

LECCIÓN 1

Ejercicio 1

Comparativa RDBMS vs NoSQL

Contexto:

Para elegir una base de datos adecuada, es fundamental entender las diferencias entre los modelos relacional y NoSQL en función de sus características técnicas y sus casos de uso.

Paso a paso:

1. Lean la descripción del caso que les entregará el docente.
2. Identifiquen los requisitos clave (consistencia, volumen de datos, disponibilidad, velocidad, etc.).

Consigna:

3. Selecciónen el tipo de base de datos más apropiado (RDBMS o NoSQL).

En grupos, analicen un caso de negocio (ej. red social, banco digital o tienda online) y determinen si es más adecuado utilizar una base de datos relacional o NoSQL. Justifiquen su elección en función de consistencia, escalabilidad y disponibilidad.

4. Argumenten su decisión considerando los principios ACID o BASE y el Teorema de CAP.

Tiempo: 30 minutos

5. Expliquen cómo este tipo de base de datos resuelve los desafíos del caso.
6. Compartan sus conclusiones oralmente o mediante una breve presentación.

Ahora te paso el caso a analizar con estas instrucciones

Caso: FlashMarket

Contexto

Una startup chilena llamada **FlashMarket** ha lanzado una aplicación móvil para compras en supermercados y tiendas de barrio con entregas en menos de 30 minutos.

El sistema permite:

- Gestionar perfiles de usuario, direcciones y métodos de pago.
- Mostrar inventario actualizado en tiempo real por tienda.
- Registrar cientos de pedidos por minuto en horas punta.
- Funcionar establemente en varias ciudades simultáneamente.
- Mantener la experiencia del cliente incluso con desconexiones o latencias.

Los usuarios pueden:

- Consultar su historial de compras.
 - Recibir recomendaciones personalizadas según su ubicación y hábitos.
-

Solución propuesta extendida:

Consigna

Analizar este caso y determinar si es más adecuado utilizar una base de datos relacional (RDBMS) o NoSQL. Justificar la elección considerando:

- Consistencia
- Escalabilidad
- Disponibilidad

También incluir fundamentos técnicos sobre:

- Principios ACID o BASE
- Teorema CAP
- Cómo la base de datos elegida resuelve los desafíos del caso

1. Análisis técnico del caso

Requisito del sistema	¿Qué necesita?	¿Qué tipo de BD lo resuelve mejor?
Pedidos por minuto muy altos	Escalabilidad horizontal	NoSQL
Operación en varias ciudades	Tolerancia a particiones, velocidad	NoSQL
Múltiples fuentes y datos cambiantes	Estructura flexible	NoSQL
Pagos y seguridad	Transacciones confiables	RDBMS (ACID)
Recomendaciones personalizadas	Lecturas rápidas, datos semiestructurados	NoSQL
Inventario actualizado en tiempo real	Alta velocidad y disponibilidad	NoSQL (BASE)

2. Fundamentos teóricos aplicados

A. Teorema CAP

Una base de datos distribuida solo puede garantizar **2 de los 3** aspectos al mismo tiempo:

- **C: Consistency** → Todos los nodos ven los mismos datos al mismo tiempo
- **A: Availability** → El sistema siempre responde (aunque con datos antiguos)
- **P: Partition Tolerance** → El sistema sigue funcionando aun si hay fallas de red

FlashMarket prioriza:

- ✓ Availability
- ✓ Partition Tolerance
- ✗ Sacrifica algo de Consistency → Se puede usar **consistencia eventual**

B. ACID vs BASE

ACID (Relacional)	BASE (NoSQL)
Atomicidad, Consistencia, Aislamiento, Durabilidad	Básicamente Disponible, Estado blando, Consistencia eventual
Ideal para pagos y operaciones bancarias	Ideal para apps distribuidas como FlashMarket
Preciso, pero más lento	Flexible y rápido

FlashMarket necesita ambos:

- ACID → módulo de pagos
- BASE → inventario, recomendaciones, historial de compras

3. Recomendación de arquitectura

Arquitectura híbrida

Módulo	Tipo de base de datos	Ejemplo
Usuarios y pagos	Relacional (ACID)	PostgreSQL, MySQL
Inventario y recomendaciones	NoSQL (BASE)	MongoDB, Cassandra
Historial de compras	NoSQL	DynamoDB, MongoDB

4. Ventajas de usar NoSQL en FlashMarket

- Escalabilidad horizontal → Soporta muchas solicitudes por segundo
- Alta disponibilidad → La app responde incluso con caídas de red
- Flexibilidad → Se adapta a estructuras cambiantes sin redefinir esquema
- Rendimiento → Ideal para lecturas rápidas como recomendaciones o stock

5. Cómo resuelve la base de datos los desafíos del negocio

Desafío	Solución con NoSQL/RDBMS
Operar con muchos pedidos simultáneos	NoSQL escalable horizontalmente
Datos semi-estructurados como hábitos	NoSQL flexible como MongoDB
Fallas de red o lentitud	NoSQL tolerante a particiones (CAP)
Pagos y seguridad	RDBMS con transacciones ACID
Recomendaciones personalizadas	NoSQL con consultas rápidas (Redis o MongoDB)

Conclusión final

- **NoSQL** es la base adecuada para FlashMarket por su distribución, escalabilidad y disponibilidad.
- **RDBMS** sigue siendo útil para tareas críticas como pagos, donde se necesita consistencia fuerte.
- La **mejor solución es una arquitectura híbrida**, donde se combinan ambas tecnologías según el tipo de dato y requerimiento del sistema.

