**1. Cuál es la principal diferencia entre SQL y NoSQL**

La principal diferencia entre SQL y NoSQL radica en cómo estructuran, almacenan y gestionan los datos:

SQL (Structured Query Language) – Bases de datos relacionales

* Ejemplo: MySQL, PostgreSQL, SQL Server, Oracle.
* Modelo: Relacional. Usa tablas con filas y columnas.
* Esquema fijo: Requiere definir un esquema (estructura) antes de insertar datos (tipos, relaciones, claves).
* Relaciones: Usa claves primarias y foráneas para mantener integridad referencial entre tablas.
* Lenguaje estándar: Usa SQL para consultar, insertar, actualizar y borrar datos.

NoSQL (Not Only SQL) – Bases de datos no relacionales

* Ejemplo: MongoDB, CouchDB, Cassandra, Redis.
* Modelo: No relacional. Hay varios tipos (documentos, clave-valor, grafos, columnas).
* Esquema flexible: No requiere un esquema fijo. Cada registro puede tener una estructura distinta.
* Escalabilidad horizontal: Se adapta mejor al crecimiento masivo de datos, especialmente en sistemas distribuidos.
* Consultas específicas por motor: MongoDB, por ejemplo, usa una sintaxis propia basada en JSON.

. Integridad y consistencia de los datos (ACID)

* Las bases de datos SQL como MySQL siguen las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad).
* Esto garantiza que las transacciones sean confiables, seguras y coherentes, incluso ante fallos del sistema.
* Ideal para aplicaciones donde la precisión de los datos es crítica (bancos, contabilidad, sistemas de reservas, etc.).

Soporte para relaciones complejas

* SQL maneja muy bien relaciones entre entidades mediante claves foráneas.
* Podés hacer joins complejos entre múltiples tablas para obtener información de forma muy potente y estructurada.
* Esto es más difícil de lograr en NoSQL, donde generalmente se evitan relaciones entre documentos.

3.- Una clave primaria (o primary key) es un campo o conjunto de campos en una tabla de base de datos SQL que:

Identifica de forma única cada fila (registro) de la tabla

Características principales:

1. Única: No se puede repetir. Cada valor debe ser diferente en cada fila.
2. No nula: No puede contener valores NULL.
3. Inmutable: En lo posible, no debería cambiar con el tiempo.
4. Índice automático: Al declarar una clave primaria, se crea un índice para acelerar búsquedas.

Función principal:

Garantizar que cada registro en una tabla pueda ser identificado de forma única y segura, lo que:

* Permite mantener la integridad de los datos.
* Sirve como punto de referencia para relacionar tablas mediante claves foráneas.
* Facilita operaciones como **actualizar** o **eliminar** registros específicos sin ambigüedad.

**4.- Menciona 2 ventajas de NoSQL sobre SQL para tener la comparación completa?**

1. Flexibilidad en el esquema

* En NoSQL (por ejemplo, MongoDB), no necesitás definir un esquema fijo antes de insertar datos.
* Cada documento puede tener diferentes campos y estructuras.
* Esto lo hace ideal para aplicaciones con datos que cambian constantemente o no están bien definidos desde el inicio (como apps web o móviles en evolución).

*Ejemplo: en MongoDB podés guardar un documento con campos como nombre y edad, y otro con nombre, apellido y correo, sin errores.*

2.- Alta escalabilidad horizontal

* Las bases de datos NoSQL están diseñadas para distribuir los datos fácilmente en múltiples servidores (nodos).
* Esto permite manejar grandes volúmenes de datos y muchas operaciones por segundo con mejor rendimiento en sistemas distribuidos.
* Es muy usado en aplicaciones en la nube, big data, redes sociales, etc.

SQL escala verticalmente (mejorando el servidor). NoSQL escala horizontalmente (agregando más servidores comunes).

6.- **Para un** **sistema bancario**, **se recomienda usar una** **base de datos SQL** **(relacional), como MySQL**, **PostgreSQL** **u** **Oracle**.

### **Justificación:**

#### **Consistencia y transacciones seguras (ACID)**

* Los sistemas bancarios requieren que **cada transacción sea precisa y confiable** (por ejemplo, una transferencia debe debitar de una cuenta y acreditar en otra sin errores).
* Las bases de datos SQL **cumplen con las propiedades ACID**, lo cual **garantiza que los datos estén siempre en un estado válido**, incluso ante fallos o caídas del sistema.

#### **Integridad referencial**

* Permite definir relaciones entre entidades como **clientes, cuentas, movimientos, préstamos**, etc., usando **claves primarias y foráneas**.
* Esto asegura que, por ejemplo, no existan movimientos bancarios sin una cuenta asociada válida.

