
Bases
de
datos

Base de datos

Tema 1. Fundamentos de bases de datos

¿Qué es una base de datos?

Definición y concepto

Una base de datos es un sistema que permite almacenar, organizar y gestionar datos de forma estructurada. Su objetivo principal es facilitar el acceso, modificación y eliminación de datos de manera eficiente, lo que resulta esencial para el funcionamiento de aplicaciones, sistemas empresariales y herramientas digitales modernas.

Las bases de datos son fundamentales para analizar grandes volúmenes de datos y generar información útil para la toma de decisiones estratégicas.

Datos vs. Información

- Datos: Son hechos crudos y sin procesar. Por ejemplo: "25", "2025-01-01", "Cliente A".
- Información: Surge al procesar los datos, dándoles contexto y significado útil. Por ejemplo: "El cliente A realizó una compra de 25 unidades el 1 de enero de 2025".

Los datos por sí mismos pueden no ser útiles, pero una vez procesados y analizados, se convierten en información que puede emplearse para tomar decisiones, identificar patrones o resolver problemas.

Ejemplos cotidianos de bases de datos

- E-commerce: Los catálogos de productos, información de clientes, registros de pedidos y datos de envío están almacenados en bases de datos, permitiendo que las plataformas gestionen transacciones y ofrezcan recomendaciones personalizadas.

- Hospitales: Los historiales médicos de los pacientes, la programación de citas y los registros de medicación se almacenan para garantizar un seguimiento adecuado y evitar errores en los tratamientos.

- Redes sociales: Perfiles de usuarios, publicaciones, comentarios, reacciones y conexiones entre usuarios están gestionados mediante bases de datos para ofrecer experiencias interactivas y personalizadas.

En cada uno de estos ejemplos, las bases de datos no solo almacenan información, sino que también permiten acceder a ella de forma rápida y eficiente, lo que es crucial para el éxito de estas aplicaciones.

Tipos de bases de datos

Relacionales vs. No Relacionales

- Relacionales: Organizan los datos en tablas (estructuras de filas y columnas) con relaciones definidas entre ellas. Utilizan SQL (Structured Query Language) para gestionar y consultar datos. Aunque se asocian principalmente con datos estructurados, también pueden manejar grandes volúmenes de datos en entornos de big data mediante tecnologías adicionales como Hadoop o Spark.
  - Ejemplo: MySQL, PostgreSQL, Oracle.

- No Relacionales: Diseñadas para trabajar con datos no estructurados o semiestructurados, como documentos, grafos o pares clave-valor. Ofrecen flexibilidad y escalabilidad horizontal, lo que las hace ideales para aplicaciones modernas que manejan grandes volúmenes de datos y requieren respuestas rápidas.
  - Ejemplo: MongoDB, Cassandra, Redis.

Bases de datos transaccionales vs. analíticas

- Transaccionales (OLTP): Optimizadas para operaciones diarias en tiempo real, como la gestión de pedidos en un e-commerce. Permiten muchas transacciones simultáneas, garantizando la consistencia y la integridad de los datos.
  - Caso de uso: Procesamiento de pagos, gestión de inventarios.

- Analíticas (OLAP): Diseñadas para realizar consultas complejas sobre grandes volúmenes de datos históricos. Son ideales para generar informes, identificar tendencias y apoyar la toma de decisiones estratégicas.
  - Caso de uso: Análisis de datos de ventas, generación de dashboards.

Otros tipos de bases de datos

- Jerárquicas: Organizan datos en una estructura de árbol, donde cada nodo tiene un único padre. Aunque son menos comunes en la actualidad, se usaron ampliamente en sistemas antiguos.
  - Caso de uso: Sistemas de gestión de archivos.

- En red: Amplían el modelo jerárquico permitiendo relaciones múltiples entre nodos. Ofrecen mayor flexibilidad que las bases jerárquicas, pero su complejidad las hace menos populares.
  - Caso de uso: Sistemas bancarios antiguos.

- Orientadas a objetos: Combinan conceptos de bases de datos con la programación orientada a objetos. Almacenan datos como objetos, lo que facilita su uso en aplicaciones que también emplean este paradigma.
  - Caso de uso: Aplicaciones de diseño asistido por computadora (CAD/CAM).

