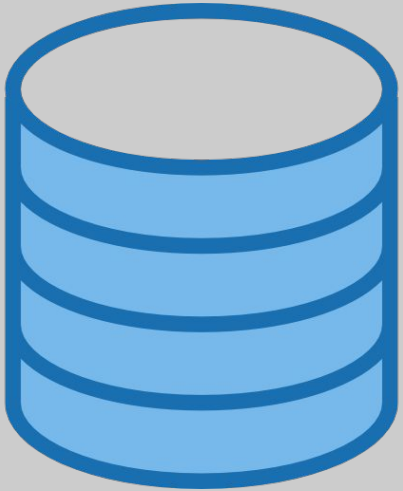
Tecnicatura en Software (a distancia)

Administración de Base de Datos



Conceptos iniciales

Introducción.

- ¿Qué es un dato?
- ¿Qué es información?
- ¿Qué es una base de datos?
- Sistema de información
- Sistemas de gestión de base de datos
- Base de Datos y SGBD
- Aplicaciones de los Sistemas de Bases de Datos
- Conceptos clave de base de datos
- Niveles de abstracción

¿Qué es un dato?

Un dato es una representación simbólica de una realidad o hecho del mundo real.
Por sí solo, un dato no tiene un significado específico.

Por ejemplo, "1911" podría ser un año, un código de una materia en la facultad, o el número de teléfono de un servicio de radiotaxi Es solo cuando el dato se procesa o se contextualiza, que se convierte en información.

Ejemplos:

¿Qué es información?

La información es el resultado del procesamiento y la interpretación de los datos.
Cuando un dato es procesado y comunicado de manera que pueda ser entendido e interpretado, se convierte en información.

Por ejemplo, si "1911" se contextualiza como el año en que se inauguró una universidad, se convierte en una información relevante.

Ejemplo:

Pedro tiene 23 Rosas Rojas

¿Qué es una base de datos?

Una base de datos es un conjunto organizado de datos interrelacionados que se almacenan sin redundancias perjudiciales o innecesarias, garantizando su consistencia, integridad y seguridad.
Una base de datos tiene como objetivo servir a una o varias aplicaciones de la manera más eficiente posible.

Información y Sistema de información

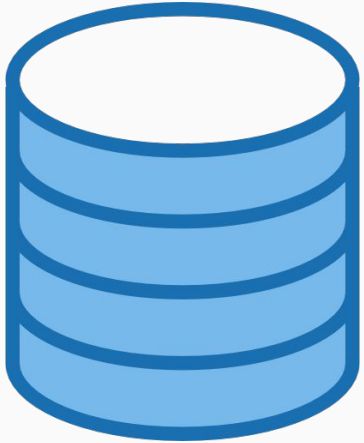
La información es el conocimiento derivado del análisis o tratamiento de los datos que se utiliza para tomar decisiones con vistas a un accionar concreto.

Un sistema de información es una colección de datos debidamente recopilados y estructurados que proporcionan información sobre la realidad.

En informática, ayudan a administrar, recolectar, recuperar, procesar, almacenar y distribuir información relevante para los procesos fundamentales de cada organización. Se destacan por su diseño, facilidad de uso, flexibilidad, mantenimiento automático, apoyo en toma de decisiones, y el anonimato y la privacidad de los datos.

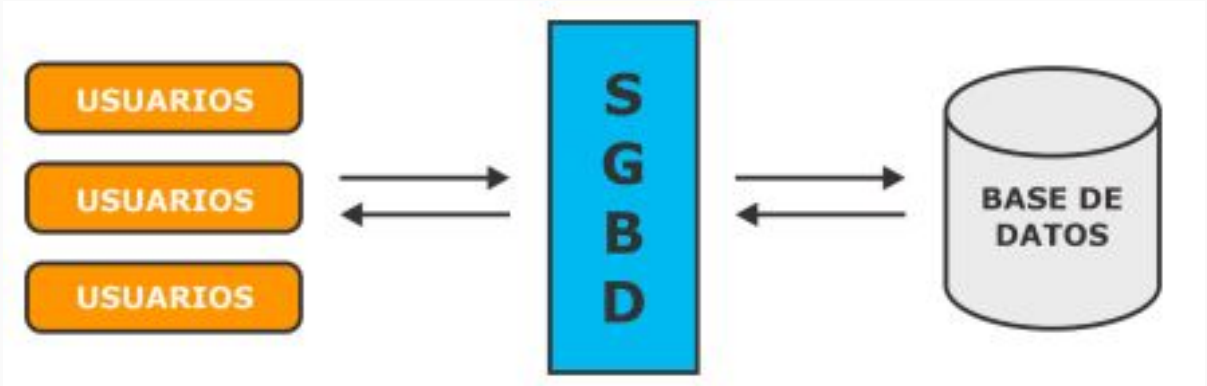
Sistemas de gestión de base de datos

Un **Sistema de Gestión de Bases de Datos (SGBD)** es un software que permite a los usuarios crear, mantener, administrar y manejar bases de datos. El SGBD proporciona herramientas para insertar, actualizar, eliminar y consultar datos de manera eficiente y segura.



Ejemplo de SGBD:

- MySQL, PostgreSQL, Oracle, MongoDB.



Base de Datos y SGBD

Una Base de Datos (BD) es un conjunto de datos relacionados entre sí. No se debe confundir con el Sistema Gestor de Base de Datos (SGBD) que proporciona una forma de almacenar y recuperar la información de una base de datos de manera práctica y eficiente.

Un SGBD debe cumplir con el paradigma ACID, que garantiza que las transacciones en las bases de datos se realicen de forma confiable:

- **Atomicidad:** Todas las operaciones de una transacción deben completarse con éxito; de lo contrario, ninguna operación debe aplicarse.
- **Consistencia:** La base de datos debe pasar de un estado válido a otro estado válido después de cada transacción.
- **Aislamiento:** Las transacciones concurrentes no deben interferir entre sí, manteniendo la integridad de los datos.
- **Durabilidad:** Una vez que una transacción se ha completado, los cambios realizados deben persistir incluso si ocurre un fallo del sistema.

Aplicaciones de los SGBD y Conceptos clave de base de datos

Las **aplicaciones de los sistemas de bases de datos** son variadas e incluyen cualquier software que necesite almacenar, gestionar y recuperar grandes volúmenes de datos. Ejemplos incluyen sistemas bancarios, sistemas de gestión de inventario, sistemas de reservas, y redes sociales.

Algunos **conceptos clave de base de datos** incluyen:

- **Tabla**: La estructura básica para almacenar datos en una base de datos.
- **Registro**: Una fila en una tabla, que representa una entrada completa en la base de datos.
- **Campo**: Una columna en una tabla, que representa un tipo específico de dato.
- **Clave primaria**: Un campo o combinación de campos que identifica de manera única cada registro en una tabla.

Niveles de abstracción

Para simplificar la interacción con las bases de datos, los desarrolladores utilizan tres niveles de abstracción:

1. **Nivel Interno o Físico:** Describe cómo se almacenan realmente los datos en la base de datos, incluyendo detalles sobre las estructuras de datos.
   Ejemplo: Detalles técnicos sobre cómo los datos de los empleados se almacenan en discos duros.
2. **Nivel Conceptual o Lógico:** Describe qué datos se almacenan y las relaciones entre ellos.
   Ejemplo: Un modelo que muestra cómo los empleados están relacionados con los departamentos dentro de una empresa.
3. **Nivel Externo o de Visión:** Proporciona una vista personalizada de la base de datos para cada usuario o aplicación.
   Ejemplo: Un sistema de gestión de recursos humanos podría mostrar solo la información relevante para un gerente, como los informes de desempeño de los empleados.

Niveles de abstracción

- Niveles de abstracción
- Uso de los modelos
- Niveles de abstracción
- Modelos de datos
  - Modelos de datos conceptuales
  - Modelos de datos físicos
  - Modelos de datos lógicos
- Ventajas y desventajas de los modelos de datos

Niveles de abstracción

Los **niveles de abstracción** en bases de datos permiten manejar la complejidad de las estructuras de datos. Estos niveles se utilizan para separar las preocupaciones de los usuarios y desarrolladores de la base de datos en diferentes capas.