Ejercicio 2

MAPA VISUAL DEL TEOREMA CAP

Contexto:

Comprender el Teorema de CAP es clave para diseñar soluciones distribuidas con bases de datos modernas.

Paso a paso:

1. Repasa la definición del Teorema de CAP.
2. Define cada propiedad (C, A, P) con un ejemplo simple.

Consigna: 40

Crea un mapa visual (diagrama, Infografía o esquema) que explique el Teorema de CAP y ubique diferentes tecnologías de bases de datos según sus prioridades (CP, CA, AP).

Tiempo: 30 minutos

3. Ubica al menos 4 tecnologías reales (MongoDB, Cassandra, PostgreSQL, Redis) en el triángulo de CAP según lo aprendido.
4. Opcional: destaca qué tipo de casos de uso prioriza cada combinación.

5. Comparte tu mapa con la clase (puede ser físico o digital).

Mapa Visual del Teorema de CAP

El Teorema de CAP establece que en un sistema distribuido **solo se puede garantizar dos de estas tres propiedades al mismo tiempo**:

Letra	Nombre Completo	¿Qué significa?
C	Consistency	Todos los nodos ven los mismos datos al mismo tiempo.
A	Availability	El sistema siempre responde, aunque algunos datos estén desactualizados.
P	Partition Tolerance	El sistema sigue funcionando incluso con fallos de red o particiones.

Combinaciones posibles:

1. CP (Consistency + Partition Tolerance)

- ✓ Garantiza consistencia aunque haya fallos de red.
- ✗ Puede no estar disponible momentáneamente.
- Ejemplos: PostgreSQL en clúster, HBase.

2. AP (Availability + Partition Tolerance)

- ✓ Siempre responde y es tolerante a fallos.
- ✗ Puede haber diferencias temporales en los datos entre nodos.
- Ejemplos: Cassandra, CouchDB.

3. CA (Consistency + Availability) – *Solo si no hay particiones de red*

- ✓ Alta disponibilidad y consistencia.
- ✗ No funciona bien si hay problemas de red (no tolera particiones).
- Ejemplos: Redis (modo standalone), MySQL en nodo único.

Ubicación de Tecnologías Reales en el Triángulo CAP

Tecnología	Clasificación CAP	Observaciones
MongoDB	AP	Consistencia eventual, alta disponibilidad.
Cassandra	AP	Muy tolerante a fallos, escalable.
PostgreSQL	CP	Fuerte consistencia, menos tolerante a fallos.
Redis	CA/CP	CA en modo simple, CP en modo clúster.

Casos de Uso por Combinación

- CP:** Bancos, pagos, sistemas financieros.
- AP:** Redes sociales, chats, streaming.
- CA:** Aplicaciones internas o con red muy estable.

Notas

El Teorema de CAP no implica perder una propiedad para siempre.

Solo durante una partición de red Se debe priorizar 2 de las 3.

Es una guía de diseño para sistemas distribuidos eficientes.

✓ LECCIÓN 2

Ejercicio 1

Diagnóstico y mejora de una consulta lenta

Contexto:

Una consulta SQL ineficiente puede comprometer el rendimiento de toda una aplicación. Este ejercicio busca que los estudiantes apliquen buenas prácticas para optimizar una consulta.

Objetivo:

Optimizar una consulta SQL mal diseñada para mejorar su rendimiento mediante análisis y reestructuración.

Paso a paso:

1. Lee y comprende la consulta entregada por el docente.
 2. Analiza qué recursos puede estar consumiendo en exceso.
 3. Aplica herramientas como EXPLAIN o EXPLAIN ANALYZE para visualizar el plan de ejecución.
 4. Redacta una versión optimizada y explica cada cambio.
 5. Comparte tus mejoras y justificaciones con el grupo o en el plenario.
-

Tiempo estimado: 30 minutos

Caso de estudio:

La empresa **TechStore** desea analizar los pedidos realizados por sus clientes durante el año 2023. En particular, el equipo de ventas necesita un informe que muestre todos los pedidos que cumplan con las siguientes condiciones:

- El pedido debe haber sido realizado por el cliente llamado **Juan Pérez**.
 - La fecha del pedido debe estar dentro del año **2023**.
 - El estado del pedido debe ser "**entregado**".
 - El producto comprado debe pertenecer a una categoría que contenga la palabra "**tecnología**" (por ejemplo: *Tecnología*, *Tecnología WiFi*, etc.).
-

Estructura de tablas y datos:

Tabla `clientes`:

```
CREATE TABLE clientes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100),
    correo VARCHAR(100),
    telefono VARCHAR(20)
);
```

Tabla `productos`:

```
CREATE TABLE productos (
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```
nombre VARCHAR(100),
categoria VARCHAR(100),
precio DECIMAL(10, 2)
);
```

Tabla pedidos:

```
CREATE TABLE pedidos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    cliente_id INT,
    producto_id INT,
    fecha DATE,
    estado VARCHAR(50),
    cantidad INT,
    FOREIGN KEY (cliente_id) REFERENCES clientes(id),
    FOREIGN KEY (producto_id) REFERENCES productos(id)
);
```

