleados})$$

Estas consultas listan los nombres y apellidos de todos los empleados.

- ```
  SELECT nombre FROM Empleados
  UNION
  SELECT nombre FROM Clientes;
  ```

$$\pi_{\text{nombre}}(\text{Empleados}) \cup \pi_{\text{nombre}}(\text{Clientes})$$

Estas consultas generan una lista de nombres únicos tanto de empleados como de clientes.

- ```
  SELECT nombre FROM Empleados
  INTERSECT
  SELECT nombre FROM Clientes;
  ```

$$\pi_{\text{nombre}}(\text{Empleados}) \cap \pi_{\text{nombre}}(\text{Clientes})$$

Estas consultas encuentran los nombres que son comunes entre empleados y clientes.

- ```
  SELECT nombre FROM Empleados
  EXCEPT
  SELECT nombre FROM Clientes;
  ```

$$\pi_{\text{nombre}}(\text{Empleados}) - \pi_{\text{nombre}}(\text{Clientes})$$

Estas consultas devuelven los nombres que aparecen entre los empleados pero no entre los clientes.

- ```
  SELECT Empleados.nombre, Empleados.departamento, Departamentos.mánager
  FROM Empleados
  JOIN Departamentos ON Empleados.departamento = Departamentos.nombre;
  ```

$$\text{Empleados} \bowtie_{\text{Empleados.departamento=Departamentos.nombre}} \text{Departamentos}$$

Estas consultas combinan la información de empleados y departamentos basándose en el departamento al que pertenece cada empleado y listan los nombres de los empleados, sus departamentos y los mánagers de cada departamento.

El comando **GROUP BY** se utiliza en SQL para agrupar filas que tienen los mismos valores en columnas especificadas. Es frecuentemente usado junto con funciones de agregación (como COUNT, MAX, MIN, SUM, AVG) para realizar cálculos en cada grupo de filas.

- **Forma de uso**: `SELECT column1, function(column2) FROM table GROUP BY column1;`
- **Ejemplo 1**: Agrupar empleados por departamento y contarlos.

```
SELECT departamento, COUNT(*) FROM Empleados GROUP BY departamento;
```

- Esta consulta cuenta el número de empleados en cada departamento.

El comando **HAVING** se utiliza en SQL para añadir una condición de filtro a las funciones de agregación, similar a WHERE pero utilizado después de un GROUP BY.

- **Forma de uso**: `SELECT column1, function(column2) FROM table GROUP BY column1 HAVING condition;`
- **Ejemplo 1**: Agrupar empleados por departamento y contar solo aquellos departamentos con más de 10 empleados.

```
SELECT departamento, COUNT(*) FROM Empleados
GROUP BY departamento HAVING COUNT(*) > 10;
```

- Esta consulta devuelve los departamentos que tienen más de 10 empleados.

Las **consultas anidadas** en SQL son consultas que se utilizan dentro de otras consultas SQL para proporcionar datos que serán utilizados por la consulta externa. Pueden ser muy útiles para realizar comparaciones complejas y cálculos.

- **Forma de uso general**: `SELECT column1 FROM table1 WHERE column2 OPERATOR (SELECT column3 FROM table2);`
- **Ejemplo 1**: Seleccionar empleados cuyo salario es mayor que el promedio de los salarios.

```
SELECT nombre FROM Empleados
WHERE salario > (SELECT AVG(salario) FROM Empleados);
```

- Esta consulta devuelve los nombres de los empleados que ganan más que el salario promedio.

Ejercicios propuestos

I Escribe una consulta SQL para encontrar todos los empleados que trabajan en el departamento 'Ventas'.

II ¿Qué hace el siguiente código SQL? `SELECT nombre, salario FROM Empleados WHERE salario >(SELECT AVG(salario) FROM Empleados);`

III Escribe una consulta SQL para mostrar la suma total de salarios pagados por departamento.

IV Explica qué hace esta consulta SQL: `SELECT COUNT(*), ciudad FROM Empleados GROUP BY ciudad HAVING COUNT(*) >5;`

V Escribe una consulta SQL para encontrar los tres salarios más altos en la empresa.

VI ¿Qué devuelve este código SQL? `SELECT * FROM Clientes WHERE NOT EXISTS (SELECT * FROM Pedidos WHERE Clientes.cliente_id = Pedidos.cliente_id);`

VII Escribe una consulta SQL para listar todos los productos que nunca han sido ordenados.