1.2 Nivel Físico:

- **Definición:** El nivel más bajo de abstracción, que describe cómo se almacenan realmente los datos en el hardware.
- **Ejemplos:** Organización de archivos, índices, acceso secuencial o directo.
- **Propósito:** Optimizar el almacenamiento y el rendimiento de la base de datos.

1.2 Nivel Lógico (o Conceptual):

- **Definición:** Describe la estructura lógica de la base de datos, incluyendo las relaciones entre los diferentes datos, sin considerar cómo se almacenan físicamente.
- **Ejemplos:** Tablas, relaciones, restricciones.
- **Propósito:** Definir qué datos se almacenan y cómo se relacionan entre sí, sin preocuparse por su implementación física.

1.3 Nivel Externo (o de Vista):

- **Definición:** Es el nivel más cercano a los usuarios finales, que define cómo los datos son vistos e interactuados por diferentes usuarios o aplicaciones.
- **Ejemplos:** Vistas personalizadas de la base de datos para diferentes departamentos en una empresa.
- **Propósito:** Proporcionar diferentes perspectivas de la misma base de datos según las necesidades del usuario.

Uso de los Modelos

Los **modelos de datos** son herramientas que se utilizan para representar los niveles de abstracción en una base de datos. Cada modelo de datos es una representación abstracta de los datos y las relaciones entre ellos.

Objetivo de los Modelos:

- **Claridad:** Simplificar la comunicación entre desarrolladores, administradores y usuarios finales.
- **Organización:** Estructurar la información de manera que sea fácil de entender y mantener.
- **Eficiencia:** Mejorar la eficiencia en el almacenamiento y recuperación de datos.

● Modelos de Datos

Un **modelo de datos** es una representación abstracta que organiza los elementos de datos y estandariza cómo se relacionan entre sí y con las propiedades del mundo real.

Tipos de Modelos de Datos:

- **Conceptuales**
- **Lógicos**
- **Físicos**

Modelos de Datos Conceptuales

Los **modelos de datos conceptuales** proporcionan una visión general de alto nivel de lo que contiene la base de datos, incluyendo entidades, relaciones y restricciones, sin entrar en detalles sobre cómo se implementan o almacenan.

Características:

- **Abstracción:** Se enfoca en qué datos se van a almacenar, sin preocuparse por cómo se almacenarán.
- **Entidades y Relaciones:** Identificación de las entidades (p. ej., "Clientes", "Productos") y cómo se relacionan entre sí.

Ejemplos:

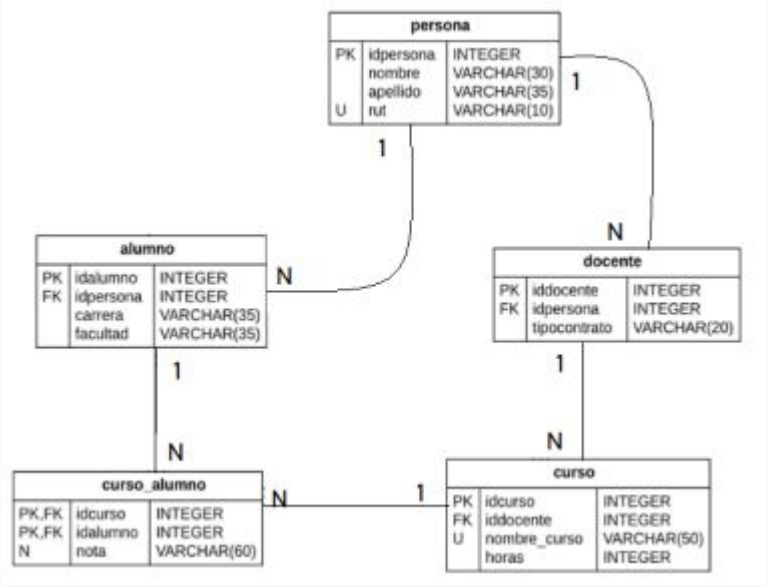
- **Diagrama Entidad-Relación (ER):** Una técnica común para representar un modelo de datos conceptual.

Modelos de Datos Físicos

El **modelo de datos físicos** es la representación más cercana a la manera en que los datos se almacenan realmente en el hardware. Detalla cómo se implementarán los datos dentro de la base de datos.

Características:

- **Esquema Físico:** Incluye detalles como el tipo de datos, el tamaño del campo, las ubicaciones de almacenamiento, índices, etc.
- **Optimización:** Se enfoca en el rendimiento, asegurando que el acceso y almacenamiento de datos sean eficientes.



Ejemplos:

- **Tablas e Índices:** Estructuras físicas que se crean en la base de datos para almacenar datos y mejorar la velocidad de las consultas.

Modelos de Datos Lógicos

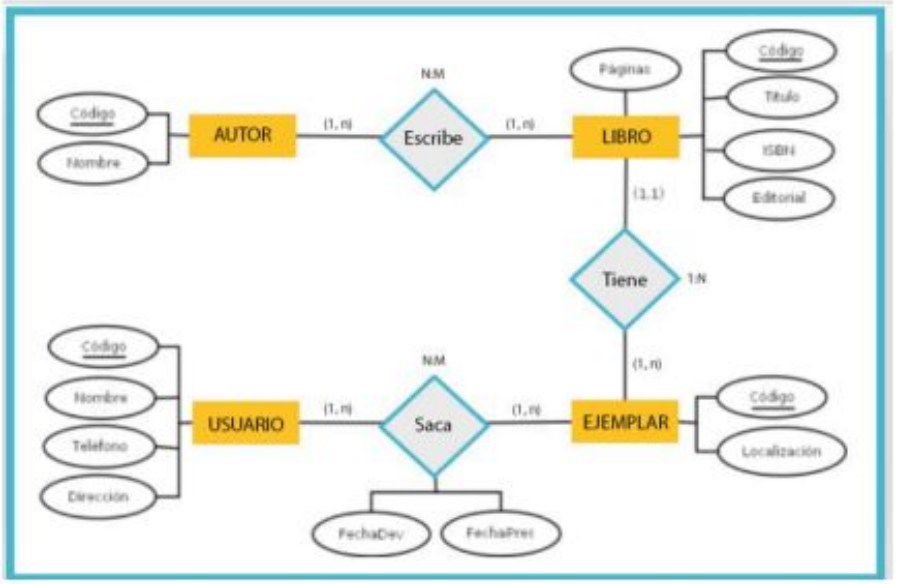
El **modelo de datos lógicos** actúa como un puente entre el modelo conceptual y el modelo físico. Describe la estructura de la base de datos sin especificar detalles físicos.

Características:

- **Independencia Física:** No depende de la tecnología de almacenamiento o la plataforma de base de datos.
- **Relaciones y Atributos:** Detalla las relaciones, claves primarias y foráneas, atributos, etc.

Ejemplos:

- **Esquema Relacional:** Una representación lógica de la base de datos con tablas, columnas y relaciones.



Ventajas y Desventajas de los Modelos de Datos

Ventajas:

- **Claridad:** Facilitan la comprensión de la estructura de la base de datos.
- **Documentación:** Proporcionan una base sólida para la documentación y el mantenimiento.
- **Diseño Eficiente:** Ayudan a diseñar bases de datos que sean eficientes y fáciles de gestionar.