#### **Auditoría y control de datos**

* Es más fácil implementar **controles, reglas, restricciones y trazabilidad** de operaciones en una base relacional.
* Se pueden registrar con precisión todos los cambios, ideal para auditorías y cumplimiento de normas legales.

#### **Herramientas maduras**

* SQL tiene herramientas consolidadas para **respaldo, recuperación, replicación y monitoreo**, fundamentales en sistemas bancarios críticos que deben funcionar 24/7.

### 5.- ¿Por qué no NoSQL?

Aunque NoSQL ofrece escalabilidad, **no garantiza la misma consistencia y control de integridad por defecto**, lo que puede ser **riesgoso en un entorno donde cada centavo cuenta**.

Para un sistema bancario, **SQL** es la mejor opción porque ofrece **seguridad, consistencia, integridad y trazabilidad**, que son pilares fundamentales en el manejo de dinero.

**7- define ACID en bases de datos relacionales**

En bases de datos relacionales (como MySQL, PostgreSQL, Oracle), ACID es un conjunto de propiedades que garantizan que las transacciones se manejen de forma segura, confiable y coherente.

Significado de ACID:

A – Atomicidad

* Una transacción debe ser todo o nada:  
  O se completan todas las operaciones, o ninguna se ejecuta.
* Si ocurre un error, se revierte todo lo que se haya hecho en esa transacción.

*Ejemplo:* Si transferís dinero de una cuenta a otra, el débito y el crédito deben ejecutarse juntos o ninguno.

C – Consistencia

* La transacción debe llevar la base de datos de un estado válido a otro válido, cumpliendo todas las reglas de integridad (como tipos de datos, claves foráneas, restricciones, etc.).

*Ejemplo:* No debería ser posible insertar una transacción con una cuenta que no existe.

I – Aislamiento

* Cada transacción debe ejecutarse como si fuera la única en el sistema, incluso si hay muchas ejecutándose al mismo tiempo.
* Evita errores como lecturas sucias o modificaciones no visibles.

*Ejemplo:* Dos usuarios no deben poder gastar el mismo saldo al mismo tiempo por interferencia entre transacciones.

D – Durabilidad

* Una vez que una transacción se confirma (commit), sus efectos permanecen en la base de datos, incluso si el sistema falla justo después.

*Ejemplo:* Si transferís dinero y se confirmó, ese cambio no se pierde aunque se apague el servidor.

ACID asegura que las transacciones en bases de datos sean seguras, coherentes, independientes y permanentes, incluso ante errores o fallos del sistema.

**8.- El sharding es una técnica clave en bases de datos NoSQL (como MongoDB, Cassandra, etc.) para dividir y distribuir los datos en varios servidores o nodos.**

¿Qué es el sharding? Es el proceso de fragmentar una base de datos grande en partes más pequeñas, llamadas shards, que se almacenan en diferentes servidores. Cada shard contiene una porción de los datos, y juntos forman la base completa.

¿Para qué se usa?

Para lograr:

* Escalabilidad horizontal: se agregan más servidores en lugar de mejorar uno solo.
* Mejor rendimiento: las consultas y escrituras se reparten entre múltiples nodos.
* Mayor disponibilidad: si un nodo falla, los demás pueden seguir funcionando.

Ejemplo:

Imaginá que tenés una base de datos de usuarios con millones de registros. En lugar de guardar todo en un solo servidor, podés aplicar sharding así:

* Shard 1: usuarios cuyo ID va del 1 al 1 millón.
* Shard 2: usuarios del 1 millón al 2 millones.
* Shard 3: usuarios del 2 millones en adelante.

Cada shard está en un servidor distinto y funciona en paralelo, lo que reduce la carga y aumenta la velocidad.

¿Cómo sabe el sistema dónde buscar?

Se utiliza un módulo de enrutamiento (router) que:

* Determina en qué shard están los datos según una clave de partición (por ejemplo, el ID del usuario).
* Redirige la consulta al shard correcto.

Ventajas del sharding:

* Escala fácilmente al agregar más máquinas.
* Mejora el rendimiento con datos grandes.
* Evita cuellos de botella en un solo servidor.

Desventajas:

* Mayor complejidad en la configuración y mantenimiento.
* Requiere una buena estrategia de particionado para evitar "shards calientes" (sobrecargados).