VIII Explica qué hace esta consulta SQL: `SELECT producto_id, COUNT(*) FROM Pedidos GROUP BY producto_id ORDER BY COUNT(*) DESC;`

IX Escribe una consulta SQL para encontrar el nombre del empleado con el tercer salario más alto.

X ¿Qué hace el siguiente código SQL? `SELECT nombre, fecha_ingreso FROM Empleados WHERE fecha_ingreso BETWEEN '2020-01-01' AND '2020-12-31';`

XI Escribe una consulta SQL que liste todos los clientes junto con el número de pedidos que han hecho, ordenado por número de pedidos de forma descendente.

XII Explica qué realiza este código SQL: `UPDATE Productos SET precio = precio * 1.10 WHERE categoria = 'Electrónica';`

XIII Escribe una consulta SQL para mostrar todos los detalles de los empleados cuyo apellido comience con 'A'.

XIV ¿Qué devuelve esta consulta SQL? `DELETE FROM Empleados WHERE departamento = 'RRHH';`

XV Escribe una consulta SQL para encontrar todos los clientes que han realizado al menos un pedido en los últimos 30 días.

Dependencias y formas normales

Las formas normales son un concepto fundamental en el diseño de bases de datos relacionales, cuyo objetivo es reducir la redundancia de datos y mejorar la integridad estructural al minimizar las anomalías de inserción, actualización y eliminación. Las dependencias funcionales juegan un papel crucial en la determinación de estas formas normales.

Una **dependencia funcional**, denotada como $X \to Y$, ocurre dentro de un conjunto de atributos en una relación, donde X y Y son subconjuntos de atributos de esa relación. Se dice que Y es funcionalmente dependiente de X si y solo si cada valor de X está asociado a precisamente un valor de Y. Matemáticamente, esto se expresa como:

$$\forall x_1, x_2 \in X, \quad (x_1 = x_2) \implies (f(x_1) = f(x_2))$$

donde f es una función que mapea valores de X a valores de Y.

Una relación está en **Tercera Forma Normal (3NF)** si satisface las dos condiciones siguientes:

- Está en Segunda Forma Normal (2NF).
- No contiene dependencias transitivas; es decir, para cada dependencia funcional $X \to Y$, o bien X es una superllave, o Y es un atributo primo, es decir, Y es parte de alguna llave candidata.

La 3NF asegura que los datos están descompuestos de manera eficiente, eliminando las dependencias entre atributos no llave y minimizando así la duplicación de datos.

La **Forma Normal de Boyce-Codd (BCNF)** es una versión más estricta de la 3NF. Una relación está en BCNF si para cada una de sus dependencias funcionales no triviales $X \to Y$, X es una superllave. Esto implica que:

$$\text{Si } X \to Y \text{ en una relación } R, \text{ entonces } X \text{ debe ser una superllave de } R.$$

BCNF aborda algunas de las deficiencias de la 3NF al asegurar que cada determinante es una superllave, lo que elimina aún más las anomalías y aumenta la robustez del diseño de la base de datos.

Para demostrar la normalización de bases de datos hasta la Tercera Forma Normal (3NF) y la Forma Normal de Boyce-Codd (BCNF), consideraremos una tabla de ejemplo que incluye datos de empleados, sus departamentos y proyectos en los que trabajan.

Supongamos que tenemos la siguiente tabla:

Emp_ID	Emp_Name	Dept_Name	Dept_Head	Proj_Name
1	Alice	Finance	Bob	Budgeting
2	Bob	Finance	Bob	Forecast
3	Charlie	IT	Alice	Migration
4	Alice	IT	Alice	Cloud

Esta tabla está no normalizada porque incluye datos redundantes y no separa las entidades de manera eficiente.

Pasos para alcanzar 1NF: 1. Eliminar grupos repetitivos: Asegurarse de que haya solo valores atómicos. 2. Crear una llave primaria única para cada fila.

Pasos para alcanzar 2NF: 1. Estar en 1NF. 2. Eliminar dependencias parciales: atributos no llave deben depender solo de la llave primaria.

Pasos para alcanzar 3NF: 1. Estar en 2NF. 2. Eliminar dependencias transitivas: los atributos no llave no deben depender de otros atributos no llave.

Para pasar nuestra tabla a 3NF, se separarían los datos en tres tablas:

Tabla de Empleados

Emp_ID	Emp_Name	Dept_ID
1	Alice	1
2	Bob	1
3	Charlie	2
4	Alice	2

Tabla de Departamentos

Dept_ID	Dept_Name	Dept_Head
1	Finance	Bob
2	IT	Alice

Tabla de Proyectos

Emp_ID	Proj_Name

```
| 1          | Budgeting |
| 2          | Forecast  |
| 3          | Migration |
| 4          | Cloud     |
+------------+-----------+
```

Pasos para alcanzar BCNF: 1. Estar en 3NF. 2. Para cada dependencia funcional $X \to Y$, X debe ser una superllave.