Casos de uso para cada tipo

- Relacionales: Sistemas de gestión financiera, ERPs, CRMs. También son comunes en análisis de datos mediante clústeres distribuidos o herramientas de big data.
- No Relacionales: Aplicaciones en redes sociales, IoT (Internet de las cosas), almacenamiento de documentos y big data.
- Transaccionales: Registro de transacciones bancarias, gestión de reservas de vuelos.
- Analíticas: Predicción de tendencias de mercado, análisis de comportamiento del cliente.
- Jerárquicas: Gestión de datos jerárquicos, como organigramas o sistemas de archivos.
- En red: Modelado de relaciones complejas, como redes de transporte.
- Orientadas a objetos: Aplicaciones multimedia y sistemas científicos que requieren estructuras de datos complejas.

Componentes de una base de datos

En bases de datos relacionales

- Tablas: Son el núcleo de las bases de datos relacionales. Una tabla organiza los datos en filas y columnas. Cada tabla representa una entidad, como "Clientes" o "Productos".

- Registros: También llamados filas, representan una instancia única dentro de una tabla. Por ejemplo, un cliente específico dentro de la tabla "Clientes".

- Columnas: También conocidas como campos, representan los atributos de la entidad. Por ejemplo, "Nombre" y "Correo electrónico" en la tabla "Clientes".

- Claves primarias: Un identificador único para cada registro en una tabla. Garantiza que cada fila sea única.

- Claves foráneas: Un campo en una tabla que referencia una clave primaria en otra tabla, estableciendo relaciones entre ellas.

En bases de datos no relacionales

- Documentos: Equivalentes a los registros en las bases relacionales. Almacenan datos en formatos como JSON o BSON. Un documento puede contener toda la información relacionada con una entidad.

- Colecciones: Agrupaciones de documentos, similares a las tablas en las bases relacionales. Cada colección almacena documentos relacionados entre sí.

- Claves: Identificadores únicos dentro de un modelo de clave-valor. En bases de datos como Redis, las claves permiten acceder rápidamente a valores específicos.

Estos componentes, aunque varían entre los tipos de bases de datos, cumplen el mismo objetivo: organizar los datos de manera que puedan ser utilizados de forma eficiente y estructurada.

Tipos de datos en bases de datos

¿Qué son los tipos de datos?

Los tipos de datos definen qué clase de información puede almacenarse en una columna de una tabla en una base de datos. Elegir el tipo de dato correcto es crucial para garantizar que:

- Los datos se almacenen de manera eficiente.
- Se eviten errores de validación.
- El rendimiento del sistema sea óptimo.

Clasificación general de los tipos de datos

Cadenas de texto

- CHAR(n): Cadena de texto de longitud fija. Por ejemplo, CHAR(5) almacena exactamente 5 caracteres.
- VARCHAR(n): Cadena de texto de longitud variable, con un límite máximo definido. Por ejemplo, VARCHAR(50) puede almacenar hasta 50 caracteres.
- TEXT: Almacena cadenas de texto largas. Ideal para descripciones extensas, aunque puede ser menos eficiente para búsquedas.

Números

- INTEGER o INT: Enteros (números sin decimales).
- BIGINT: Enteros grandes, útiles para identificadores o conteos extensos.
- SMALLINT: Enteros pequeños, usados para ahorrar espacio cuando los valores son limitados.
- DECIMAL(p, s) o NUMERIC(p, s): Números con decimales. p es el número total de dígitos y s los dígitos después del punto decimal.
- FLOAT o REAL: Números en punto flotante, útiles para cálculos científicos o valores aproximados.

Fechas y horas

- DATE: Almacena fechas (año, mes, día).
- TIME: Almacena horas (horas, minutos, segundos).
- DATETIME: Combina fecha y hora.
- TIMESTAMP: Similar a DATETIME, pero incluye información de zona horaria.

Booleanos

- BOOLEAN: Solo admite los valores TRUE o FALSE.

Otros tipos de datos

- BLOB (Binary Large Object): Para almacenar datos binarios, como imágenes o archivos.
- UUID (Universally Unique Identifier): Identificadores únicos a nivel global.
- JSON: Almacena datos en formato JSON, útil en bases no relacionales o híbridas.

Base de datos

Tema 2. Modelado de bases de datos

Conceptos clave

Normalización y sus niveles

La normalización es una técnica utilizada para organizar los datos en una base de datos, reduciendo la redundancia y mejorando la integridad. Se divide en varios niveles o formas normales:

- **Primera forma normal (1FN)**: Los datos deben estar en formato tabular, sin duplicados y con valores atómicos (no divisibles). En este nivel, los datos deben organizarse en una tabla en la que:

- Cada celda contenga un único valor (valores atómicos).
- No haya conjuntos repetidos de datos en una sola columna.
- No existan filas duplicadas.