**9.- En MongoDB, un documento es la unidad básica de almacenamiento de datos, equivalente a una fila (registro) en una base de datos relacional.**

Un documento es un **objeto en formato BSON** (una extensión binaria de JSON), que almacena datos en **pares clave–valor**, como:

{

"\_id": 1,

"nombre": "Ana",

"edad": 30,

"email": "ana@gmail.com"

}

Características principales de los documentos en MongoDB:

1. Estructura flexible (esquema dinámico)

Cada documento puede tener diferentes campos y estructuras, a diferencia de las tablas SQL.

1. Se almacenan en colecciones

En lugar de "tablas", MongoDB usa colecciones, que contienen muchos documentos.

1. Incluyen un campo \_id único

MongoDB asigna automáticamente un campo \_id a cada documento si no lo especificás, que actúa como clave primaria.

1. Anidamiento y arrays

Los documentos pueden contener subdocumentos y arreglos, permitiendo representar estructuras complejas de datos.

Ejemplo con anidamiento:

{

"nombre": "Carlos",

"direccion": {

"calle": "Av. Siempre Viva",

"numero": 123

},

"telefonos": ["12345678", "98765432"]

}

Un documento en MongoDB es un registro autosuficiente, en formato JSON/BSON, que puede tener estructura flexible, contener datos complejos y es almacenado dentro de una colección.

**10.- Cuales son los tipos principales de base de datos nosql**

Claro, los principales tipos de bases de datos NoSQL son cuatro, cada uno pensado para distintos casos de uso y tipos de datos:

1. Bases de datos de documentos

* Almacenan datos en documentos JSON, BSON o XML.
* Cada documento es una unidad independiente con estructura flexible.
* Ideal para datos semiestructurados o cambiantes.
* Ejemplos: MongoDB, CouchDB.
* Características:
  + Almacenan datos en documentos tipo JSON, BSON, o XML.
  + Documentos con estructura flexible, cada uno puede tener diferentes campos.
  + Permiten anidamiento de documentos y arrays.
  + Buen soporte para consultas complejas sobre los datos del documento.
* Casos de uso:
  + Aplicaciones web y móviles que manejan datos semi-estructurados.
  + Sistemas donde el esquema cambia frecuentemente.
  + Catálogos de productos, perfiles de usuario, blogs, contenido dinámico.
  + Ejemplos: MongoDB, CouchDB.

2. Bases de datos clave-valor

* Almacenan pares clave-valor simples.
* Muy rápidas y escalables para acceso rápido a datos mediante la clave.
* Se usan para caches, sesiones, configuraciones.
* Ejemplos: Redis, DynamoDB, Riak.
* Características:
  + Almacenan datos como pares clave-valor simples.
  + Muy rápidas en lectura y escritura.
  + No tienen estructura para los valores, pueden ser cualquier dato binario.
  + Escalan fácilmente y tienen alta disponibilidad.
* Casos de uso:
  + Caches (memoria rápida para acelerar aplicaciones).
  + Sesiones de usuario en aplicaciones web.
  + Almacenamiento de configuraciones.
  + Sistemas que requieren accesos ultra-rápidos con clave conocida.
  + Ejemplos: Redis, DynamoDB, Riak.

3. Bases de datos en columnas (column-family)

* Organizan datos en filas y columnas, pero optimizadas para lectura y escritura masiva.
* Útiles para big data, análisis y sistemas distribuidos.
* Ejemplos: Cassandra, HBase.
* Características:
  + Organizan datos en filas y columnas, pero no como tablas relacionales estrictas.
  + Columnas agrupadas en familias que se almacenan juntas para optimizar acceso.
  + Buen rendimiento en consultas masivas y escritura en gran volumen.
  + Diseñadas para sistemas distribuidos y escalabilidad horizontal.
* Casos de uso:
  + Big Data y análisis en tiempo real.
  + Sistemas que requieren almacenar grandes cantidades de datos con consultas analíticas.
  + Sistemas de registro y monitoreo.
  + Ejemplos: Apache Cassandra, HBase.

4. Bases de datos de grafos

* Almacenan datos como nodos, aristas y propiedades para representar relaciones complejas.
* Ideales para redes sociales, recomendaciones, análisis de relaciones.
* Ejemplos: Neo4j, ArangoDB, JanusGraph.
* Características:
  + Almacenan datos en nodos y relaciones (aristas).
  + Optimizada para consultas que involucran relaciones y caminos entre datos.
  + Permite análisis complejo de redes sociales, rutas, recomendaciones.
* Casos de uso:
  + Redes sociales (amistades, seguidores).
  + Motores de recomendación (productos relacionados, amigos comunes).
  + Gestión de fraudes y detección de patrones.
  + Análisis de rutas y logística.
  + Ejemplos: Neo4j, ArangoDB, JanusGraph.