En este caso, las tablas ya cumplen con BCNF porque cada determinante es una superllave en sus respectivas tablas.

Ejercicios propuestos

I Considera la siguiente tabla con datos de 'Empleado(ID, Nombre, Departamento, Jefe, Salario)'. Supón que el 'Jefe' es un atributo que depende del 'Departamento'. Convierte esta tabla a la Tercera Forma Normal.

II Dada una tabla 'Curso(Profesor, Asignatura, Texto, PrecioTexto)', donde 'Profesor' y 'Asignatura' son claves primarias y cada profesor enseña una única asignatura con un único texto requerido. El precio del texto depende solo del texto. Normaliza la tabla hasta BCNF.

III Tienes una tabla 'Paciente(ID, Nombre, FechaNacimiento, Doctor, EspecialidadDoctor)', donde 'Doctor' es el médico asignado al paciente y 'EspecialidadDoctor' depende directamente de 'Doctor'. Normaliza esta tabla hasta 3NF.

IV Considera una tabla 'Pedido(NumeroPedido, FechaPedido, Cliente, Producto, PrecioProducto)', donde 'PrecioProducto' depende únicamente de 'Producto'. Descompón la tabla para que cumpla con BCNF.

V Dada una tabla 'Libro(ISBN, Título, Autor, Editorial, CiudadEditorial)', donde 'Editorial' y 'CiudadEditorial' son dependientes. El ISBN es único para cada libro. Normaliza la tabla a 3NF.

VI Supón que tienes una tabla 'EmpleadoDepartamento(IDEmpleado, NombreEmpleado, IDDepartamento, NombreDepartamento, FunciónDepartamento)',

donde 'NombreDepartamento' y 'FunciónDepartamento' dependen de 'IDDepartamento'. Lleva esta tabla a BCNF.

VII Dada la tabla 'Investigación(Investigador, Proyecto, Rol, Financiamiento)', donde 'Financiamiento' depende del 'Proyecto' y no del investigador ni del rol que desempeña en el proyecto. Normaliza esta tabla hasta 3NF.

VIII Considera una tabla 'Factura(NumeroFactura, Fecha, Cliente, Producto, Cantidad, PrecioUnitario)'. Suponiendo que el 'PrecioUnitario' es siempre el mismo para el mismo 'Producto', normaliza la tabla hasta BCNF.

IX Tienes una tabla ‘Clase(CodigoClase, NombreProfesor, Horario, Salon)‘, donde ‘Horario‘ y ‘Salon‘ dependen de ‘CodigoClase‘. Convierte esta tabla a 3NF.

X Dada una tabla ‘Evento(CodigoEvento, Fecha, Ubicación, Organizador)‘, donde ‘Ubicación‘ depende de ‘Fecha‘ y ‘Evento‘. Normaliza la tabla hasta BCNF.

XI Considera la tabla ‘Venta(Tienda, Producto, Mes, Ventas, Precio)‘, donde ‘Precio‘ depende exclusivamente de ‘Producto‘. Descompón la tabla para que cumpla con 3NF.

XII Supón que tienes una tabla ‘Usuario(ID, Nombre, Ciudad, Estado, País)‘, donde ‘Estado‘ y ‘País‘ son dependientes de ‘Ciudad‘. Lleva esta tabla a BCNF.

Algunas soluciones

I La solución es el ejemplo.

II La tabla ya está en 1NF porque todos los atributos contienen solo valores atómicos. También está en 2NF porque no hay dependencias parciales de la clave primaria sobre los atributos no clave, dado que la clave primaria completa es necesaria para determinar los otros atributos.

- Curso(Profesor, Asignatura, Texto)
  - Clave Primaria: (Profesor, Asignatura)
  - Dependencias Funcionales: Profesor $\rightarrow$ Asignatura, Texto
- TextoLibro(Texto, PrecioTexto)
  - Clave Primaria: Texto
  - Dependencias Funcionales: Texto $\rightarrow$ PrecioTexto

III La tabla está en 1NF ya que todos los atributos son atómicos. También está en 2NF porque todos los atributos no clave son completamente dependientes de la clave primaria. Aquí, la clave primaria es ID, y no hay dependencias funcionales que violen la 2NF, dado que Doctor y EspecialidadDoctor no dependen de una parte de la clave primaria.