Ejemplo antes de la 1FN:

ID Cliente	Nombre	Teléfonos
1	Juan	655765577,644342211

Ejemplo en 1FN:

ID Cliente	Nombre	Teléfonos
1	Juan	655765577
1	Juan	644342211

- Segunda forma normal (2FN): Requiere que todos los atributos no clave dependan completamente de la clave primaria. Aquí, además de cumplir con la 1FN, se requiere que:

- Todos los atributos que no son clave dependan completamente de la clave primaria (es decir, no deben depender de una parte de una clave compuesta).

Ejemplo antes de la 2FN: Supongamos una tabla de pedidos

ID Pedido	ID Cliente	Nombre cliente	Producto
1	1	Juan	camiseta
2	1	Juan	zapatos

Aquí, "Nombre Cliente" depende de "ID Cliente", no de "ID Pedido". Este dato debería moverse a otra tabla.

Ejemplo en 2FN:

Tabla clientes

ID Cliente	Nombre
1	Juan

Tabla pedidos

ID Pedido	ID Cliente	Producto
1	1	camiseta
2	1	zapatos

- Tercera forma normal (3FN): Elimina las dependencias transitivas, asegurando que los atributos no clave dependan únicamente de la clave primaria. Además de cumplir con la 2FN, elimina dependencias transitivas:

- Un atributo no clave no debe depender de otro atributo no clave.

Ejemplo antes de la 3FN:

ID Pedido	ID Cliente	Nombre cliente	Ciudad cliente
1	1	Juan	Madrid
2	1	Juan	Madrod

Aquí, "Ciudad Cliente" depende de "ID Cliente" a través de "Nombre Cliente".

Ejemplo en 3FN:

Tabla clientes

ID Cliente	Nombre	Ciudad
1	Juan	Madrid

Tabla pedidos

ID Pedido	ID Cliente	Producto
1	1	camiseta
2	1	zapatos

-La normalización es un proceso progresivo para organizar los datos y evitar redundancias.

- La 1FN organiza los datos en tablas limpias y sin duplicados.
- La 2FN asegura que todos los atributos dependen de toda la clave primaria.
- La 3FN elimina dependencias indirectas entre los datos.

Entidades, relaciones y cardinalidades

- Entidades: Representan objetos o conceptos del mundo real, como "Clientes" o "Productos".
- Relaciones: Representan las asociaciones entre entidades, como "Clientes realizan Pedidos".
- Cardinalidades: Describen el número de asociaciones posibles entre entidades (uno a uno, uno a muchos, muchos a muchos).

Diagramas ER (Entidad-Relación)

Un diagrama ER es una representación gráfica del modelo de datos. Incluye entidades, relaciones y sus atributos, permitiendo visualizar cómo se estructuran los datos.

Buenas prácticas para el diseño de bases de datos

- Analizar los requisitos del sistema antes de modelar.
- Aplicar la normalización para evitar redundancias.
- Definir claves primarias y foráneas adecuadamente.
- Documentar el modelo de datos para facilitar su mantenimiento.

Ejemplo práctico

Diseño de un modelo simple de base de datos para una tienda online

Paso 1. Identificar las entidades principales:

- Clientes
- Productos
- Pedidos

Paso 2. Definir las relaciones entre entidades:

Un cliente puede realizar uno o más pedidos (relación uno a muchos).

Un pedido puede incluir uno o más productos, y cada producto puede estar en varios pedidos (relación muchos a muchos).

Paso 3. Definir las claves primarias y foráneas:

Clientes:

Clave primaria: IDCliente.

Pedidos:

Clave primaria: IDPedido.

Clave foránea: IDCliente (referencia a la tabla "Clientes").

Productos:

Clave primaria: IDProducto.

DetallePedido: (tabla intermedia para la relación muchos a muchos entre pedidos y productos):

Clave primaria compuesta: IDPedido + IDProducto.

Clave foránea: IDPedido (referencia a la tabla "Pedidos").

Clave foránea: IDProducto (referencia a la tabla "Productos").