PARTE 2

### ****1. Crear una tabla llamada**** empleados

CREATE TABLE empleados (

id INT PRIMARY KEY,

nombre VARCHAR(100),

edad INT,

salario DECIMAL(10, 2)

);

### ****2. Insertar 3 registros en la tabla**** empleados

INSERT INTO empleados (id, nombre, edad, salario) VALUES

(1, 'Ana Pérez', 30, 60000.00),

(2, 'Carlos Gómez', 45, 45000.00),

(3, 'Alberto Díaz', 38, 52000.00);

### ****3. Consulta para obtener empleados con salario mayor a 50000****

SELECT \* FROM empleados

WHERE salario > 50000;

### ****4. Consulta para actualizar la edad de un empleado con**** id = 2

UPDATE empleados

SET edad = 46

WHERE id = 2;

### ****5. Consulta para eliminar un empleado con**** id = 3

DELETE FROM empleados

WHERE id = 3;

### ****6. Consulta que cuenta cuántos empleados hay en la tabla****

SELECT COUNT(\*) AS cantidad\_empleados

FROM empleados;

### ****7. Consulta que selecciona empleados cuyo nombre empieza con "A"****

SELECT \* FROM empleados

WHERE nombre LIKE 'A%';

### ****8. Crear una tabla**** departamentos

CREATE TABLE departamentos (

id INT PRIMARY KEY,

nombre VARCHAR(100),

ubicacion VARCHAR(100)

);

### ****9. Agregar una clave foránea a la tabla**** empleados

Primero se debe agregar una columna id\_departamento en empleados, y luego establecer la clave foránea:

ALTER TABLE empleados

ADD id\_departamento INT;

ALTER TABLE empleados

ADD CONSTRAINT fk\_departamento

FOREIGN KEY (id\_departamento)

REFERENCES departamentos(id);

### ****10. Consulta para obtener el salario promedio por departamento****

SELECT d.nombre AS departamento, AVG(e.salario) AS salario\_promedio

FROM empleados e

JOIN departamentos d ON e.id\_departamento = d.id

GROUP BY d.nombre;

### ****11. Consulta para buscar empleados con edad entre 25 y 40 años****

SELECT \* FROM empleados

WHERE edad BETWEEN 25 AND 40;

### ****12. Consulta que devuelve los 3 empleados con mejor salario****

SELECT \* FROM empleados

ORDER BY salario DESC

LIMIT 3;

PARTE 2 NOSQL

**1. Crear colección clientes e insertar 3 documentos**

db.clientes.insertMany([

{ nombre: "Juan", email: "juan@example.com", edad: 35 },

{ nombre: "Ana", email: "ana@example.com", edad: 28 },

{ nombre: "Carlos", email: "carlos@example.com", edad: 40 }

]);

**Consulta: obtener clientes mayores de 30 años**

db.clientes.find({ edad: { $gt: 30 } });

**2. Actualizar el email de un cliente con nombre = "Juan"**

db.clientes.updateOne(

{ nombre: "Juan" },

{ $set: { email: "nuevojuan@example.com" } }

);

**3. Eliminar un cliente con email = "test@example.com"**

db.clientes.deleteOne({ email: "test@example.com" });

**4. Insertar múltiples documentos en colección productos**

db.productos.insertMany([

{ nombre: "Televisor", categoria: "Electrónica", precio: 1200 },

{ nombre: "Celular", categoria: "Electrónica", precio: 800 },

{ nombre: "Silla", categoria: "Muebles", precio: 150 },

{ nombre: "Notebook", categoria: "Electrónica", precio: 2000 },

{ nombre: "Mesa", categoria: "Muebles", precio: 300 }

]);

**5. Obtener todos los productos de la categoría "Electrónica"**

db.productos.find({ categoria: "Electrónica" });

**6. Obtener productos cuyo precio sea mayor a 100**

db.productos.find({ precio: { $gt: 100 } });

**7. Crear colección empleados con validación: edad > 18**

db.createCollection("empleados", {

validator: {

$jsonSchema: {

bsonType: "object",

required: ["nombre", "edad"],

properties: {

nombre: {

bsonType: "string",

description: "Debe ser un string y es requerido"

},

edad: {

bsonType: "int",

minimum: 19,

description: "Debe ser un entero mayor a 18"

}

}

}

}

});

**8. Modificar múltiples documentos en clientes, agregando campo activo: true**

db.clientes.updateMany(

{},

{ $set: { activo: true } }

);

**9. ¿Cuándo usar MongoDB en lugar de SQL?**

Usarías **MongoDB** en lugar de **SQL** cuando:

* Los datos no tienen una estructura fija (por ejemplo, cada documento puede tener campos distintos).
* Se necesita escalar horizontalmente (más fácil con bases NoSQL).
* Querés alta velocidad para lecturas/escrituras sin requerir relaciones complejas.
* El proyecto requiere una evolución rápida del esquema (por ejemplo, apps móviles/web que cambian seguido).
* Trabajás con datos en formato JSON o estructuras anidadas (arrays, objetos).