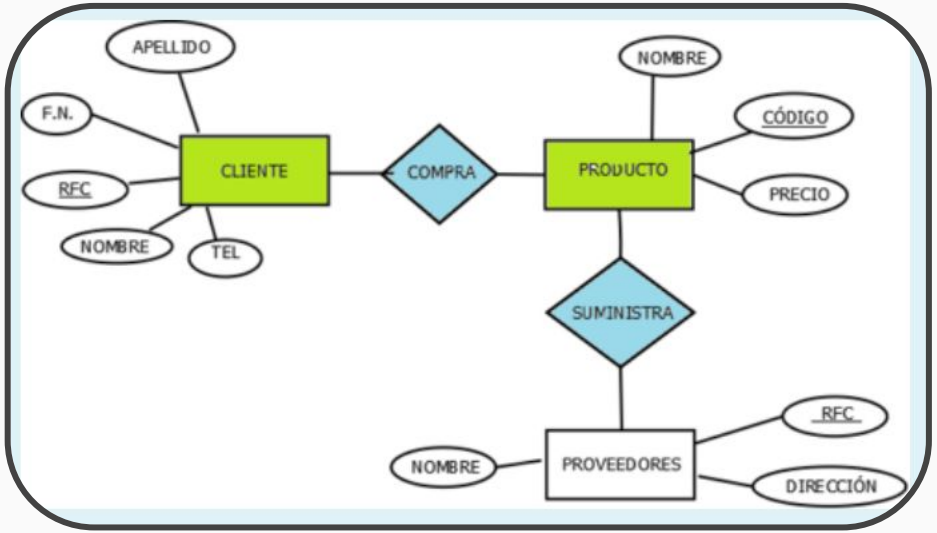
Desventajas:

- **Complejidad:** Pueden volverse complicados, especialmente cuando los sistemas son grandes o tienen muchas interrelaciones.
- **Rigidez:** Algunos modelos, como los físicos, pueden ser difíciles de modificar una vez implementados.
- **Tiempo de Desarrollo:** Crear modelos detallados puede ser un proceso largo que requiere mucho tiempo y recursos.

- Modelo entidad-relación.
- Elementos del modelo entidad-relación: entidad, relación, atributo, cardinalidad de una relación.
- Relaciones.
- Propiedades de la relación.
- Grado de una relación.
- Conectividad de una relación.
- Condicionalidad de una relación.
- Base de Datos y SGBD.
- Normalización:
    - Primera forma normal.
    - Segunda forma normal.
    - Tercera forma normal.

Modelo entidad-relación (ER)

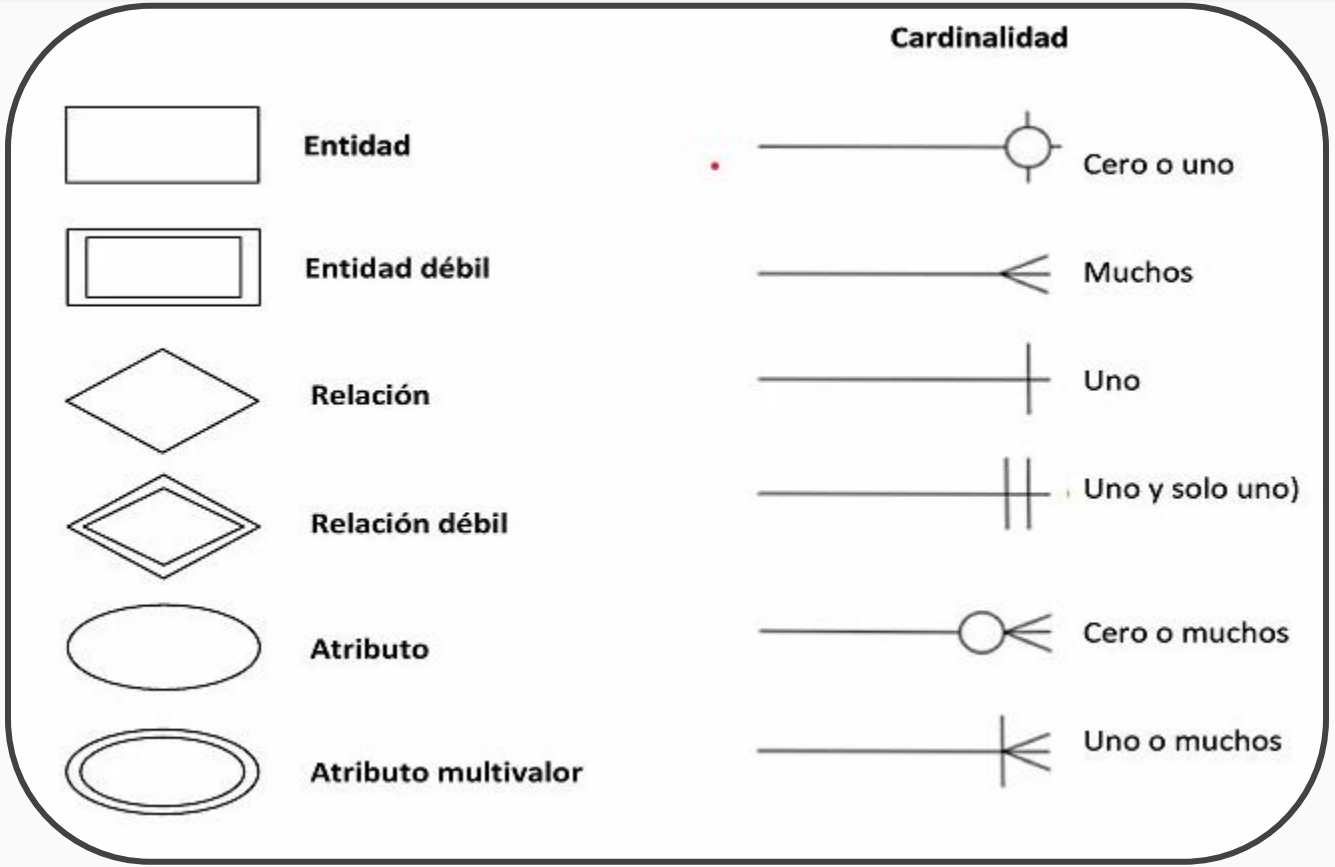
Un **Modelo Entidad-Relación (ER)** es una representación gráfica que ilustra las entidades (objetos), sus atributos y las relaciones entre ellas en una base de datos. Es una herramienta utilizada en el diseño de bases de datos.



Ejemplo:
Supongamos que estamos diseñando una base de datos para una escuela.

- **Entidad:** Estudiante, Curso, Profesor
- **Relación:** Inscripción (relación entre Estudiante y Curso)
- **Atributo:** Nombre del Estudiante, Código del Curso, Nombre del Profesor

Anotaciones ERD



Elementos del modelo entidad-relación: entidad, relación, atributo, cardinalidad de una relación.

1. Entidad

Una **entidad** es un objeto o cosa del mundo real que tiene existencia propia y es distinguible de otros objetos. Las entidades son importantes porque se almacenan en la base de datos con sus atributos específicos. Cada entidad se representa en un diagrama entidad-relación (ER) como un rectángulo.

- **Ejemplo:**
  La entidad "Libro" en una biblioteca. Cada libro tiene atributos como ISBN, título, año de publicación, editorial, etc.

Dentro de las entidades existen dos tipos principales:

- **Entidades fuertes:** No dependen de otras entidades para existir.
  - **Ejemplo:** "Libro" es una entidad fuerte porque existe independientemente de otras entidades.
- **Entidades débiles:** Dependen de otra entidad para existir.
  - **Ejemplo:** "Capítulo" puede ser una entidad débil si depende de la entidad "Libro", ya que un capítulo no tiene sentido sin un libro.

2. Relación

Una **relación** es un vínculo entre dos o más entidades que establece cómo se asocian entre sí. En un diagrama ER, las relaciones se representan mediante un rombo.

- **Ejemplo:**
  La relación "Escrito por" entre las entidades "Libro" y "Autor". Esto indica que un autor puede escribir varios libros, y un libro puede ser escrito por varios autores.

El grado de una relación se refiere al número de entidades que participan en ella:

- **Grado 1:** Relación que asocia una entidad consigo misma.
  - **Ejemplo:** Un empleado puede supervisar a otros empleados.
- **Grado 2:** Relación que asocia dos entidades distintas.
  - **Ejemplo:** Un estudiante se inscribe en un curso.
- **Grado n:** Relación que asocia más de dos entidades distintas.
  - **Ejemplo:** Un proyecto puede involucrar varios empleados, proveedores y recursos.