- **Tabla Paciente:** Paciente(ID, Nombre, FechaNacimiento, Doctor)
  - Clave Primaria: ID
- **Tabla Doctor:** Doctor(Doctor, EspecialidadDoctor)
  - Clave Primaria: Doctor
  - Dependencias Funcionales: Doctor $\rightarrow$ EspecialidadDoctor

IV
- **Tabla Pedido:** Pedido(NumeroPedido, FechaPedido, Cliente, Producto)

- Clave Primaria: (NumeroPedido, Producto)

- **Tabla Producto:** Producto(Producto, PrecioProducto)
  - Clave Primaria: Producto

Lógica en bases de datos

Una *vista* es una tabla virtual basada en el resultado de una consulta SQL. Las vistas se utilizan para simplificar consultas complejas, mejorar la seguridad y ocultar detalles de la estructura de datos subyacente.

```sql
CREATE VIEW VistaClientesImportantes AS
SELECT ID, Nombre, IngresosAnuales
FROM Clientes
WHERE IngresosAnuales > 100000;
```

Un *trigger* es un procedimiento que se ejecuta automáticamente en respuesta a ciertos eventos sobre tablas o vistas en la base de datos, como inserciones, actualizaciones o eliminaciones.

```sql
CREATE TRIGGER VerificarStock
AFTER INSERT ON Pedidos
FOR EACH ROW
BEGIN
    UPDATE Productos
    SET Stock = Stock - NEW.Cantidad
    WHERE ProductoID = NEW.ProductoID;
END;
```

Un *procedimiento almacenado* es un conjunto de instrucciones SQL que se compilan y almacenan en la base de datos. Se pueden ejecutar con una llamada desde una aplicación.

```sql
CREATE PROCEDURE AgregarCliente
    @Nombre VARCHAR(100),
    @Ciudad VARCHAR(100)
AS
BEGIN
    INSERT INTO Clientes (Nombre, Ciudad)
    VALUES (@Nombre, @Ciudad);
END;
```

Un *cursor* permite recorrer las filas de una tabla de forma controlada. Es útil para operaciones que requieren procesamiento de una fila a la vez.

```sql
DECLARE cursorCliente CURSOR FOR
SELECT ID, Nombre FROM Clientes;
OPEN cursorCliente;
FETCH NEXT FROM cursorCliente INTO @ID, @Nombre;
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @Nombre;
    FETCH NEXT FROM cursorCliente INTO @ID, @Nombre;
END;
CLOSE cursorCliente;
DEALLOCATE cursorCliente;
```

A continuación una lista de ejemplos de cursores en python:

I Conectar a una Base de Datos y Crear un Cursor

```
import sqlite3
conexion = sqlite3.connect('mi_base_de_datos.db')
cursor = conexion.cursor()

```

II Crear una Tabla

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS empleados (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    departamento TEXT NOT NULL,
    salario INTEGER NOT NULL
)
''')
conexion.commit()

```

III Insertar Datos Usando el Cursor

```
cursor.execute('''
INSERT INTO empleados (nombre, departamento, salario)
VALUES ('Ana P rez', 'Recursos Humanos', 45000)
''')
conexion.commit()

```

IV Consultar Datos

```
cursor.execute('SELECT * FROM empleados')
registros = cursor.fetchall()
for registro in registros:
    print(registro)

```

V Actualizar y Eliminar Datos

```
cursor.execute('''
UPDATE empleados SET salario = 48000 WHERE nombre = 'Ana P rez'
''')
conexion.commit()
```

```
cursor.execute('''
DELETE FROM empleados WHERE nombre = 'Ana P rez'
''')
conexion.commit()

```

VI Uso de Cursores en un Contexto

```
with sqlite3.connect('mi_base_de_datos.db') as conexion:
    with conexion.cursor() as cursor:
        cursor.execute('SELECT * FROM empleados')
        registros = cursor.fetchall()
        for registro in registros:
            print(registro)

```

VII Uso de Parámetros en las Consultas

```
nombre_empleado = 'Juan Mart nez'
cursor.execute('SELECT * FROM empleados WHERE nombre = ?',
(nombre_empleado,))
registro = cursor.fetchone()
print(registro)

```

El *ORM* es una técnica de programación que convierte datos entre sistemas incompatibles utilizando un lenguaje orientado a objetos. Se usa comúnmente en desarrollo de software para interactuar con bases de datos relacional.

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship

Base = declarative_base()

class Cliente(Base):
    __tablename__ = 'clientes'
    id = Column(Integer, primary_key=True)
    nombre = Column(String)
    pedidos = relationship("Pedido", back_populates="cliente")

class Pedido(Base):
    __tablename__ = 'pedidos'
    id = Column(Integer, primary_key=True)
    cliente_id = Column(Integer, ForeignKey('clientes.id'))
    cliente = relationship("Cliente", back_populates="pedidos")

# Crear una sesi n para realizar operaciones de base de datos
engine = create_engine('sqlite:///ejemplo.db')
Session = sessionmaker(bind=engine)
session = Session()