Datos de ejemplo:

```
-- Clientes
INSERT INTO clientes (nombre, correo, telefono) VALUES
('Juan Pérez', 'juan.perez@email.com', '987654321'),
('María López', 'maria.lopez@email.com', '912345678'),
('Carlos Núñez', 'carlos.nunez@email.com', '923456789');

-- Productos
INSERT INTO productos (nombre, categoria, precio) VALUES
('Notebook Lenovo', 'Tecnología', 850000),
('Celular Samsung', 'Tecnología', 650000),
('Silla Ergonómica', 'Oficina', 120000),
('Router TP-Link', 'Tecnología WiFi', 45000),
('Monitor LG 24"', 'Tecnología', 180000);

-- Pedidos
INSERT INTO pedidos (cliente_id, producto_id, fecha, estado, cantidad) VALUES
(1, 1, '2023-01-15', 'entregado', 1),
(1, 2, '2023-02-20', 'pendiente', 2),
(2, 3, '2023-03-10', 'entregado', 1),
(3, 4, '2022-12-05', 'entregado', 1),
(1, 5, '2023-05-22', 'entregado', 1),
(2, 1, '2023-07-01', 'entregado', 1),
(3, 2, '2023-08-12', 'cancelado', 1);
```

Consulta original (no optimizada):

```
SELECT *
FROM pedidos
JOIN clientes ON pedidos.cliente_id = clientes.id
JOIN productos ON pedidos.producto_id = productos.id
WHERE UPPER(clientes.nombre) = 'JUAN PÉREZ'
    AND YEAR(pedidos.fecha) = 2023
```

```
AND productos.categoría LIKE '%tecnología%'  
AND pedidos.estado = 'entregado';
```

Solución: Diagnóstico y mejora de la consulta

Problemas detectados:

1. SELECT * trae todas las columnas innecesariamente.
2. UPPER(clientes.nombre) impide el uso de índices.
3. YEAR(pedidos.fecha) también impide el uso de índices.
4. LIKE '%tecnología%' fuerza un escaneo completo, no puede usar índice.
5. Posible falta de índices en columnas filtradas.

Propuestas de mejora:

1. Evitar funciones sobre columnas:

- Reemplazar UPPER(nombre) por comparación directa estandarizando datos en minúsculas.

2. Reemplazar YEAR.fecha por rango:

```
pedidos.fecha BETWEEN '2023-01-01' AND '2023-12-31'
```

3. Evitar SELECT *:

- Seleccionar solo las columnas necesarias mejora rendimiento y claridad.

4. Crear índices en campos clave:

```
CREATE INDEX idx_clientes_nombre ON clientes(nombre);  
CREATE INDEX idx_pedidos_fecha ON pedidos(fecha);  
CREATE INDEX idx_pedidos_estado ON pedidos(estado);  
CREATE INDEX idx_productos_categoria ON productos(categoría);
```

Consulta optimizada:

```
SELECT pedidos.id, pedidos.fecha, pedidos.estado, pedidos.cantidad,  
       clientes.nombre, productos.nombre AS producto, productos.categoría  
  FROM pedidos  
 JOIN clientes ON pedidos.cliente_id = clientes.id  
 JOIN productos ON pedidos.producto_id = productos.id  
 WHERE clientes.nombre = 'Juan Pérez'  
   AND pedidos.fecha BETWEEN '2023-01-01' AND '2023-12-31'  
   AND productos.categoría LIKE '%Tecnología%'  
   AND pedidos.estado = 'entregado';
```

Uso de EXPLAIN para verificar mejoras:

```
EXPLAIN SELECT pedidos.id, pedidos.fecha, pedidos.estado, pedidos.cantidad,  
       clientes.nombre, productos.nombre AS producto, productos.categoría  
  FROM pedidos
```

```
JOIN clientes ON pedidos.cliente_id = clientes.id
JOIN productos ON pedidos.producto_id = productos.id
WHERE clientes.nombre = 'Juan Pérez'
    AND pedidos.fecha BETWEEN '2023-01-01' AND '2023-12-31'
    AND productos.categoría LIKE '%Tecnología%'
    AND pedidos.estado = 'entregado';
```

Esto mostrará el plan de ejecución y permitirá verificar si los índices están siendo utilizados correctamente.

Ejercicio 2

Actividad: Diseño de un esquema eficiente con vistas e índices

Tema: Optimización de consultas en una base de datos relacional (RDBMS)

Duración estimada: 30 minutos

Contexto

El diseño del esquema de base de datos afecta directamente el rendimiento de las consultas, especialmente en aplicaciones que manejan grandes volúmenes de información. Un diseño bien estructurado, con vistas e índices adecuados, permite acelerar las consultas más frecuentes y reducir la carga del sistema.

Consigna

Diseña una estructura de base de datos simple para una aplicación de ecommerce que permita consultas eficientes sobre ventas, productos y usuarios. Incluye al menos **una vista y dos índices** justificados.

Tiempo: 30 minutos

Paso a paso:

1. Piensa qué entidades necesita tu esquema (por ejemplo: usuarios, productos, órdenes).
 2. Define las claves primarias y foráneas necesarias.
 3. Elige qué columnas indexar para mejorar consultas típicas (por fecha, por usuario, etc.).
 4. Diseña una vista que facilite un análisis de ventas por mes.
 5. Justifica por qué tu diseño mejora el rendimiento.
-

Paso a paso

1. Entidades necesarias

- **usuarios:** almacena los datos de los clientes.
 - **productos:** contiene los detalles de los productos disponibles.
 - **órdenes:** almacena las órdenes de compra realizadas por los usuarios.
 - **detalle_orden:** tabla intermedia que relaciona órdenes con productos comprados.
-

Esquema propuesto

```
```sql CREATE TABLE usuarios ( id_usuario INT PRIMARY KEY, nombre TEXT, email TEXT UNIQUE, fecha_registro DATE );
```