Paso4. Tablas resultantes y estructura:

Tabla Clientes

IDCliente (Clave primaria)

Nombre

Email

Dirección

Tabla Pedidos

IDPedido (Clave primaria)

Fecha

IDCliente (Clave foránea, referencia a "Clientes")

Tabla Productos

IDProducto (Clave primaria)

NombreProducto

Precio

Tabla DetallePedido (Tabla intermedia)

IDPedido (Clave foránea, referencia a "Pedidos")

IDProducto (Clave foránea, referencia a "Productos")

Cantidad

Paso 5. Crear un diagrama ER (Entidad-Relación):

Entidades: Clientes, Pedidos, Productos.

Relaciones:

Cliente → Pedidos (uno a muchos).

Pedidos ↔ Productos (muchos a muchos, resuelto mediante DetallePedido)

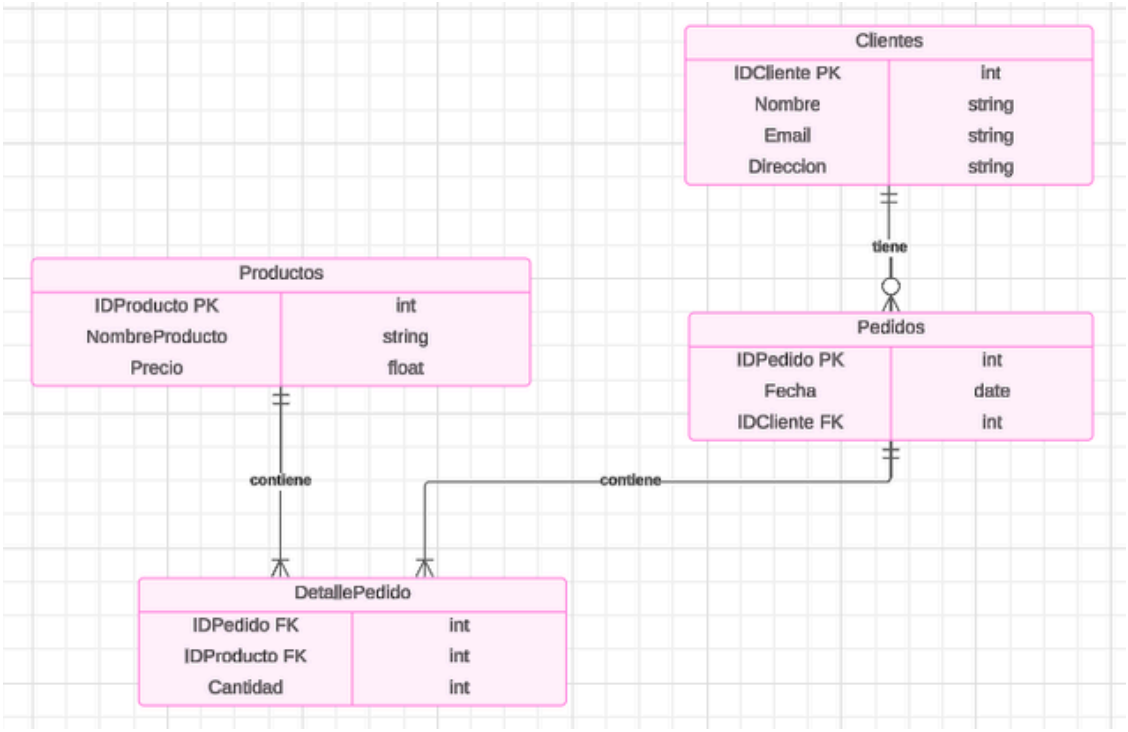

*Diagrama realizado con Lucidchart.

Explicación de las claves:

Claves primarias (PK “Primary key”: Garantizan que cada registro sea único dentro de una tabla. Por ejemplo, IDCliente asegura que no haya dos clientes con el mismo identificador.

Claves foráneas (FK “Foreign key”: Establecen las relaciones entre tablas. Por ejemplo, IDCliente en la tabla "Pedidos" conecta cada pedido con su cliente correspondiente.

Este diseño garantiza la integridad referencial y permite evitar inconsistencias en los datos. Además, al usar claves primarias y foráneas, es más fácil gestionar relaciones complejas entre las entidades.

Base de datos

Tema 3. Introducción a Sistemas de Gestión de Bases de Datos (SGBD)

¿Qué es un SGBD?