```

```python
# Agregar un nuevo cliente
nuevo_cliente = Cliente(nombre='Juan Perez')
session.add(nuevo_cliente)
session.commit()
```

Ejercicios propuestos

I Diseña una vista en SQL que muestre todos los clientes que han realizado compras superiores a $500 en el último año. Incluye el nombre del cliente y el total gastado.

II Dado el siguiente trigger en SQL, identifica cuál es su propósito y explica qué hace cuando se inserta un nuevo pedido en la base de datos:

```sql
CREATE TRIGGER ActualizarStock
AFTER INSERT ON Pedidos
FOR EACH ROW
BEGIN
    UPDATE Inventario
    SET cantidad = cantidad - NEW.cantidad
    WHERE producto_id = NEW.producto_id;
END;

```

III Crea un procedimiento almacenado que permita insertar un nuevo empleado en la tabla ‘Empleados‘, pasando como parámetros el nombre, departamento y salario.

IV Escribe un cursor en SQL que recorra todos los registros de la tabla ‘Productos‘ y actualice el precio en un 10 % para los productos cuyo stock sea inferior a 50 unidades.

V Implementa un ORM utilizando Python y SQLAlchemy para mapear una tabla ‘Libros‘ que incluya los campos ‘id‘, ‘titulo‘, ‘autor‘ y ‘precio‘.

VI Describe cómo podrías usar un trigger para validar que el email de un nuevo cliente cumple con un formato específico antes de insertarlo en la base de datos.

VII Proporciona un ejemplo de cómo crear una vista que combine datos de las tablas ‘Clientes‘ y ‘Pedidos‘, mostrando el total de pedidos por cliente.

VIII Crea un procedimiento almacenado que actualice el salario de un empleado por su ID, pasando el nuevo salario como parámetro.

IX Explica cómo un cursor podría ser utilizado para generar un reporte que liste todos los empleados y su salario, ordenando los resultados por departamento.

X Escribe un trigger que, al eliminar un cliente, también elimine todos sus pedidos asociados en la base de datos.

XI Desarrolla una aplicación sencilla en Python utilizando SQLAlchemy ORM para insertar, actualizar y eliminar datos en la tabla ‘Clientes‘.

XII Diseña un cursor que encuentre el producto más vendido cada mes y lo registre en una tabla de 'ProductosDestacados'.

XIII Formula un procedimiento almacenado que calcule el total de ventas de cada producto y lo almacene en una nueva tabla 'TotalVentasPorProducto'.

XIV Utilizando triggers, asegúrate de que no se puedan insertar valores nulos en las columnas críticas de la tabla 'Usuarios'.

XV Crea una vista que muestre la información de contacto de los clientes junto con el historial de todos sus pedidos en el último año.

ACID

El concepto de ACID se refiere a un conjunto de propiedades que garantizan la fiabilidad de las transacciones en un sistema de gestión de bases de datos. Estas propiedades son fundamentales para asegurar que las transacciones se realicen de manera segura y consistente. A continuación se describen cada una de las propiedades ACID:

I **Atomicidad (Atomicity):**

- La atomicidad garantiza que una transacción se trata como una unidad completa e indivisible. Esto significa que o bien todas las operaciones de la transacción se ejecutan con éxito, o si alguna falla, todas las operaciones se deshacen (*rollback*) y el estado de la base de datos vuelve al punto inicial antes de la transacción.

II **Consistencia (Consistency):**

- La consistencia asegura que cualquier transacción llevará la base de datos de un estado válido a otro estado válido. Esto significa que las restricciones de integridad (reglas definidas para mantener la calidad de los datos) no se violan antes o después de que una transacción se complete con éxito.

III **Aislamiento (Isolation):**

- El aislamiento asegura que el resultado de una transacción sea independiente y no afecte a otras transacciones que se están ejecutando concurrentemente en el sistema. Proporciona un nivel de protección contra los efectos secundarios de las transacciones concurrentes, garantizando que cada transacción vea una vista consistente de la base de datos.