```
CREATE TABLE productos (id_producto INT PRIMARY KEY, nombre TEXT, precio DECIMAL, categoria TEXT);
CREATE TABLE ordenes (id_orden INT PRIMARY KEY, id_usuario INT, fecha DATE, total DECIMAL, FOREIGN KEY (id_usuario) REFERENCES usuarios(id_usuario));
CREATE TABLE detalle_orden (id_detalle INT PRIMARY KEY, id_orden INT, id_producto INT, cantidad INT, subtotal DECIMAL, FOREIGN KEY (id_orden) REFERENCES ordenes(id_orden), FOREIGN KEY (id_producto) REFERENCES productos(id_producto));
```

### 3. Índices propuestos Índice 1: sobre la fecha en la tabla de órdenes

Para acelerar búsquedas y agrupaciones por período de tiempo.

sql Copiar código CREATE INDEX idx\_ordenes\_fecha ON ordenes(fecha); Índice 2: sobre el id\_usuario en órdenes

Permite encontrar rápidamente todas las compras de un usuario específico.

CREATE INDEX idx\_ordenes\_usuario ON ordenes(id\_usuario);

### Índice 2: sobre el id\_usuario en órdenes

Permite encontrar rápidamente todas las compras de un usuario específico.

sql Copiar código CREATE INDEX idx\_ordenes\_usuario ON ordenes(id\_usuario);

### 4. Vista para análisis mensual de ventas Agrupa las ventas por mes y año, mostrando el total acumulado.

sql Copiar código CREATE VIEW ventas\_mensuales AS SELECT EXTRACT(YEAR FROM fecha) AS anio, EXTRACT(MONTH FROM fecha) AS mes, SUM(total) AS total\_ventas FROM ordenes GROUP BY anio, mes ORDER BY anio, mes;

### 5. Justificación del diseño La separación en entidades normalizadas evita redundancias y mejora la integridad de los datos.

Los índices propuestos están orientados a las consultas más comunes:

Buscar órdenes por usuario

Filtrar o agrupar órdenes por fecha

La vista ventas\_mensuales permite realizar reportes sin recalcular las ventas desde cero cada vez, mejorando el tiempo de respuesta en paneles y dashboards.

El uso de claves foráneas asegura la coherencia referencial entre los datos.

Este diseño es escalable y adecuado para manejar grandes volúmenes de datos en un sistema de ecommerce, optimizando las consultas más críticas para el negocio.

## LECCIÓN 3

### ▼ Ejercicio 1

Match entre tipo de NoSQL y caso de uso real

Contexto Comprender cuál es el tipo de base de datos NoSQL más adecuado según el problema o aplicación es clave para tomar decisiones de diseño efectivas, sobre todo en entornos con grandes volúmenes de datos, estructuras flexibles o necesidades de alta disponibilidad.

Consigna Relaciona diferentes tipos de bases de datos NoSQL con escenarios reales de uso. Luego justifica tu elección y describe qué ventajas aporta ese modelo en particular.

Paso a paso

En grupos, analicen los 4 casos de uso presentados más abajo.

Seleccione para cada caso el tipo de base de datos NoSQL más adecuado:

Clave-valor

Documental

Columnar

Grafos

En memoria

Justifiquen su elección considerando:

Estructura de los datos

Escalabilidad

Rendimiento

Flexibilidad del esquema

Expongan sus decisiones en una breve puesta en común o en un muro colaborativo.

Tiempo estimado: 30 minutos

## Casos de uso

Caso 1: Plataforma de mensajería en tiempo real para equipos de trabajo Una app de colaboración (como Slack) permite a usuarios:

Enviar mensajes instantáneos

Compartir archivos

Crear canales por proyecto

Características clave:

Alta demanda de escritura

Baja latencia

Almacenamiento flexible de mensajes (mensajes, adjuntos, reacciones, menciones)

Caso 2: Sistema de recomendaciones para una plataforma de streaming Una app de películas/series recomienda contenido personalizado en base a:

Historial de visualización

Preferencias de género

Conexiones sociales entre usuarios

Características clave:

Relaciones complejas entre usuarios y contenido

Navegación eficiente entre nodos

Análisis de afinidad

Caso 3: Gestión documental para una clínica veterinaria Una clínica almacena fichas médicas de cada mascota, incluyendo:

Datos clínicos

Recetas

Vacunaciones

Visitas

Evolución médica

Características clave:

Estructura flexible por paciente

Acceso rápido por ID o nombre

Necesidad de actualizar sin migraciones estructurales

Caso 4: Aplicación de cupones y promociones geolocalizadas Una app envía cupones personalizados según:

Ubicación del usuario

Preferencias de compra

Características clave:

Búsquedas en memoria

Consultas por coordenadas

Baja latencia

Alta escalabilidad para eventos masivos (Black Friday, Navidad)

Match entre tipo de base de datos NoSQL y caso de uso real