Un Sistema de Gestión de Bases de Datos (SGBD) es un software diseñado para crear, gestionar y mantener bases de datos de forma eficiente. Su función principal es actuar como intermediario entre los usuarios y las bases de datos, permitiendo almacenar, organizar, recuperar y manipular datos de manera estructurada.
Un SGBD ofrece funcionalidades esenciales como:

- Control de acceso: Gestionar quién puede acceder o modificar los datos.
- Integridad de los datos: Garantizar que los datos sean precisos y coherentes.
- Escalabilidad: Manejar grandes volúmenes de datos y usuarios simultáneamente.
- Gestión de transacciones: Asegurar que las operaciones sean completas y consistentes (ACID).
- Facilidad de consulta: Usar lenguajes como SQL (Structured Query Language) para interactuar con los datos.

Ejemplo práctico: Cuando un usuario compra un producto en una tienda en línea, el SGBD gestiona el registro del pedido, actualiza el inventario y almacena la información de pago en diferentes tablas relacionadas.

Rol del SGBD y cómo interactúa con aplicaciones y usuarios

El SGBD actúa como un puente entre las aplicaciones y los datos almacenados. Su papel incluye:
Interacción con aplicaciones:

- Las aplicaciones envían solicitudes al SGBD a través de consultas (como SQL).
- El SGBD procesa las solicitudes, consulta la base de datos y devuelve los resultados a la aplicación.
-

Ejemplo: Una aplicación de e-commerce consulta el inventario de productos. El SGBD responde con la lista de productos disponibles.

Interacción con los usuarios:

- Los usuarios pueden interactuar directamente con el SGBD mediante herramientas gráficas (interfaces de usuario) o lenguajes de consulta.
- Los administradores utilizan el SGBD para configurar permisos, realizar copias de seguridad y optimizar el rendimiento.

Ejemplo: Un analista utiliza el SGBD para extraer datos y generar informes de ventas. El SGBD asegura que las aplicaciones y los usuarios trabajen con datos consistentes, independientemente del volumen de datos o de la complejidad de las consultas.

Ejemplos de sistemas populares

Los SGBD pueden clasificarse en relacionales y no relacionales. Aquí algunos ejemplos:

SGBD Relacionales

MySQL:

- Open source y ampliamente utilizado.
- Ideal para aplicaciones web y sistemas de gestión empresarial.
- Soporta estructuras tabulares y consultas SQL.

PostgreSQL:

- Open source y muy robusto.
- Ofrece soporte avanzado para datos estructurados y no estructurados (como JSON).
- Excelente para aplicaciones que requieren transacciones complejas.

Oracle Database:

- SGBD comercial utilizado por grandes empresas.
- Ofrece alto rendimiento, escalabilidad y características avanzadas para grandes volúmenes de datos.

SGBD No Relacionales

MongoDB:

- Almacena datos en formato de documentos JSON.
- Ideal para aplicaciones ágiles y proyectos que manejan datos no estructurados.
-

Cassandra:

- Diseñada para manejar grandes volúmenes de datos distribuidos.
- Altamente escalable y utilizada en big data.

Redis:

- Sistema basado en clave-valor.
- Ofrece tiempos de respuesta ultrarrápidos, ideal para sistemas de caché o análisis en tiempo real.

Diferencias clave entre sistemas

Características	Relacionales (RDBMS)	No Relacionales (NoSQL)
Modelo de datos	Tablas con filas y columnas	Documentos, grafos, clave-valor
Estructura de datos	Datos estructurados	Datos no estructurados o semiestructurados
Lenguaje de consulta	SQL	Propietarios o APIs específicas
Escalabilidad	vertical	horizontal
Consistencia	Alta consistencia (ACID)	Flexibilidad en consistencia
Casos de uso	Aplicaciones empresariales ERPs	Big Data, análisis en tiempo real, loT

¿Cuándo elegir uno u otro?

- Relacional (RDBMS): Si necesitas consistencia y estructuras claras, como en sistemas financieros o de gestión empresarial.
- No Relacional (NoSQL): Si trabajas con grandes volúmenes de datos no estructurados o necesitas alta escalabilidad, como en redes sociales o big data.

Base de datos

Tema 4. Introducción a SQL

¿Qué es SQL?

SQL (Structured Query Language) es un lenguaje de programación diseñado para gestionar y manipular bases de datos relacionales. Permite realizar operaciones como la inserción, actualización, eliminación y consulta de datos de manera eficiente.

SQL es un estándar utilizado por sistemas de gestión de bases de datos relacionales como MySQL, PostgreSQL, Oracle y SQL Server.