IV **Durabilidad (Durability):**

- La durabilidad garantiza que una vez que una transacción se ha completado con éxito (es decir, se ha confirmado), los cambios realizados por esa transacción

persistirán incluso en caso de falla del sistema (por ejemplo, corte de energía o fallo de hardware). Los cambios se guardan permanentemente y no se pierden.

Una transacción se refiere a una secuencia de operaciones que se ejecutan como una sola unidad de trabajo. Estas operaciones pueden incluir inserciones, actualizaciones o eliminaciones de datos en la base de datos. Una transacción debe seguir las propiedades ACID para garantizar la integridad y consistencia de los datos.

I **Transferencia Bancaria:**

- **Operaciones:** Deducción del saldo de una cuenta y adición al saldo de otra cuenta.
- **Ejemplo:** Una transacción que transfiere $100 de la cuenta A a la cuenta B. Se realiza una deducción de $100 de la cuenta A y una adición de $100 a la cuenta B en una única operación atómica.
- **Propiedades ACID:** Garantiza atomicidad para evitar que solo una parte de la transacción se realice; consistencia para mantener la integridad de los saldos; aislamiento para evitar interferencias con otras transacciones; y durabilidad para asegurar que la transferencia sea persistente.

II **Compra en Línea:**

- **Operaciones:** Registro del pedido, actualización del inventario y procesamiento del pago.
- **Ejemplo:** Una transacción que registra un pedido en línea, reduce la cantidad disponible en el inventario y procesa el pago del cliente.
- **Propiedades ACID:** Debe ser atómica para evitar que una parte del pedido se complete sin las otras; consistente para mantener la integridad de los datos de inventario; aislada para evitar conflictos con otros pedidos; y durable para asegurar que el registro del pedido y la actualización del inventario persistan.

Una **schedule** (programación) se refiere a una secuencia de operaciones realizadas por múltiples transacciones de forma concurrente. El objetivo de la programación es controlar y coordinar la ejecución de estas transacciones para garantizar la consistencia y la integridad de la base de datos.

Características de un Schedule:

- Un schedule está compuesto por operaciones de múltiples transacciones que pueden ejecutarse de manera concurrente.
- El schedule define el orden en el cual las operaciones de las transacciones son ejecutadas.

- Un schedule puede ser serializable o no serializable, dependiendo de si su ejecución produce el mismo resultado que alguna secuencia serial (no concurrente) equivalente de las mismas transacciones.

Ejemplos de Schedules:

I **Schedule Serializable (Serializable):**

- Un schedule es serializable si su ejecución produce el mismo resultado que alguna secuencia serial equivalente de las mismas transacciones.
- **Ejemplo:** Considera dos transacciones T_1 y T_2 con las siguientes operaciones:

$$T_1 : \text{read}(X), \text{write}(Y), \text{commit}$$

$$T_2 : \text{write}(X), \text{read}(Y), \text{commit}$$

Un schedule serializable podría ser T_1 seguido por T_2 $(T_1; T_2)$ o T_2 seguido por T_1 $(T_2; T_1)$.

II **Schedule No Serializable (No Serializable):**

- Un schedule no serializable produce un resultado diferente que cualquier secuencia serial equivalente de las mismas transacciones.
- **Ejemplo:** Si tenemos dos transacciones T_1 y T_2 con operaciones concurrentes como read(X) y write(X) en diferentes órdenes dentro del schedule, esto podría llevar a resultados no deseados o inconsistencias en la base de datos.

Un **grafo de precedencia** es una representación gráfica que muestra las dependencias de orden entre las operaciones de diferentes transacciones. Estos grafos son útiles para analizar la concurrencia y la serializabilidad de las transacciones en un sistema de base de datos.

- Cada transacción se representa como un conjunto de operaciones (lecturas o escrituras) sobre datos en la base de datos.
- Se establecen arcos dirigidos entre las operaciones para mostrar la dependencia de orden entre ellas.
- Las operaciones de escritura (write) de una transacción deben preceder a cualquier operación de lectura (read) de otra transacción si ambas operaciones afectan al mismo dato.

Considera dos transacciones T_1 y T_2 con las siguientes operaciones sobre datos X y Y:

$$T_1 : \text{write}(X), \text{read}(Y), \text{commit}$$

$$T_2 : \text{read}(X), \text{write}(Y), \text{commit}$$

En este grafo de precedencia:

- Las operaciones de escritura (write) preceden a las operaciones de lectura (read) sobre el mismo dato (por ejemplo, write$(X)_{T1} \rightarrow$ read$(X)_{T2}$). - Las operaciones de cada transacción siguen un orden secuencial hasta el commit.