## Caso 1: Plataforma de mensajería en tiempo real para equipos de trabajo

Tipo de base de datos NoSQL recomendado:

Base de datos en memoria (ej. Redis)

Justificación:

Se requiere baja latencia (respuesta en milisegundos).

Se producen altas tasas de escritura (mensajes constantes entre usuarios).

Redis permite almacenar mensajes recientes en memoria y luego persistir en disco si se desea.

Soporta estructuras flexibles: listas, hashes, sets, streams (para chats en tiempo real).

Alta escalabilidad horizontal para soportar miles de usuarios simultáneos.

Ventajas del modelo en memoria:

Respuesta inmediata.

Muy buen rendimiento en lectura y escritura concurrente.

Ideal para sesiones activas, chats y datos temporales.

## Caso 2: Sistema de recomendaciones para una plataforma de streaming

Tipo de base de datos NoSQL recomendado:

Base de datos de grafos (ej. Neo4j)

Justificación:

Hay relaciones complejas: usuario → género → contenido → popularidad.

Se necesita navegar fácilmente entre nodos y relaciones.

Ideal para calcular afinidades, sugerencias y caminos entre elementos.

Ventajas del modelo de grafos:

Las consultas sobre relaciones profundas son rápidas y naturales.

No requiere JOINs costosos.

Permite personalizar recomendaciones a partir de conexiones reales en los datos.

### Caso 3: Gestión documental para una clínica veterinaria

Tipo de base de datos NoSQL recomendado:

Base de datos documental (ej. MongoDB)

Justificación:

Cada mascota puede tener una estructura de ficha distinta (especie, tratamientos, alergias).

Se requiere flexibilidad de esquema, sin necesidad de migraciones complejas.

Acceso frecuente por ID (mascota, historia clínica), lo cual es muy eficiente en bases documentales.

Ventajas del modelo documental:

Permite almacenar documentos JSON con estructuras variadas.

Se puede actualizar un solo campo sin afectar todo el modelo.

Fácil de escalar y replicar.

### Caso 4: Aplicación de cupones y promociones geolocalizadas

Tipo de base de datos NoSQL recomendado:

Base de datos clave-valor o en memoria (ej. Redis con módulo de geolocalización)

Justificación:

Se necesita respuesta en tiempo real y búsquedas por ubicación.

Redis permite usar comandos como GEOSEARCH para localizar puntos cercanos.

Alta concurrencia en fechas pico como Black Friday.

Ventajas del modelo en memoria / clave-valor:

Muy baja latencia.

Eficiente en búsquedas simples por ID o geolocalización.

Escalable y distribuible fácilmente.

Resumen visual del match

1 En memoria Redis Velocidad, baja latencia, alta escritura

2 Grafos Neo4j Relaciones complejas entre usuarios y datos

3 Documental MongoDB Estructura flexible por paciente

4 Clave-valor / En memoria Redis Geolocalización, velocidad, escalabilidad

## ✓ Ejercicio 2:

Diseña tu primera base NoSQL

# Consigna

## Contexto

Aplicar lo aprendido sobre estructura y tipos de NoSQL a un caso práctico ayuda a fijar conceptos y desarrollar criterio técnico para seleccionar el modelo más adecuado según el tipo de aplicación.

## Objetivo

Diseñar una base de datos NoSQL para una aplicación de mensajería en tiempo real (tipo WhatsApp), considerando qué tipo de NoSQL es más adecuado, cómo se modelan los datos y cómo escalar el sistema.

## Instrucciones

1. Determinen qué tipo de base de datos NoSQL usarían (por ejemplo: clave-valor, documental, grafos, columnas o en memoria).
2. Esbocen un modelo de datos simplificado que incluya:
  - Colecciones (en bases documentales)
  - Claves y valores (en bases clave-valor)
  - Nodos y aristas (en bases de grafos)
3. Expliquen cómo manejarían la escalabilidad del sistema:
  - Horizontal (agregar más servidores)
  - Vertical (mejorar recursos del servidor actual)
4. Describan cómo cada decisión resuelve los desafíos técnicos del caso:
  - Velocidad
  - Flexibilidad de estructura
  - Volumen de usuarios y mensajes
  - Comunicación en tiempo real

**Tiempo estimado:** 30 minutos **Nota:** No es necesario mencionar marcas específicas de bases de datos.

---

## Propuesta de solución

### Tipo de base de datos NoSQL elegido

Se elige una combinación de dos tipos de NoSQL:

- **Documental:** para almacenar usuarios, chats y mensajes con estructuras flexibles.
  - **En memoria:** para el manejo en tiempo real de usuarios conectados, mensajes no leídos y notificaciones instantáneas.
- 

### Modelo de datos simplificado

#### Colección: usuarios