3. Atributo

Un **atributo** es una característica o propiedad que describe una entidad o relación. Cada atributo tiene un dominio, que es el conjunto de valores válidos que puede tomar.

- **Ejemplo:**
  La entidad "Libro" tiene atributos como "Título", "Año de Publicación", y "Editorial". Estos atributos se representan como óvalos conectados a la entidad en el diagrama ER.

Tipos de atributos:

- **Identificador único o clave primaria:** Es un atributo o conjunto de atributos que identifica de manera única a cada entidad.
  - **Ejemplo:** El ISBN es un identificador único para la entidad "Libro".
- **Atributos derivables:** Son aquellos que se pueden calcular a partir de otros atributos.
  - **Ejemplo:** La "Edad" de un empleado se puede calcular a partir de su "Fecha de Nacimiento".
- **Atributos compuestos:** Son aquellos que están formados por otros subatributos.
  - **Ejemplo:** El atributo "Dirección" podría descomponerse en "Calle", "Número", "Ciudad", "Código Postal", etc.

4. Cardinalidad de una relación

La **cardinalidad** de una relación describe el número de instancias de una entidad que pueden asociarse con una instancia de otra entidad. Hay tres tipos principales de cardinalidad:

- **Uno a uno (1:1):**
  Cada instancia de una entidad se relaciona con una y solo una instancia de otra entidad.
  - **Ejemplo:** Un estudiante tiene una única tarjeta de identificación.
- **Uno a muchos (1:M):**
  Una instancia de una entidad se relaciona con muchas instancias de otra entidad.
  - **Ejemplo:** Un profesor puede dar muchas clases, pero una clase solo tiene un profesor.
- **Muchos a muchos (M:M):**
  Las instancias de dos entidades pueden estar relacionadas entre sí en múltiples formas.
  - **Ejemplo:** Un estudiante puede estar inscrito en varios cursos, y un curso puede tener varios estudiantes inscritos.

Relaciones.

Una **Relación** es la asociación entre dos o más entidades en un modelo ER.

Ejemplo:
En una base de datos de una biblioteca, la relación entre las entidades "Libro" y "Autor" podría ser "Escrito por". Cada libro tiene uno o más autores, y un autor puede haber escrito uno o más libros.

Grado de una relación

El **grado de una relación** indica el número de entidades que participan en ella. Las relaciones pueden ser:

- **Binarias:** Relacionan dos entidades.
- **Ternarias:** Relacionan tres entidades.
- **n-arias:** Relacionan n entidades.
- **Ejemplo:**
  Una relación ternaria podría ser "Proveedor - Suministra - Producto a - Almacén", donde tres entidades están relacionadas.

Propiedades de la relación

Las **propiedades de la relación** incluyen su nombre, los atributos que la describen, y su cardinalidad. Estas propiedades ayudan a entender mejor la relación entre las entidades.

- **Ejemplo:**
  En la relación "Compra" entre "Cliente" y "Producto", las propiedades pueden incluir la "Fecha de Compra" y "Cantidad Comprada".

Conectividad de una relación

La **conectividad de una relación** describe cómo las entidades están asociadas, en términos de uno a uno, uno a muchos, o muchos a muchos.

- **Ejemplo:**
  La relación "Enseña" entre "Profesor" y "Curso" podría ser uno a muchos (1:M), donde un profesor enseña muchos cursos, pero cada curso tiene solo un profesor.

Base de Datos y SGBD

Base de Datos

Una **Base de Datos** es una colección organizada de datos estructurados, generalmente almacenados y accesibles electrónicamente desde un sistema informático. Las bases de datos permiten la gestión eficiente de grandes volúmenes de información.

- **Ejemplo:**
  Una base de datos de una tienda en línea podría contener tablas para productos, clientes, pedidos y proveedores.

Sistema de Gestión de Bases de Datos (SGBD)

Un **SGBD** es un software que permite crear, gestionar y manipular bases de datos de manera eficiente. Los SGBD proporcionan herramientas para el control de acceso, la integridad de los datos, la concurrencia y la recuperación ante fallos.

- **Ejemplo:**
  MySQL, PostgreSQL, Oracle Database, y Microsoft SQL Server son ejemplos de SGBD.

Normalización

Primera forma normal (1NF):

- **Requisitos:** Asegurar que todos los valores en una tabla sean atómicos (no divisibles) y eliminar los grupos repetitivos.
- **Ejemplo:** Convertir una tabla que almacena múltiples números de teléfono en un solo campo a una tabla donde cada número de teléfono tiene su propio registro.

Segunda forma normal (2NF):

- **Requisitos:** Cumplir con la 1NF y asegurarse de que no haya dependencias parciales (atributos que dependen solo de una parte de la clave primaria).
- **Ejemplo:** Separar los datos de los clientes de los datos de los pedidos en tablas diferentes, ya que los detalles del cliente no dependen del pedido específico.

Tercera forma normal (3NF):

- **Requisitos:** Cumplir con la 2NF y eliminar dependencias transitivas (cuando un atributo depende de otro atributo no clave).
- **Ejemplo:** Crear una tabla separada para almacenar la ciudad del cliente en lugar de incluirla en la tabla de pedidos, ya que la ciudad depende del cliente, no del pedido.

Biografía

- **Modelo y Normalización de Bases de Datos**
  https://gbbdd.abrilcode.com/doku.php?id=bloque2:diseno
- **Modelamiento de Datos**
  https://bookdown.org/paranedagarcia/database/modelamiento-de-datos.html

- **Operaciones de álgebra relacional**
  - **Introducción**
  - **Operaciones de selección**
  - **Operaciones de proyección**
  - **Operaciones de unión**
  - **Operaciones de intersección**
  - **Operaciones de diferencia**
  - **Operaciones de producto cartesiano**
  - **Operaciones de producto cartesiano natural**
- **SQL**
  - **Conceptos**
  - **Sintaxis**
    - **InnoDB**
    - **DML**
  - **Álgebra relacional migrar a SQL**
- **Integridad y restricciones**
  - **Integridad de atributo**
  - **Integridad de filas**
  - **Integridad de claves**

Operaciones de álgebra relacional

El álgebra relacional es un conjunto de operaciones que permiten manipular y consultar datos en una base de datos relacional. Estas operaciones permiten obtener nuevas relaciones a partir de otras, transformando los datos y extrayendo la información deseada.

1. Operaciones de Selección (σ)

La selección permite extraer tuplas que cumplen con una condición determinada. Es una operación unaria, ya que se aplica sobre una única relación. Se utiliza para filtrar datos específicos.

- **Ejemplo:** Obtener todos los empleados cuyo salario es mayor a $2000.

2. Operaciones de Proyección (π)

La proyección selecciona columnas específicas de una relación, eliminando atributos innecesarios. También elimina las tuplas duplicadas.

- **Ejemplo:** Obtener solo los nombres y apellidos de todos los empleados.

3. Operaciones de Unión (∪)

La unión combina las tuplas de dos relaciones que tienen la misma aridad (mismo número de columnas) y tipos de datos compatibles. Las tuplas duplicadas son eliminadas.

- **Ejemplo:** Obtener una lista única de todos los clientes y empleados.

Operaciones de álgebra relacional

4. Operaciones de Intersección (∩)

La intersección obtiene las tuplas que están presentes en ambas relaciones. Las dos relaciones deben tener el mismo número de columnas y tipos de datos.

- **Ejemplo:** Obtener una lista de personas que son tanto empleados como clientes.

5. Operaciones de Diferencia (−)

La diferencia devuelve las tuplas que están en la primera relación pero no en la segunda.

- **Ejemplo:** Obtener todos los empleados que no son clientes.

6. Operaciones de Producto Cartesiano (×)

El producto cartesiano combina cada tupla de una relación con cada tupla de otra relación. El resultado es el conjunto de todas las combinaciones posibles.

- **Ejemplo:** Listar todas las combinaciones posibles de productos y proveedores.