Este grafo muestra la dependencia de orden entre las operaciones de las transacciones T_1 y T_2, lo que facilita el análisis de la concurrencia y la serializabilidad.

Recuperaciones de fallas

La recuperación de fallas en bases de datos es esencial para mantener la integridad y disponibilidad de los datos frente a errores, fallos de hardware, desastres naturales y otros eventos imprevistos. Este proceso asegura que las transacciones se gestionen de manera que cumplan con las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad).

Datos Erróneos

- **Restricciones de Integridad:** Implementar restricciones a nivel de base de datos como claves primarias y foráneas, restricciones de unicidad y verificación.
- **Limpieza de Datos:** Procesos para detectar y corregir registros erróneos o incompletos.

Fallas en el Almacenamiento

- **RAID (Redundant Array of Independent Disks):** Configuraciones de RAID protegen los datos mediante duplicación y paridad.
- **Copias Redundantes:** Realizar copias de seguridad regulares en dispositivos separados.

Fallas del Sistema

- **Log y Recovery Manager:** Gestionan los logs de transacciones para recuperación.

Técnicas de Logging

Undo Logging

- **Reglas:**
    - Todos los logs <T, X, t> deben ser escritos antes que el valor X sea escrito en disco.

- <COMMIT T> debe ser escrito después de que todos los datos modificados estén en disco.

- **Recuperación:** Deshacer cambios de transacciones no confirmadas (sin COMMIT) en caso de falla.

Redo Logging

- **Reglas:**
  - Todos los logs deben ser escritos en disco antes de modificar cualquier elemento X en disco, incluyendo el <COMMIT T>.
- **Recuperación:** Rehacer cambios de transacciones confirmadas (con COMMIT) que no se han reflejado en el disco.

Undo/Redo Logging

- **Logs:** <T, X, vantiguo, vnuevo> antes de modificar X; <COMMIT T> y <T,END> según sea necesario.
- **Recuperación:** Combina ambas técnicas para deshacer transacciones no confirmadas y rehacer las confirmadas.

Checkpoints

Checkpoints Tradicionales

- Detener transacciones, guardar el estado en disco y reanudar transacciones. Requiere .apagar.el sistema temporalmente.

Nonquiescent Checkpoints

- Escribir logs <START CKPT (T1, ..., Tn)> y <END CKPT> sin detener el sistema. Permite continuar operaciones mientras se asegura la integridad.

Algoritmo de Recuperación

Undo Logging

- Procesar el log desde el final hacia el inicio.
- Restituir valores antiguos de transacciones no confirmadas.
- Ignorar transacciones confirmadas.

Redo Logging

- Procesar el log desde el inicio hasta el final.
- Reescribir valores de transacciones confirmadas.
- Abortar transacciones sin confirmación.

En resumen, la recuperación de fallas en bases de datos se basa en la correcta implementación de técnicas de logging (Undo, Redo y Undo/Redo) y el uso eficiente de checkpoints para garantizar la integridad y durabilidad de los datos tras cualquier fallo.

Ejercicios propuestos

I Explica la diferencia entre **atomicidad débil** y **atomicidad fuerte** en el contexto de transacciones de bases de datos. ¿En qué escenarios preferirías una sobre la otra?

II Considera una transacción que involucra múltiples operaciones de escritura. Describe cómo podrías implementar un mecanismo de **punto de recuperación** (*savepoint*) para permitir deshacer parte de la transacción en caso de un error.

III Investigación sobre **invariantes de la base de datos** y cómo se relacionan con la propiedad de consistencia. Proporciona un ejemplo concreto de un invariante de base de datos y explica cómo se puede mantener durante una transacción.

IV Discute la diferencia entre la consistencia a nivel de aplicación y la consistencia a nivel de base de datos. ¿Por qué es importante mantener ambas en un sistema de bases de datos transaccionales?

V Explora el fenómeno de **lecturas sucias** (*dirty reads*), **lecturas no repetibles** (*non-repeatable reads*) y **lecturas fantasma** (*phantom reads*). Describe cómo diferentes niveles de aislamiento pueden prevenir cada uno de estos problemas.

VI ¿Qué es la **serialización de transacciones**? Explica por qué puede ser necesario utilizar técnicas de serialización para mantener el aislamiento en un entorno de base de datos multiusuario.

VII Investiga sobre **registros de cambio** (*redo logs*) y **puntos de comprobación** (*checkpoints*) en sistemas de bases de datos. Describe cómo estos mecanismos contribuyen a la durabilidad de las transacciones.