Los comandos SQL se dividen en diferentes categorías según su funcionalidad.

1. **Crear la base de datos**. Para ello, utilizaremos el siguiente comando:

```
CREATE DATABASE  + “nombre_base_de_datos”
```

2. **Crear una tabla**. Al crear una tabla, tendremos en cuenta el nombre que le queremos dar a la tabla, su clave primaria, los campos que queremos que tenga dicha tabla y qué tipo de datos almacenan dichos campos.

Para crear una tabla, por ejemplo, clientes, utilizaríamos el siguiente comando:

```
CREATE TABLE clientes (
  id_cliente SERIAL PRIMARY KEY,
    nombre VARCHAR(100),
    correo VARCHAR(100),
    fecha_nacimiento DATE
);
```

3. Para poder visualizar datos, la tabla tiene que contener registros, para **insertar** datos en una tabla, se utiliza el siguiente comando:

```
INSERT INTO clientes ( nombre, correo, fecha_nacimiento) VALUES
('Ana Pérez', 'perezana@gmail.com', '22/03/1987'),
('Luis Márquez', 'luismar@hotmail.com', '02/10/2005'),
('Marcos González', 'marcosgp@gmail.com', '18/07/1992')
;
```

4. Para **consultar** datos utilizamos el siguiente comando:

```
SELECT * FROM clientes;
```

Select es la sentencia principal para realizar consultas en bases de datos, (*) significa que solicitamos toda la información de dicha tabla, pero podemos hacer una consulta únicamente de los campos que necesitemos, por ejemplo:

```
SELECT nombre FROM clientes;
```

Lo cual nos devolverá un listado con todos los nombres de la tabla clientes.

Podemos **filtrar** la búsqueda añadiendo condiciones con la cláusula WHERE:

```
SELECT nombre FROM clientes
WHERE nombre = 'Marcos González';
```

La anterior consulta nos devolverá el listado de todos los clientes cuyo nombre sea 'Marcos' en la base de datos.

5. Para **actualizar** un registro de la base de datos utilizaremos el siguiente comando:

UPDATE clientes SET nombre = 'Marcos Jesús González' WHERE nombre = 'Marcos González' ;

Si realizamos la consulta de búsqueda anterior no nos devolverá ningún resultado, ya que hemos actualizado el nombre del cliente en el registro.

6. Para **eliminar** un registro utilizamos el siguiente comando:

DELETE FROM clientes
WHERE nombre = 'Ana Pérez' AND correo = 'perezana@gmail.com';

Recuerda SIEMPRE aplicar la cláusula WHERE si no quieres eliminar todos los registros de la tabla. Ahora ya sabes hacer un CRUD (Create Read Update Delete) en una base de datos .

7. Podemos **ordenar** los resultados obtenidos de la consulta con los siguientes comandos:

ORDER BY permite ordenar los resultados en orden ascendente (ASC) o descendente (DESC):

SELECT * FROM clientes
ORDER BY nombre DESC;

En este caso, obtendremos un listado de todos los registros de todos los campos de la tabla clientes por orden alfabético descendiente del campo nombre.

GROUP BY se usa para **agrupar** registros con valores comunes y aplicar funciones de agregación:

SELECT COUNT(*) FROM clientes GROUP BY fecha_nacimiento;

Esta consulta nos daría como resultado el **conteo** de clientes que hay para cada fecha de nacimiento. De esta manera llegamos a las funciones de agregación como es el caso de COUNT que acabamos de utilizar y sirve para contar registros en una consulta.

8. COUNT función de **agregación** que nos devuelve el número de registros de la consulta. Por ejemplo, para saber el total de clientes:

SELECT COUNT(*) FROM clientes;

SUM función que **suma** todos los registros de una columna con valor numérico.

Imagina que quieres sumar el precio total de varios productos en una compra. En tu tabla “Pedidos” quieres sumar todos los valores del campo “precio”:

SELECT SUM(precio) FROM Pedidos;

AVG para calcular el **promedio** de una columna numérica, es decir con datos de tipo INT.

Imagina que la tabla clientes tiene un campo “edad” de tipo INT y quieres saber la edad media de tus clientes. utilizamos el siguiente comando:

SELECT AVG(edad) FROM clientes;

Base de datos

Tema 5. SQL Avanzado – Integración de Datos y Optimización

Introducción: La importancia de estructurar la información

Las bases de datos son fundamentales en cualquier negocio, ya que permiten gestionar grandes volúmenes de información de forma eficiente. Pensemos en una empresa que vende productos tecnológicos: necesita saber cuántos productos tiene en stock, gestionar pedidos, controlar las ventas y almacenar información sobre sus clientes y proveedores.