7. Operaciones de Producto Cartesiano Natural (⋈)

Es una variante del producto cartesiano, pero elimina duplicados de las columnas que coinciden en ambas relaciones.

- **Ejemplo:** Combinar la información de empleados y departamentos en base al ID del departamento.

SQL

SQL (Structured Query Language) es el lenguaje estándar utilizado para interactuar con bases de datos relacionales. Permite realizar operaciones como insertar, actualizar, eliminar y consultar datos.

1. Conceptos

SQL está basado en el modelo relacional y permite gestionar datos mediante tablas que se relacionan entre sí. Utiliza comandos estructurados para interactuar con las bases de datos.

2. Sintaxis

La sintaxis de SQL se organiza en declaraciones que permiten realizar diversas operaciones:

- **SELECT**: Consulta datos.
- **INSERT**: Inserta datos.
- **UPDATE**: Actualiza datos.
- **DELETE**: Elimina datos.

3. InnoDB

Es un motor de almacenamiento para bases de datos MySQL que soporta transacciones ACID y claves foráneas. InnoDB es conocido por su alta fiabilidad y rendimiento en aplicaciones de misión crítica.

4. DML (Data Manipulation Language)

Es el subconjunto de SQL utilizado para manipular los datos de la base de datos. Incluye comandos como:

- **SELECT**: Recuperar datos.
- **INSERT**: Insertar nuevos datos.
- **UPDATE**: Modificar datos existentes.
- **DELETE**: Eliminar datos.

SQL

Migración del Álgebra Relacional a SQL

El álgebra relacional puede ser implementada directamente en SQL mediante comandos específicos. Por ejemplo:

- La **selección** en álgebra relacional corresponde a la cláusula **WHERE** en SQL.
- La **proyección** corresponde a la selección de columnas específicas en una instrucción **SELECT**.
- Las operaciones de **unión**, **intersección**, y **diferencia** se implementan en SQL mediante las sentencias **UNION**, **INTERSECT**, y **EXCEPT** respectivamente.

Integridad y Restricciones

Las bases de datos relacionales se rigen por restricciones de integridad que aseguran la coherencia de los datos.

1. Integridad de Atributo

Se refiere a que los valores de cada columna deben pertenecer a un dominio válido (por ejemplo, tipo de dato, rango de valores).

2. Integridad de Filas

Las tuplas de una tabla deben cumplir ciertas condiciones para mantener la coherencia de la base de datos (por ejemplo, claves únicas o condiciones **CHECK**).

3. Integridad de Claves

Cada tabla debe tener una clave primaria que identifique de manera única a sus tuplas, y las claves foráneas deben hacer referencia correctamente a claves primarias de otras tablas.

- **Introducción al entorno de MYSQL**
  - **Introducción**
  - **¿Cómo lo instalamos?**
  - **Sentencias DDL y DML**
    - **Sentencias DDL**
      - **Creando la base de datos**
      - **Base de datos con clave foránea**
    - **Sentencias DML**
      - **Funciones de agregación**
      - **Funciones de agrupación**
      - **Operaciones de pertenencia a conjuntos**
      - **Operaciones con valores nulos**

Introducción al entorno de MYSQL

Introducción

MySQL es un sistema de gestión de bases de datos relacional de código abierto basado en SQL (Structured Query Language). Es ampliamente utilizado debido a su eficiencia, facilidad de uso y robustez, lo que lo convierte en una opción popular tanto para pequeñas aplicaciones como para grandes sistemas empresariales.

¿Cómo lo instalamos?

La instalación de MySQL varía dependiendo del sistema operativo.
Aquí se resumen los pasos más comunes:

1. Instalación en Windows:

- Descarga el instalador desde la página oficial de MySQL: https://dev.mysql.com/downloads/installer/
- Ejecuta el instalador y selecciona el tipo de instalación.
- Configura las opciones de seguridad, como la contraseña para el usuario **root**.
- Completa la instalación y verifica que el servicio esté corriendo.

Sentencias DDL y DML

1. Sentencias DDL (Data Definition Language):

Las sentencias **DDL** se utilizan para definir o modificar la estructura de las bases de datos y las tablas. Incluyen operaciones como crear, alterar o eliminar tablas.

a) Creando la base de datos:

Para crear una base de datos:

```
CREATE DATABASE nombre_base_de_datos;
```

b) Base de datos con clave foránea:

Cuando creamos una tabla con claves foráneas, podemos usar la siguiente estructura:

```
CREATE TABLE Empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(50),
    departamento_id INT,
    FOREIGN KEY (departamento_id) REFERENCES Departamentos(id)
);
```

Sentencias DDL y DML

Sentencias DML (Data Manipulation Language):

Las sentencias **DML** permiten manipular los datos de una base de datos. Estas sentencias incluyen la inserción, actualización y eliminación de datos.

a) Operaciones de insertar:

Para insertar un registro en una tabla:

```
INSERT INTO Empleados (id, nombre, departamento_id) VALUES (1, 'Juan', 101);
```

b) Operaciones de borrar:

Para eliminar registros de una tabla:

```
DELETE FROM Empleados WHERE id = 1;
```

c) Operaciones de modificar:

Para actualizar registros en una tabla:

```
UPDATE Empleados SET nombre = 'Ana' WHERE id = 1;
```

Sentencias DDL y DML

- ## Funciones de agregación

Las **funciones de agregación** permiten realizar cálculos en un conjunto de valores y devolver un solo valor. Algunas funciones comunes son:

SUM(): Calcula la suma de un conjunto de valores.

```sql
SELECT SUM(salario) FROM Empleados;
```

AVG(): Calcula el promedio de un conjunto de valores.

```sql
SELECT AVG(salario) FROM Empleados;
```

COUNT(): Cuenta el número de filas que cumplen con una condición.

```sql
SELECT COUNT(*) FROM Empleados WHERE departamento_id = 101;
```

- ## Funciones de agrupación

Las **funciones de agrupación** se utilizan junto con la cláusula `GROUP BY` para organizar datos en grupos basados en un valor común.

GROUP BY: Agrupa los resultados según una o más columnas.

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

Sentencias DDL y DML

- ## Operaciones de pertenencia a conjuntos

En SQL, existen operadores que permiten trabajar con conjuntos:

IN: Verifica si un valor está en un conjunto de valores.

```sql
SELECT * FROM Empleados WHERE departamento_id IN (101, 102, 103);
```

NOT IN: Verifica que un valor **no** esté en un conjunto de valores.

```sql
SELECT * FROM Empleados WHERE departamento_id NOT IN (101, 102, 103);
```

- ## Operaciones con valores nulos

Los valores **NULL** representan la ausencia de un valor. Es importante saber cómo manejarlos en consultas.

Para verificar si un valor es `NULL`, se utiliza `IS NULL`:

```sql
SELECT * FROM Empleados WHERE salario IS NULL;
```

Para verificar si un valor **no** es `NULL`, se utiliza `IS NOT NULL`:

```sql
SELECT * FROM Empleados WHERE salario IS NOT NULL;
```

- Consultas SQL con dos tablas
  - Analizar tablas
  - Validar filas
  - Salvar errores
  - Evitar errores
  - Resultado final
- Consultas SQL con más de dos tablas
  - ¿Qué es y para qué sirve la subconsulta?
  - Subconsultas en el *"Select"*
  - Subconsultas en el *"From"*
  - Subconsultas en el *"Where"*
  - Errores frecuentes a nivel del *"Where"*

Consultas SQL con dos tablas

Las **consultas SQL con dos tablas** son esenciales cuando necesitamos extraer información relacionada entre diferentes entidades. Esto se logra mediante **joins**, que permiten combinar los datos de varias tablas en una sola consulta.

1. Analizar tablas

Antes de realizar una consulta con dos tablas, es importante conocer su estructura:

- Identificar las claves primarias y foráneas.
- Verificar la relación entre las tablas.