**Ejemplo real:** una app de redes sociales con publicaciones, comentarios y reacciones que cambian de estructura fácilmente.

**10. Diferencia entre filter y agregación**

| **filter (find)** | **agregación (aggregate)** |
| --- | --- |
| Devuelve documentos que cumplen una condición | Permite transformar, agrupar, ordenar, contar, calcular y más |
| Simpler y más rápida para búsquedas directas | Más poderosa y flexible, útil para reportes y estadísticas complejas |
| Ejemplo: find({ edad:  { $gt: 30 } }) | Ejemplo: aggregate([{ $group: { \_id: "$categoria", total: { $sum: "$precio" } } }]) |

**PARTE 3**

**Conversión del modelo NoSQL a relacional (MySQL)**

En MongoDB, cada documento puede tener un arreglo anidado (como pedidos), pero en SQL necesitamos **separar los datos en tablas relacionadas** mediante **claves foráneas**.

**Modelo Relacional propuesto**

1. **Tabla clientes**
   * id\_cliente (PK)
   * nombre
   * email
2. **Tabla pedidos**
   * id\_pedido (PK)
   * total
   * id\_cliente (FK hacia clientes)

**Script SQL equivalente**

-- Crear tabla clientes

CREATE TABLE clientes (

id\_cliente INT PRIMARY KEY,

nombre VARCHAR(100),

email VARCHAR(100)

);

-- Crear tabla pedidos

CREATE TABLE pedidos (

id\_pedido INT PRIMARY KEY,

total DECIMAL(10,2),

id\_cliente INT,

FOREIGN KEY (id\_cliente) REFERENCES clientes(id\_cliente)

);

-- Insertar clientes

INSERT INTO clientes (id\_cliente, nombre, email) VALUES

(1, 'Juan', 'juan@example.com'),

(2, 'Maria', 'maria@example.com');

-- Insertar pedidos

INSERT INTO pedidos (id\_pedido, total, id\_cliente) VALUES

(1, 100.50, 1),

(2, 200.75, 2);

**PARTE 4**

**Mini-proyecto: Cada grupo debe modelar la base de datos para un sistema de reservas de vuelos en MongoDB usando una sola colección.**

**Cada documento representará una reserva e incluirá toda la información necesaria.**

**Requisitos:**

* **Pasajero: Nombre, contacto, identificación.**
* **Vuelo: Aerolínea, origen, destino, horario.**
* **Reserva: Código de reserva, estado (confirmada/cancelada), fecha de creación.**

**Preguntas obligatorias**

**1-Obtener todas las reservas confirmadas para vuelos cuyo destino sea "Madrid" y cuya fecha de vuelo sea posterior a "2025-06-01".**

**2-Contar cuántas reservas hay por aerolínea ($group).**

**3-Modificar solo las reservas con destino "París", asignando un descuento (nuevo campo).**

Modelo de documento (una sola colección)

Colección llamada reservas.

Cada documento representa una reserva completa:

{

"\_id": ObjectId("..."),

"codigo\_reserva": "ABC123",

"estado": "confirmada",

"fecha\_creacion": "2025-05-20",

"pasajero": {

"nombre": "Juan Pérez",

"contacto": "+54113456789",

"identificacion": "DNI 34567890"

},

"vuelo": {

"aerolinea": "Iberia",

"origen": "Buenos Aires",

"destino": "Madrid",

"horario": "2025-06-15T10:00:00Z"

}

}

**1. Obtener todas las reservas confirmadas con destino "Madrid" y vuelo posterior a "2025-06-01"**

db.reservas.find({

"estado": "confirmada",

"vuelo.destino": "Madrid",

"vuelo.horario": { $gt: new Date("2025-06-01") }

});

**2. Contar cuántas reservas hay por aerolínea ($group)**

db.reservas.aggregate([

{

$group: {

\_id: "$vuelo.aerolinea",

total\_reservas: { $sum: 1 }

}

}

]);

**3. Modificar solo las reservas con destino "París", asignando un nuevo campo descuento**

db.reservas.updateMany(

{ "vuelo.destino": "París" },

{ $set: { descuento: 0.25 } }

);