Si toda esta información estuviera en una sola tabla, sería difícil de manejar, propensa a errores y difícil de optimizar. Por eso, es fundamental organizar los datos en varias tablas y definir relaciones entre ellas. Aquí es donde entran en juego conceptos como normalización, claves primarias y foráneas, y operaciones de unión entre tablas.

¿Cómo organizar los datos?

Antes de diseñar una base de datos, es importante pensar en cómo estructurar la información. Algunas preguntas clave:

- ¿Cuántas tablas necesitamos?
- ¿Cómo se relacionan entre sí?
- ¿Cuáles serán las claves primarias y foráneas?
- ¿Cómo garantizamos la integridad de los datos?

Un buen diseño de base de datos se basa en la normalización, que evita la redundancia y mejora la eficiencia del almacenamiento.

Ejemplo: Para nuestra tienda de tecnología, podemos definir las siguientes tablas:

1. productos (información de los productos)
2. categorías (para clasificar los productos)
3. clientes (información de los compradores)
4. pedidos (registro de compras realizadas)
5. detalle_pedidos (productos dentro de cada pedido)

Cada una de estas tablas se relaciona mediante claves primarias y foráneas, lo que permite recuperar la información de manera eficiente.

Relacionando tablas: Claves primarias y foráneas

- Clave primaria (PRIMARY KEY): Identifica de manera única cada registro en una tabla.
- Clave foránea (FOREIGN KEY): Es un campo en una tabla que hace referencia a la clave primaria de otra tabla, permitiendo establecer relaciones.

Veamos un ejemplo de creación de tablas con claves primarias y foráneas:

```
CREATE TABLE categorias (
id_categoria INT AUTO_INCREMENT PRIMARY KEY,
nombre_categoria VARCHAR(100) NOT NULL
);

CREATE TABLE productos (
id_producto INT AUTO_INCREMENT PRIMARY KEY,
nombre VARCHAR(255) NOT NULL,
precio DECIMAL(10,2),
stock INT DEFAULT 0,
id_categoria INT, FOREIGN KEY (id_categoria) REFERENCES categorias(id_categoria)
 );
```

Esto establece una relación uno a muchos entre categorías y productos, donde cada producto pertenece a una categoría, pero una categoría puede tener muchos productos.

Combinando información de varias tablas: UNION, JOIN e INTERSECT

Cuando los datos están repartidos en varias tablas, necesitamos herramientas para combinarlos y extraer información útil.

JOIN: Relacionando datos de múltiples tablas

JOIN nos permite combinar registros de dos o más tablas basándonos en una relación entre ellas.
Ejemplo: Obtener un listado de productos con su categoría asociada.

SELECT productos.nombre, productos.precio, categorias.nombre_categoria FROM productos INNER JOIN categorias ON productos.id_categoria = categorias.id_categoria;

*Para seleccionar los campos de las tablas que sean de nuestro interés, primero ponemos el nombre de la tabla seguido de un punto y seguido del nombre del campo que queremos, podemos poner tantos campos como queramos separados por comas, luego realizamos la relación entre tablas.

Tipos de JOIN

- INNER JOIN: Devuelve solo los registros que tienen coincidencias en ambas tablas.
- LEFT JOIN: Devuelve todos los registros de la tabla izquierda y solo las coincidencias de la derecha.
- RIGHT JOIN: Devuelve todos los registros de la tabla derecha y solo las coincidencias de la izquierda.
- FULL JOIN (no soportado en MySQL): Devuelve todos los registros de ambas tablas, con o sin coincidencias.

UNION: Combinando resultados de consultas

La cláusula UNION nos permite combinar el resultado de dos consultas SQL en una sola lista.

Ejemplo: Obtener una lista única de productos de dos proveedores distintos.

SELECT nombre FROM productos_proveedor1
UNIONSELECT nombre FROM productos_proveedor2;

*Nota: UNION elimina los duplicados automáticamente. Si queremos incluir los duplicados, usamos UNION ALL.

INTERSECT: Elementos comunes entre dos consultas

INTERSECT no está disponible en MySQL, pero se puede simular con INNER JOIN.

Ejemplo: Encontrar productos que existen en ambas tablas de proveedores.