Por ejemplo, si tenemos las tablas `Empleados` y `Departamentos`:

```sql
CREATE TABLE Empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(50),
    departamento_id INT
);

CREATE TABLE Departamentos (
    id INT PRIMARY KEY,
    nombre_departamento VARCHAR(50)
);
```

En este caso, el campo `departamento_id` en la tabla `Empleados` actúa como una clave foránea que hace referencia a la clave primaria `id` en la tabla `Departamentos`.

2. Validar filas

Para asegurar que la consulta devuelva los resultados esperados, debes validar las relaciones entre las tablas. Se puede usar una consulta que combine ambas tablas con un **JOIN** y revisar los resultados:

```
SELECT Empleados.nombre, Departamentos.nombre_departamento
FROM Empleados
JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

Esta consulta muestra los nombres de los empleados junto con los nombres de los departamentos a los que pertenecen.

3. Salvar errores

Para evitar errores como combinaciones incorrectas o datos faltantes, es útil emplear los **joins** apropiados:

- **INNER JOIN** devuelve solo las filas con coincidencias en ambas tablas.
- **LEFT JOIN** devuelve todas las filas de la primera tabla y las coincidencias de la segunda tabla (rellenando con `NULL` si no hay coincidencias).

```
SELECT Empleados.nombre, Departamentos.nombre_departamento
FROM Empleados
LEFT JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

Este ejemplo incluiría empleados que no están asignados a ningún departamento.

4. Evitar errores

Para evitar errores comunes en consultas con dos tablas:

- Verifica que las claves primarias y foráneas coincidan.
- Usa el **tipo correcto de join** según las necesidades de la consulta.
- Si trabajas con datos que pueden estar vacíos o incompletos, considera usar joins externos (como `LEFT JOIN`).

5. Resultado final

El resultado final de la consulta con dos tablas debe ser revisado para asegurar que contenga los datos esperados y que esté correctamente estructurado. En el ejemplo anterior, esperamos ver una lista de empleados y sus departamentos correspondientes.

Consultas SQL con más de dos tablas

Cuando se necesita consultar información de más de dos tablas, los **joins múltiples** son la solución. Aquí, las tablas adicionales se unen una tras otra.

Ejemplo con tres tablas

Si además de `Empleados` y `Departamentos`, tienes una tabla llamada `Proyectos`, podrías hacer algo como lo siguiente:

```
SELECT Empleados.nombre, Departamentos.nombre_departamento, Proyectos.nombre_proyecto
FROM Empleados
JOIN Departamentos ON Empleados.departamento_id = Departamentos.id
JOIN Proyectos ON Empleados.id = Proyectos.empleado_id;
```

En este caso, se combinan los nombres de empleados, departamentos y proyectos.

¿Qué es y para qué sirve la subconsulta?

Una **subconsulta** es una consulta anidada dentro de otra consulta principal. Las subconsultas son útiles cuando quieres obtener resultados en una consulta secundaria que se utilizan en la consulta principal.

- **Uso principal:** Filtrar, seleccionar o calcular datos específicos que luego se integran en una consulta más grande.

Subconsultas en el "SELECT"

Las **subconsultas en el `SELECT`** son útiles cuando se necesita calcular un valor en base a otra tabla. Por ejemplo, si deseas mostrar el salario más alto en una tabla junto a cada empleado:

```
SELECT nombre, (SELECT MAX(salario) FROM Empleados) AS salario_maximo
FROM Empleados;
```

¿Qué es y para qué sirve la subconsulta?

Subconsultas en el "FROM"

Las **subconsultas en el `FROM`** permiten crear una tabla temporal basada en una consulta, que luego es utilizada en la consulta principal.

```
SELECT empleados.nombre, salarios.salario_promedio
FROM Empleados empleados,
        (SELECT departamento_id, AVG(salario) AS salario_promedio FROM Empleados GROUP BY
departamento_id) salarios
WHERE empleados.departamento_id = salarios.departamento_id;
```

Subconsultas en el "WHERE"

Las **subconsultas en el `WHERE`** se usan frecuentemente para comparar los resultados de una consulta secundaria con los valores de la consulta principal.

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

Este ejemplo devuelve los empleados cuyo salario es mayor que el salario promedio.

Errores frecuentes a nivel del "WHERE"

Algunos errores comunes al usar subconsultas en el `WHERE` son:

1. Uso incorrecto de operadores con subconsultas que devuelven más de un valor. Si la subconsulta devuelve varios resultados, debes usar `IN` en lugar de `=`:

```
SELECT nombre
FROM Empleados
WHERE departamento_id IN (SELECT id FROM Departamentos WHERE
nombre_departamento = 'Ventas');
```

2. Comparar columnas incorrectas entre la consulta principal y la subconsulta. Asegúrate de que las columnas que compares sean compatibles en tipo y contexto.

3. Uso excesivo de subconsultas: En algunos casos, las subconsultas pueden ser innecesarias o reemplazables por un **JOIN**, lo que mejora la legibilidad y rendimiento de la consulta.

- Join y sus combinaciones
  - Inner join
  - Left outer join
  - Left outer join with exclusion
  - Right outer join
  - Right outer join with exclusion
  - Unión

Join y sus combinaciones

Los **joins** en SQL permiten combinar filas de dos o más tablas basándose en una condición común entre ellas. A continuación, exploraremos las diferentes combinaciones de joins y sus ejemplos.

1. INNER JOIN

El **INNER JOIN** devuelve las filas que tienen coincidencias en ambas tablas. Si una fila no tiene una coincidencia en una de las tablas, no será devuelta en el resultado.

```
SELECT Empleados.nombre, Departamentos.nombre_departamento
FROM Empleados
INNER JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

Resultado: Este `INNER JOIN` muestra solo los empleados que están asignados a un departamento.

Join y sus combinaciones

2. LEFT OUTER JOIN

El **LEFT JOIN** devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha. Si no hay coincidencia en la tabla derecha, las columnas de esa tabla tendrán valores NULL.

```
SELECT Empleados.nombre, Departamentos.nombre_departamento
FROM Empleados
LEFT JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

Resultado: Devuelve todos los empleados, incluso aquellos que no están asignados a ningún departamento (en cuyo caso, el nombre del departamento será NULL).

3. LEFT OUTER JOIN with Exclusión

Este join incluye todas las filas de la tabla izquierda que **no tienen coincidencias** en la tabla derecha, excluyendo las filas que tienen coincidencias.

```
SELECT Empleados.nombre
FROM Empleados
LEFT JOIN Departamentos ON Empleados.departamento_id = Departamentos.id
WHERE Departamentos.id IS NULL;
```

Resultado: Muestra los empleados que **no están asignados** a ningún departamento.

Join y sus combinaciones

4. RIGHT OUTER JOIN

El **RIGHT JOIN** es similar al `LEFT JOIN`, pero devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si no hay coincidencias en la tabla izquierda, las columnas correspondientes tendrán valores `NULL`.

```sql
SELECT Empleados.nombre, Departamentos.nombre_departamento
FROM Empleados
RIGHT JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

Resultado: Devuelve todos los departamentos y los empleados asignados a cada uno. Si hay departamentos sin empleados, sus nombres igualmente aparecerán en el resultado, pero los valores de los empleados serán `NULL`.

5. RIGHT OUTER JOIN with Exclusión

Este join incluye todas las filas de la tabla derecha que **no tienen coincidencias** en la tabla izquierda, excluyendo las filas con coincidencias.

```sql
SELECT Departamentos.nombre_departamento
FROM Empleados
RIGHT JOIN Departamentos ON Empleados.departamento_id = Departamentos.id
WHERE Empleados.id IS NULL;
```

Resultado: Devuelve los departamentos que **no tienen empleados** asignados.

Join y sus combinaciones

6. UNIÓN

La operación **UNION** combina los resultados de dos consultas en una sola lista. Cada consulta debe devolver el mismo número de columnas y con tipos de datos compatibles. Por defecto, la operación `UNION` elimina las filas duplicadas. Si deseas incluir los duplicados, puedes usar `UNION ALL`.