```
{
 "_id": "usuario123",
 "nombre": "Ana Pérez",
 "telefono": "+56912345678",
 "estado": "Conectado",
 "contactos": ["usuario456", "usuario789"]
}
```

## Colección: chats

```
{
 "_id": "chat456",
 "tipo": "privado",
 "miembros": ["usuario123", "usuario456"],
 "creado": "2025-07-05T10:00:00Z"
}
```

## Colección: mensajes

```
{
 "_id": "msg789",
 "chat_id": "chat456",
 "emisor_id": "usuario123",
 "contenido": "Hola, ¿cómo estás?",
 "fecha": "2025-07-05T10:01:30Z",
 "leido": false
}
```

## En memoria (estructura complementaria)

- Lista de usuarios conectados y su último acceso.
- Mensajes no leídos por usuario.
- Contadores de notificaciones por chat.

## Estrategia de escalabilidad

### Escalado horizontal

- Dividir la base por zonas geográficas o por grupos de usuarios (sharding).
- Distribuir mensajes y chats por clústeres independientes.
- Usar平衡adores de carga para distribuir el tráfico.

### Escalado vertical (complementario)

- Incrementar RAM y CPU en nodos con alta demanda (por ejemplo, los de cache/mensajería activa).
- Reforzar servidores de base en memoria para responder a millones de usuarios concurrentes.

## Cómo resuelve los desafíos del caso

Desafío técnico	Solución aplicada
Alta velocidad de escritura	Inserciones rápidas en base documental, sin JOINs
Estructura flexible de mensajes	Colecciones JSON permiten cambios sin migraciones
Estado en tiempo real	Base en memoria para presencia y notificaciones
Escalabilidad por cantidad de usuarios	Sharding horizontal en usuarios y mensajes
Separación de datos por entidad	Colecciones distintas con relación por ID

## Reflexiones lección2

### Evaluación de aprendizajes sobre NoSQL\*\*

Durante estas actividades prácticas, tuve la oportunidad de aplicar y consolidar mis conocimientos sobre bases de datos NoSQL a través de diferentes escenarios del mundo real. Cada ejercicio me permitió comprender no solo los conceptos

teóricos, sino también la lógica detrás de las decisiones de diseño que se toman en contextos de alta demanda, flexibilidad de datos y escalabilidad.

En el primer ejercicio, el **match entre tipos de NoSQL y casos de uso reales**, aprendí a distinguir con mayor claridad cuándo conviene usar una base documental, una clave-valor, una base en memoria o una de grafos. Por ejemplo, comprendí que para una app de mensajería en tiempo real es clave la velocidad y por eso se opta por bases en memoria, mientras que para un sistema de recomendaciones, donde hay muchas relaciones entre usuarios y contenido, lo más adecuado es una base de grafos. Esta actividad me ayudó a desarrollar criterio técnico para elegir la herramienta correcta según el problema.

Luego, al **diseñar mi propia base de datos NoSQL para una app tipo WhatsApp**, pude aplicar estos aprendizajes para construir un modelo de datos simplificado, considerando colecciones, claves, relaciones y estrategias de escalado. Aprendí que las bases documentales permiten trabajar con estructuras dinámicas, algo ideal cuando cada mensaje, chat o usuario puede tener variaciones, y que una base en memoria puede complementar este modelo para manejar estados en tiempo real como conexiones activas o mensajes no leídos.

Otro aprendizaje clave fue reflexionar sobre la **escalabilidad**, diferenciando entre escalado horizontal y vertical. Comprendí que en proyectos de alto tráfico, escalar horizontalmente –es decir, distribuir los datos en varios servidores– es más eficiente que simplemente aumentar la capacidad de un solo servidor.

Finalmente, estas actividades me enseñaron que el diseño de una base de datos no depende solo de la teoría, sino también de una lectura profunda del contexto, del volumen de datos, de la velocidad esperada y del tipo de interacción entre usuarios y la información.

En resumen, este conjunto de ejercicios me permitió:

- Entender las diferencias entre los principales tipos de NoSQL.
- Justificar técnicamente la elección de un modelo de datos.
- Diseñar una solución aplicable, eficiente y escalable.
- Ver cómo se combinan teoría y práctica en el diseño de sistemas reales.

Este proceso fortaleció mis habilidades para tomar decisiones técnicas informadas y para pensar en soluciones más allá del modelo relacional tradicional, usando herramientas NoSQL que se adaptan mejor a los desafíos modernos en tiempo real, big data y estructuras flexibles.

## LECCION 4

### Ejercicio 1

Actividad: Simulación de diseño y operación de un clúster Cassandra

#### Objetivo

Comprender cómo se estructura Cassandra y cómo se define un keyspace es clave para operar correctamente en entornos distribuidos. Esta actividad busca aplicar conceptos como replicación, escalabilidad y organización de datos.

#### Consigna (40 puntos)

Diseña un esquema básico de clúster Cassandra para una app de streaming global. Incluye keyspaces, nodos, estrategia de replicación y tipos de consultas frecuentes.

**Tiempo estimado:** 30 minutos

#### Instrucciones paso a paso

1. Definan el número de centros de datos y nodos por región (mínimo 2 regiones).

2. Selecionen la estrategia de replicación adecuada (NetworkTopologyStrategy).
3. Propongan un keyspace y una tabla que registre visualizaciones de contenido (usuario, episodio, timestamp).
4. Escriban al menos dos consultas en CQL optimizadas para ese modelo.
5. Expongan brevemente su diseño y justificación al grupo.

## Solución propuesta:

### Diseño de Clúster Cassandra para App de Streaming Global

#### 1. Diseño de infraestructura distribuida

- **Región 1:** Norteamérica (DC1) – 3 nodos
- **Región 2:** Europa (DC2) – 3 nodos

Este diseño distribuye los datos entre dos regiones para mejorar la latencia y asegurar alta disponibilidad ante fallas regionales. Cada centro de datos tiene suficiente replicación para tolerancia local.

#### 2. Estrategia de replicación

Se utilizará NetworkTopologyStrategy , adecuada para clústeres multi-región. Se configuran dos keyspaces con distintas estrategias según el tipo de dato:

##### Keyspace 1: streaming\_usuarios

```
CREATE KEYSPACE streaming_usuarios
WITH replication = {
 'class': 'NetworkTopologyStrategy',
 'DC1': 3,
 'DC2': 3
};
```

Este keyspace guarda datos críticos de actividad de usuario, por lo que se replica completamente en ambas regiones.

##### Keyspace 2: streaming\_catalogo

```
CREATE KEYSPACE streaming_catalogo
WITH replication = {
 'class': 'NetworkTopologyStrategy',
 'DC1': 2,
 'DC2': 2
};
```

Este keyspace guarda el catálogo de contenido. Se replica dos veces por región para balance entre disponibilidad y eficiencia de espacio.

3. Modelado de tablas En streaming\_usuarios: USE streaming\_usuarios;

```
CREATE TABLE visualizaciones (
 user_id UUID,
 episodio_id UUID,
 timestamp TIMESTAMP,
```

```
duracion_segundos INT,
PRIMARY KEY (user_id, timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

Diseño orientado a consultar rápidamente las visualizaciones recientes de un usuario.