SELECT nombre FROM productos_proveedor1
INNER JOIN productos_proveedor2 ON productos_proveedor1.nombre = productos_proveedor2.nombre;

Diferencias entre MINUS y LEFT JOIN

MINUS no está disponible en MySQL, pero se puede simular con LEFT JOIN y WHERE IS NULL.

- MINUS devuelve los registros de la primera consulta que no están en la segunda.
- LEFT JOIN con WHERE IS NULL permite obtener un resultado similar.
-

Ejemplo: Obtener productos que solo existen en productos_proveedor1, pero no en productos_proveedor2.

```
SELECT nombre FROM productos_proveedor1
LEFT JOIN productos_proveedor2 ON productos_proveedor1.nombre = productos_proveedor2.nombre
WHERE productos_proveedor2.nombre IS NULL;
```

Ejemplo práctico:

Una empresa de alquiler de coches necesita gestionar su flota. Se deben registrar:

- Coches: (id_coche, marca, modelo, precio_día, disponible)
- Clientes: (id_cliente, nombre, apellido, teléfono)
- Alquileres: (id_alquiler, id_cliente, id_coche, fecha_inicio, fecha_fin)

a) Crear las tablas asegurando claves primarias y foráneas.

```
CREATE DATABASE alquiler_coches;
USE alquiler_coches;

CREATE TABLE coches (
id_coche INT AUTO_INCREMENT PRIMARY KEY,
marca VARCHAR(50),
modelo VARCHAR(50),
precio_dia DECIMAL(10,2),
disponible BOOLEAN DEFAULT TRUE
);
```

```
CREATE TABLE clientes (
id_cliente INT AUTO_INCREMENT PRIMARY KEY,
nombre VARCHAR(50),
apellido VARCHAR(50),
telefono VARCHAR(15)
);

CREATE TABLE alquileres (
id_alquiler INT AUTO_INCREMENT PRIMARY KEY,
id_cliente INT,
id_coche INT,
fecha_inicio DATE,
fecha_fin DATE,
FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente), FOREIGN KEY (id_coche)
REFERENCES coches(id_coche)
);
```

b) Insertar datos en cada tabla.

```
INSERT INTO coches (marca, modelo, precio_dia, disponible) VALUES
('Toyota', 'Corolla', 40.00, TRUE),
('Ford', 'Focus', 50.00, TRUE),
('Honda', 'Civic', 45.00, TRUE),
('BMW', 'Serie 3', 80.00, TRUE),
('Audi', 'A4', 85.00, TRUE)
;
```

```
INSERT INTO clientes (nombre, apellido, telefono) VALUES
('Carlos', 'Gómez', '123456789'),
('Ana', 'López', '987654321'),
('Luis', 'Martínez', '456123789')
;

INSERT INTO alquileres (id_cliente, id_coche, fecha_inicio, fecha_fin) VALUES
(1, 2, '2024-02-01', '2024-02-05'),
(2, 4, '2024-02-03', '2024-02-07'),
(3, 1, '2024-02-05', '2024-02-10')
;
```

c) Obtener todos los alquileres con los datos de clientes y coches.

```
SELECT alquileres.id_alquiler,
clientes.nombre AS cliente,
clientes.apellido AS apellido,
coches.marca,
coches.modelo,
alquileres.fecha_inicio,
alquileres.fecha_fin
FROM alquileres JOIN clientes ON alquileres.id_cliente = clientes.id_cliente JOIN coches ON alquileres.id_coche = coches.id_coche;
```

d) Mostrar los coches que no han sido alquilados.

```
SELECT coches.id_coche,
coches.marca,
coches.modelo
FROM coches LEFT JOIN alquileres ON coches.id_coche = alquileres.id_coche WHERE alquileres.id_coche IS NULL;
```

Utilizamos un LEFT JOIN para incluir todos los coches y filtramos aquellos que no tienen coincidencia en la tabla de alquileres (es decir, aquellos que nunca han sido alquilados).

e) Contar cuántos coches están actualmente alquilados.

```
SELECT COUNT(DISTINCT id_coche) AS coches_alquilados FROM alquileres WHERE CURDATE() BETWEEN fecha_inicio AND fecha_fin;
```

f) Listar los coches ordenados por precio de alquiler de mayor a menor.

```
SELECT id_coche, marca, modelo, precio_dia FROM coches ORDER BY precio_dia DESC;
```