```sql
SELECT nombre FROM Empleados
UNION
SELECT nombre_departamento FROM Departamentos;
```

Resultado: Combina los nombres de los empleados y los nombres de los departamentos en un solo conjunto de resultados, eliminando los duplicados.

- Consultas complejas
  - Caso 1: función de función
    i. Valor máximo
  - Caso 2: uso de combinaciones
    i. Pasos
    ii. Uso del If
    iii. Transformación de la consulta
    iv. Verificación de resultado
- Datos Fecha / Hora

Consultas Complejas en SQL

Cuando trabajamos con **consultas complejas** en SQL, necesitamos combinar funciones, subconsultas, condicionales y trabajar con diferentes tipos de datos, como fechas. Aquí presentamos dos casos que ilustran estas técnicas: **funciones anidadas** y **combinación de condiciones** con el uso de `IF`.

Caso 1: Función de Función

En este caso, anidamos varias funciones para obtener resultados más complejos.

Ejemplo: Valor máximo

Queremos obtener el valor máximo de ventas por empleado en un determinado período y, a la vez, calcular cuántos días trabajó cada empleado en ese mismo período. Esto requiere el uso de varias funciones, incluyendo `MAX` y `COUNT`.

```sql
SELECT Empleado_id,
       MAX(ventas) AS Venta_Maxima,
       COUNT(DISTINCT fecha_trabajo) AS Dias_Trabajados
FROM Ventas
WHERE fecha_trabajo BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY Empleado_id;
```

Explicación:

- La función `MAX(ventas)` devuelve la venta más alta realizada por cada empleado en el período.
- La función `COUNT(DISTINCT fecha_trabajo)` cuenta los días únicos que el empleado ha trabajado.
- `GROUP BY Empleado_id` agrupa los resultados por empleado.

Consultas Complejas en SQL

Caso 2: Uso de Combinaciones

Este caso trata sobre combinar varias condiciones, operadores lógicos y el uso de la función `IF` para hacer la consulta más dinámica y ajustada a los resultados requeridos.

Pasos para combinar condiciones

1. Definir claramente las condiciones que vamos a combinar.
2. Utilizar operadores lógicos (`AND`, `OR`) para combinar las condiciones.
3. Aplicar funciones condicionales como `IF` o `CASE` para ajustar el resultado según las condiciones.

Ejemplo: Uso de `IF` para combinar condiciones

Queremos seleccionar a los empleados con más de 100 ventas y verificar si esas ventas son mayores o iguales a un umbral específico, por ejemplo, 5000 dólares. Si es mayor a ese umbral, le asignaremos un "Bonus".

```sql
SELECT Empleado_id,
       SUM(ventas) AS Total_Ventas,
       IF(SUM(ventas) >= 5000, 'Bonus', 'No Bonus') AS Estado_Bonus
FROM Ventas
GROUP BY Empleado_id
HAVING Total_Ventas > 100;
```

Datos Fecha / Hora

Las consultas complejas a menudo requieren trabajar con tipos de datos `DATE`, `TIME` o `DATETIME`. SQL ofrece varias funciones para manipular fechas y horas, que son esenciales en análisis temporales.

Funciones de Fecha y Hora Comunes

- `NOW()`: Devuelve la fecha y hora actuales.
- `CURDATE()`: Devuelve la fecha actual.
- `DATEDIFF(fecha1, fecha2)`: Calcula la diferencia entre dos fechas en días.
- `DATE_ADD(fecha, INTERVAL valor unidad)`: Añade un intervalo a una fecha.
- `DATE_SUB(fecha, INTERVAL valor unidad)`: Resta un intervalo a una fecha

Explicación:

```sql
SELECT Empleado_id, nombre, fecha_trabajo
FROM Empleados
WHERE fecha_trabajo BETWEEN CURDATE() - INTERVAL 30 DAY AND CURDATE();
```

- Utilizamos `CURDATE()` para obtener la fecha actual.
- `INTERVAL 30 DAY` resta 30 días de la fecha actual, devolviendo los empleados que han trabajado en ese período.

- Store Procedure (Procedimientos almacenados)
  - Conceptos previos – Variables de Usuario
  - Conceptos previos - Delimitador.
  - Conceptos previos - Procedure / Function.
  - Parámetros
- Cursores
  - Cómo trabajar con un cursor
  - Ejemplo de código- Procedure con cursor

Procedimientos Almacenados en MySQL

Los procedimientos almacenados son conjuntos de instrucciones SQL que se pueden almacenar y ejecutar en el servidor de la base de datos. Esto permite una ejecución más eficiente y la reutilización de código, además de ayudar a mantener la lógica de negocio en la base de datos.

1. Conceptos Previos – Variables de Usuario

Las **variables de usuario** en MySQL son variables temporales que existen durante la duración de una sesión y permiten almacenar datos que se pueden utilizar en múltiples consultas o procedimientos almacenados.

Características de las Variables de Usuario:

- Comienzan con el símbolo `@`.
- Son **globales** dentro de la sesión del usuario que las crea, es decir, pueden ser usadas en cualquier consulta o procedimiento mientras dure la sesión.
- No es necesario declarar explícitamente su tipo de dato. El tipo de la variable se define dinámicamente según el valor asignado.
- Se pueden usar en instrucciones **SELECT**, **UPDATE**, **INSERT**, **DELETE**, y dentro de **procedimientos almacenados**.

```sql
-- Asignar un valor a una variable de usuario
SET @mi_variable = 100;

-- Utilizar la variable en una consulta
SELECT @mi_variable AS valor_inicial;

-- Cambiar el valor de la variable
SET @mi_variable = @mi_variable + 50;

-- Usar la variable en otra consulta
SELECT @mi_variable AS valor_actual;
```

2. Conceptos Previos - Delimitador

El **delimitador** es un símbolo que MySQL utiliza para reconocer el final de una sentencia SQL. Por defecto, MySQL utiliza el **punto y coma** (`;`) como delimitador, pero al trabajar con procedimientos almacenados, triggers o funciones, a veces es necesario cambiar el delimitador temporalmente.

¿Por qué cambiar el delimitador?

- Los procedimientos almacenados y triggers suelen incluir múltiples instrucciones SQL, y si se utiliza el delimitador `;` dentro del código, MySQL pensará que la definición del procedimiento ha terminado prematuramente.
- Cambiar el delimitador temporalmente permite incluir múltiples sentencias SQL dentro de una definición de procedimiento sin que se termine el bloque antes de tiempo.

Cómo Cambiar el Delimitador:

1. Se utiliza la instrucción `DELIMITER` para cambiar el símbolo de delimitador.
2. Al finalizar la creación del procedimiento o bloque, se restaura el delimitador original (`;`).

```
-- Cambiar el delimitador para definir un procedimiento
DELIMITER //

CREATE PROCEDURE ejemplo_procedimiento()
BEGIN
    DECLARE x INT;
    SET x = 10;
    SELECT x;
END //

-- Restaurar el delimitador original
DELIMITER ;
```

3. Conceptos Previos - Procedure / Function

En MySQL, los **procedimientos almacenados** y las **funciones** son dos tipos de rutinas que permiten ejecutar bloques de código SQL almacenados en el servidor de la base de datos.

Procedimientos Almacenados (Stored Procedures):

- Son **conjuntos de instrucciones SQL** que se pueden ejecutar en el servidor.
- No devuelven un valor directamente (aunque pueden usar parámetros de salida para devolver resultados).
- Se utilizan para **automatizar tareas repetitivas** o realizar operaciones complejas, como cálculos o procesos que involucran múltiples tablas y condiciones.
- Se pueden invocar utilizando la instrucción `CALL`.