En streaming\_catalogo: USE streaming\_catalogo;

```
CREATE TABLE episodios (
 episodio_id UUID PRIMARY KEY,
 titulo TEXT,
 serie TEXT,
 duracion INT,
 genero TEXT
);
```

Diseño simple y eficiente para recuperar episodios directamente por ID.

4. Consultas CQL optimizadas Consulta 1: Obtener los últimos 10 episodios vistos por un usuario

```
SELECT episodio_id, timestamp, duracion_segundos
FROM visualizaciones
WHERE user_id = 1b2c-3d4e-567f-89gh
LIMIT 10;
```

Consulta eficiente gracias al uso de user\_id como clave de partición y timestamp como columna de orden.

Consulta 2: Contar cuántos episodios vio un usuario en las últimas 24 horas

```
SELECT COUNT(*)
FROM visualizaciones
WHERE user_id = 1b2c-3d4e-567f-89gh
AND timestamp > '2025-07-07 00:00:00';
```

Ideal para estadísticas rápidas de actividad reciente, manteniéndose dentro de una sola partición.

5. Justificación general Se diseñaron dos keyspaces con estrategias de replicación diferenciadas para cubrir distintas necesidades: alta criticidad en datos de usuario y eficiencia en datos estáticos del catálogo. Las tablas fueron modeladas según el tipo de consultas requeridas, priorizando accesos rápidos y escalabilidad. El uso de NetworkTopologyStrategy permite adaptación futura a nuevas regiones o cambios en la carga.

## Ejercicio 2

### ✓ Análisis de rendimiento de operaciones CRUD en Cassandra

#### Objetivo

Analizar cómo el diseño de claves primarias, uso de índices y estructura de tablas afecta el rendimiento de operaciones CRUD (Create, Read, Update, Delete) en Cassandra.

#### Consigna

1. Analiza un conjunto de operaciones CRUD y determina cuáles están correctamente diseñadas para Cassandra.

2. Identifica si usan claves de partición adecuadas, si son eficientes o si afectan negativamente al rendimiento.
  3. Propón mejoras para cada caso (reorganización de tabla, uso o eliminación de índices, ajuste en clave primaria, etc.).
  4. Comparte tus conclusiones con otra dupla.
- 

## Estructura de la tabla base

```
CREATE TABLE reproducciones (
 id_usuario UUID,
 id_episodio UUID,
 timestamp TIMESTAMP,
 dispositivo TEXT,
 PRIMARY KEY ((id_usuario), timestamp)
);
```

Clave de partición: id\_usuario

Clustering column: timestamp

Operaciones CRUD

### Caso 1: Inserción

Registrar que el usuario con ID u001 vio el episodio e321 desde un Smart TV el 5 de julio de 2025.

```
```sql
INSERT INTO reproducciones (
    id_usuario, id_episodio, timestamp, dispositivo
)
VALUES (
    'u001', 'e321', '2025-07-05 19:30:00', 'Smart TV'
);
```

Caso 2: Lectura

Obtener todas las visualizaciones del episodio e321.

```
SELECT * FROM reproducciones
WHERE id_episodio = 321e4567-e89b-12d3-a456-426614174999;
```

Caso 3: Actualización

Actualizar el tipo de dispositivo desde el que el usuario u001 vio el episodio e321 el 5 de julio.

```
UPDATE reproducciones
SET dispositivo = 'Tablet'
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000;
```

Caso 4: Eliminación Eliminar el registro de visualización de u001 en la fecha indicada.

```
DELETE FROM reproducciones
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000;
```

Solución propuesta:

Caso 1: Inserción

Registrar que el usuario con ID u001 vio el episodio e321 desde un Smart TV el 5 de julio de 2025.

```
INSERT INTO reproducciones (
    id_usuario, id_episodio, timestamp, dispositivo
)
VALUES (
    'u001', 'e321', '2025-07-05 19:30:00', 'Smart TV'
);
```

Análisis: Consulta correcta y eficiente. Usa correctamente la clave primaria.

Caso 1: Inserción Consulta:

```
sql Copiar código INSERT INTO reproducciones ( id_usuario, id_episodio, timestamp, dispositivo ) VALUES ( 'u001', 'e321', '2025-07-05 19:30:00', 'Smart TV' );
```

Análisis:

Correcta y eficiente.

Usa correctamente la clave de partición (id_usuario) y clustering (timestamp).

Aprovecha bien el modelo de escritura rápida de Cassandra.

No se requieren mejoras.

Caso 2: Lectura

Obtener todas las visualizaciones del episodio e321.