VIII Discute las implicaciones de rendimiento de asegurar la durabilidad en una base de datos. ¿Qué estrategias podrían implementarse para mejorar la durabilidad sin comprometer significativamente el rendimiento?

IX Explique cómo se gestionan las fallas de transacciones en una base de datos utilizando el Undo Logging. Incluya en su respuesta las reglas de Undo Logging y un ejemplo de recuperación de una transacción abortada.

X Describa las principales diferencias entre los niveles RAID 0, RAID 1 y RAID 5, y explique cómo cada uno de estos niveles afecta el rendimiento y la fiabilidad del sistema de almacenamiento.

XI Analice el proceso de recuperación con Redo Logging. ¿Cómo se utilizan los checkpoints en este método y cuál es el beneficio de usar Nonquiescent Checkpoints en lugar de checkpoints tradicionales?

Almacenamiento e indices

La gestión eficiente de datos en sistemas de bases de datos implica minimizar los tiempos de acceso a la memoria secundaria (disco duro) y optimizar las consultas mediante el uso de índices y estructuras de datos adecuadas. A continuación se resumen los conceptos principales:

Memoria Principal vs. Memoria Secundaria

- **Memoria Principal (RAM):** Accesos rápidos pero limitados en capacidad.
- **Memoria Secundaria (Disco Duro):** Accesos lentos pero mayor capacidad.

Modelo de Costos

- **Costo I/O:** Medido en términos de accesos a memoria secundaria.
- **Minimización de I/O:** Fundamental para mejorar el rendimiento de las consultas.

Índices

- **Objetivo:** Optimizar el acceso a datos específicos o rangos de datos.
- **Tipos de Consultas:**
  - Consultas por valor: `SELECT * FROM Table WHERE Table.value = 'value'`
  - Consultas por rango: `SELECT * FROM Table WHERE Table.value >= 'value'`

Estructuras de Datos

- **Arreglos:** Secuencia contigua de celdas de tamaño fijo.
- **Listas Ligadas:** Secuencia de celdas enlazadas con tamaño variable.
- **Diccionarios:** Estructuras para asociación eficiente de llaves y valores.

Tablas de Hash

- **Función de Hash:** Mapea llaves a índices de manera eficiente.
- **Resolución de Colisiones:**
  - **Encadenamiento:** Utiliza listas ligadas para manejar colisiones.
  - **Direccionamiento Abierto:** Busca celdas vacías en caso de colisión.

Índices Hash y Árbol B+

- **Índices Hash:** Eficientes para búsquedas exactas pero ineficientes para búsquedas de rango.
- **Árbol B+:** Eficientes para búsquedas exactas y de rango, con tiempo de búsqueda logarítmico.

Clustered vs. Unclustered Index

- **Clustered Index:** Los datos están ordenados físicamente en el disco según el índice.
- **Unclustered Index:** El índice contiene punteros a las ubicaciones de los datos en el disco.

B+ Tree Index

- **Estructura:** Nodos del árbol representan páginas del disco, con tuplas en las hojas.
- **Consultas:** Permite búsquedas eficientes de valor y rango.
- **Ejemplo de Consulta de Rango:** Buscar artistas con `aid` entre 5 y 26.

Ejercicios propuestos

I Explica la diferencia entre memoria principal y memoria secundaria. ¿Por qué es importante minimizar el número de accesos a memoria secundaria en un DBMS?

II Describe el modelo de costos I/O y su importancia en la optimización de consultas en bases de datos.

III Dado el siguiente arreglo y una lista ligada, implementa las operaciones básicas de inserción, búsqueda y eliminación.

IV Implementa una función de hash y resuelve las colisiones usando encadenamiento.

V Explica el proceso de inserción y búsqueda en una tabla de hash con sondeo lineal. Proporciona ejemplos.

VI Compara y contrasta los índices clustered y unclustered. Proporciona ejemplos de cuándo usar cada uno.

VII Describe la estructura de un árbol B+ y cómo facilita consultas de rango.

VIII Implementa un índice hash para una tabla de artistas. Describe cómo manejarías colisiones y búsquedas.

IX Explica cómo los índices hash y los árboles B+ manejan las consultas de igualdad y de rango.

X Realiza una búsqueda de rango en un árbol B+ con los siguientes datos: {(1, 'Leonardo'), (5, 'Bernini'), (8, 'Michelangelo'), (11, 'Brunelleschi'), (12, 'Botticelli')}. Encuentra todos los artistas con `aid` entre 5 y 12.