```
CREATE PROCEDURE nombre_procedimiento([parámetros])
BEGIN
    -- Cuerpo del procedimiento
END;
```

```
DELIMITER //

CREATE PROCEDURE saludar(IN nombre_usuario VARCHAR(50))
BEGIN
    SELECT CONCAT('Hola, ', nombre_usuario, '!') AS saludo;
END //

DELIMITER ;

-- Llamar al procedimiento
CALL saludar('Eduardo');
```

3. Conceptos Previos - Procedure / Function

Funciones (Functions):

- A diferencia de los procedimientos, **las funciones deben devolver un valor**.
- Se pueden utilizar dentro de expresiones SQL como parte de una consulta.
- Son útiles para **realizar cálculos y operaciones** que devuelvan un solo valor, como sumar, concatenar, calcular promedio, etc.
- Se invocan directamente desde una instrucción SQL, igual que las funciones nativas como `SUM()` o `COUNT()`.

```
CREATE FUNCTION nombre_funcion([parámetros]) RETURNS tipo_dato
BEGIN
    -- Cuerpo de la función
    RETURN valor;
END;
```

```
DELIMITER //

CREATE FUNCTION obtener_doble(num INT) RETURNS INT
BEGIN
    RETURN num * 2;
END //

DELIMITER ;

-- Utilizar la función
SELECT obtener_doble(5) AS resultado;
```

Diferencias entre Procedure y Function:

- **Procedimientos** no devuelven un valor directamente y se invocan con `CALL`.
- **Funciones** devuelven un valor y pueden ser utilizadas en cualquier consulta SQL.
- Los **procedimientos** suelen usarse para modificar datos o realizar acciones complejas, mientras que las **funciones** se emplean para realizar cálculos o transformar datos.

3. Cursores

Los **cursores** en MySQL son estructuras que permiten recorrer fila por fila los resultados de una consulta. Se utilizan dentro de los **procedimientos almacenados** cuando necesitamos procesar los resultados de una consulta de manera secuencial, ya que una consulta SQL normal devuelve un conjunto de resultados, pero no tiene una manera directa de procesar cada fila individualmente.

¿Cuándo utilizar un Cursor?

Los cursores son útiles cuando:

- Deseamos **procesar filas una por una** de un conjunto de resultados.
- Debemos realizar **acciones iterativas** sobre cada fila recuperada de una consulta, como actualizaciones o cálculos.

Partes principales de un cursor:

1. **Declaración** del cursor: Se define el cursor con una consulta SQL.
2. **Apertura** del cursor: Inicia el cursor y prepara el conjunto de resultados para ser recorrido.
3. **Lectura** del cursor: Se recuperan filas una por una.
4. **Cierre** del cursor: Se libera el recurso una vez que el procesamiento ha terminado.

3. Cursores

Sintaxis básica de un Cursor en MySQL:

```
DECLARE nombre_cursor CURSOR FOR consulta_sql;
```

Operaciones básicas con cursores:

- **DECLARAR** el cursor.
- **ABRIR** el cursor.
- **RECUPERAR** una fila (FETCH).
- **CERRAR** el cursor.

```
CREATE PROCEDURE procesar_empleados()
BEGIN
    DECLARE fin_cursor BOOLEAN DEFAULT FALSE;
    DECLARE emp_id INT;
    DECLARE emp_nombre VARCHAR(50);

    DECLARE cursor_empleados CURSOR FOR
    SELECT id, nombre FROM empleados;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin_cursor = TRUE;

    OPEN cursor_empleados;

    leer_empleados: LOOP
        FETCH cursor_empleados INTO emp_id, emp_nombre;
        IF fin_cursor THEN LEAVE leer_empleados; END IF;

        -- Aquí se puede procesar cada fila
        SELECT emp_id, emp_nombre;
    END LOOP;

    CLOSE cursor_empleados;
END //
```

- *Trigger*
  - Creación del *trigger.*
  - *New* y *Old*.
  - Ejemplo de *trigger.*
- Transacción
  - Ejemplo de transacción.
- Vista
  - Ejemplo de vista.

1. Trigger

Un *Trigger* es un bloque de código que se ejecuta automáticamente antes o después de una operación de inserción, actualización o eliminación en una tabla. Los *Triggers* se utilizan para aplicar reglas, validaciones o cálculos automáticos.

Creación del Trigger

Un *Trigger* puede ser creado usando la sentencia `CREATE TRIGGER`. Se asocia a una tabla y puede ejecutarse antes o después de un evento como `INSERT`, `UPDATE` o `DELETE`.

New y Old

- `NEW`: Hace referencia a los valores nuevos de una fila tras un `INSERT` o un `UPDATE`.
- `OLD`: Hace referencia a los valores antiguos de una fila antes de un `DELETE` o un `UPDATE`.

Este *Trigger* actualiza automáticamente el campo `stock` en una tabla `productos` cuando se inserta una nueva venta en la tabla `ventas`.

```
CREATE TRIGGER actualizar_stock
AFTER INSERT ON ventas
FOR EACH ROW
BEGIN
    UPDATE productos
    SET stock = stock - NEW.cantidad
    WHERE id_producto = NEW.id_producto;
END;
```

Este *Trigger* se ejecuta después de que se inserte un registro en la tabla `ventas` y ajusta el stock del producto.

2. **Transacción**

Una *Transacción* es un conjunto de operaciones SQL que se ejecutan como una unidad lógica. En una transacción, todas las operaciones deben completarse con éxito; de lo contrario, ninguna operación se aplica (se hace un rollback).

Este ejemplo muestra cómo usar una transacción para asegurar que tanto la actualización de stock como el registro de una venta se realicen correctamente.

```sql
START TRANSACTION;

-- Actualizamos el stock del producto
UPDATE productos
SET stock = stock - 2
WHERE id_producto = 1;

-- Insertamos una nueva venta
INSERT INTO ventas (id_producto, cantidad, fecha)
VALUES (1, 2, NOW());

-- Si todo va bien, confirmamos la transacción
COMMIT;

-- Si hay error, revertimos los cambios
ROLLBACK;
```

En este ejemplo, si ambas operaciones (`UPDATE` y `INSERT`) se ejecutan correctamente, se confirma la transacción con `COMMIT`. Si ocurre algún error, se revierte todo con `ROLLBACK`.

3. **Vista**

Una *Vista* es una consulta predefinida que se guarda en la base de datos y se puede consultar como si fuera una tabla. Las *Vistas* son útiles para simplificar consultas complejas, ocultar detalles internos de las tablas, y ofrecer seguridad al restringir el acceso a ciertos datos.

Beneficios de una Vista

1. **Simplificación de consultas**: Evita la repetición de consultas complejas.
2. **Seguridad**: Permite ocultar ciertas columnas o datos de las tablas.
3. **Reutilización**: Una vista puede ser utilizada en varias consultas sin necesidad de redefinirla.

Aquí se crea una *Vista* que muestra los productos junto con su categoría:

```
CREATE VIEW productos_categorias AS
SELECT p.nombre AS producto, c.nombre AS categoria
FROM productos p
JOIN categorias c ON p.id_categoria = c.id_categoria;
```

Ahora puedes consultar la *Vista* como si fuera una tabla:

```
SELECT * FROM productos_categorias;
```

Esta *Vista* permite acceder fácilmente a la relación entre productos y categorías sin necesidad de escribir un `JOIN` cada vez.

- **Ejemplos gratuitos de diagramas ERD**
  https://gitmind.com/es/ejemplos-diagrama-erd.html

1.1.2001 Técnicas de Programación:
https://docs.google.com/presentation/d/15ubYB7u7O91sEeTtv6Z8ya_AZ2kvVDRoNkbWwunGixI/edit?usp=sharing
1.1.2002 Administración de BD:
https://docs.google.com/presentation/d/1pbFTtjwBTkBnNU-uPxJ8f2eTS2H9ZowmpIX8MzQkiA4/edit?usp=sharing
1.1.2003 Análisis Matemático:
https://docs.google.com/presentation/d/1aajT9oYGQ3bFvGYXqZUcKT35Fc9U-T-qdRhw5iDYyC4/edit?usp=sharing
1.1.2004 Lógica Computacional:
https://docs.google.com/presentation/d/14J2V8isuKly2brKi-0qFqk5kBufj_-cfWT93NS-OEU4/edit?usp=sharing