```
SELECT * FROM reproducciones
WHERE id_episodio = 321e4567-e89b-12d3-a456-426614174999;
```

Análisis: Ineficiente y no válida en Cassandra. id_episodio no es clave primaria ni índice. No se puede consultar así sin un índice o tabla adicional

Mejora propuesta: Crear tabla adicional para consultas por episodio:

```
CREATE TABLE visualizaciones_por_episodio (
    id_episodio UUID,
    id_usuario UUID,
    timestamp TIMESTAMP,
    dispositivo TEXT,
    PRIMARY KEY ((id_episodio), timestamp)
);
```

Caso 2: Lectura Consulta:

```
sql Copiar código SELECT * FROM reproducciones WHERE id_episodio = 321e4567-e89b-12d3-a456-426614174999; Análisis:
Ineficiente y no permitida tal como está.
```

id_episodio no forma parte de la clave primaria ni de un índice secundario. Cassandra no puede buscar por columnas que no están en la clave primaria sin un índice, y no escanea toda la tabla como lo haría SQL tradicional.

Mejora propuesta: Opción 1 (más controlada): Crear una tabla adicional optimizada para consultas por episodio:

```
CREATE TABLE visualizaciones_por_episodio ( id_episodio UUID, id_usuario UUID, timestamp TIMESTAMP, dispositivo TEXT,
PRIMARY KEY ((id_episodio), timestamp) );
```

Caso 3: Actualización

Actualizar dispositivo desde el que el usuario u001 vio el episodio e321 el 5 de julio.

```
UPDATE reproducciones
SET dispositivo = 'Tablet'
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000;
```

Análisis:

Incorrecta. Falta timestamp que es parte de la clave primaria.

Mejora propuesta:

```
UPDATE reproducciones
SET dispositivo = 'Tablet'
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000
AND timestamp = '2025-07-05 19:30:00';
```

Caso 3: Actualización Consulta:

sql Copiar código UPDATE reproducciones SET dispositivo = 'Tablet' WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000; Análisis: Incorrecta: falta el valor de timestamp, que es parte de la clave primaria. Cassandra requiere especificar toda la clave primaria para actualizar una fila.

Mejora propuesta: sql Copiar código UPDATE reproducciones SET dispositivo = 'Tablet' WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000 AND timestamp = '2025-07-05 19:30:00'; Esto garantiza que la actualización se aplique a un único registro.

Caso 4: Eliminación

Eliminar registro de visualización de u001 en la fecha indicada.

```
DELETE FROM reproducciones
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000;
```

Análisis:

Peligroso e ineficiente. Borra toda la partición del usuario, puede generar tombstones y afectar rendimiento.

Mejora propuesta:

```
DELETE FROM reproducciones
WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000
AND timestamp = '2025-07-05 19:30:00';
```

Caso 4: Eliminación Consulta:

sql Copiar código DELETE FROM reproducciones WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000;

Análisis: Peligrosa e ineficiente: esta consulta elimina toda la partición del usuario.

Si el usuario tiene muchas visualizaciones, esta operación puede ser costosa y generar tombstones (marcadores de eliminación) que afectan el rendimiento.

Mejora propuesta: Si el objetivo era borrar una visualización específica:

DELETE FROM reproducciones WHERE id_usuario = 123e4567-e89b-12d3-a456-426614174000 AND timestamp = '2025-07-05 19:30:00';

Esto borra solo un registro, sin afectar el resto de los datos del usuario.

Caso	Evaluación	Problema principal	Mejora propuesta
1 - Inserción	<input checked="" type="checkbox"/> Correcta	Ninguno	No requiere cambios
2 - Lectura	<input type="checkbox"/> Ineficiente	id_episode no es parte de la clave primaria	Rediseñar tabla o usar índice (con precaución)
3 - Actualización	<input type="checkbox"/> Incorrecta	Falta timestamp en WHERE	Agregar timestamp para completar la clave primaria
4 - Eliminación	<input type="checkbox"/> Riesgosa	Elimina toda la partición del usuario	Especificar timestamp para borrar solo un registro

En Cassandra, el rendimiento y la validez de las operaciones CRUD dependen totalmente de cómo se define la clave primaria. El modelado debe estar orientado a las consultas esperadas, y no se recomienda aplicar directamente patrones tradicionales de SQL relacional.

LECCIÓN 5

Ejercicio 1

Actividad: Modelado de colecciones para una app de e-commerce

Contexto:

MongoDB permite almacenar información flexible y semiestructurada, ideal para representar productos, usuarios y órdenes en una tienda online. Esta actividad busca ejercitarse en el diseño de documentos optimizados.

Consigna:

Diseña una estructura de colección para una tienda online que permita guardar información de productos, incluyendo variantes (talle, color), reviews y disponibilidad por sucursal.

Tiempo estimado: 30 minutos

Paso a paso:

1. Definir campos clave del producto (nombre, descripción, precio, etc.).
2. Usar objetos anidados y arrays cuando sea útil (por ejemplo: variantes, reviews).
3. Justificar las decisiones del modelo.
4. Escribir un ejemplo en JSON.
5. Compartir y discutir con otro grupo.

Estructura del documento

Cada producto incluirá:

- **nombre**: Nombre del producto.
 - **descripcion**: Detalles del producto.
 - **precio**: Precio base.
 - **categorias**: Array de etiquetas para clasificación.
 - **variantes**: Array de objetos con color y talle.
 - **disponibilidad**: Array de objetos con sucursal y stock.
 - **reviews**: Array de objetos con usuario, comentario, calificación y fecha.
-

Solución propuesta

Justificación del diseño

- **Arrays anidados** permiten agrupar información relacionada sin necesidad de múltiples consultas (como variantes o reviews).
 - **Datos embebidos** para reviews y variantes, porque pertenecen lógicamente al producto y no necesitan buscarse por separado.
 - **Sucursales con stock embebido** porque la consulta típica será "¿cuánto stock hay de este producto en cada sucursal?".
-

Ejemplo en JSON

```
{  
  "nombre": "Zapatillas Urbanas",  
  "descripcion": "Zapatillas cómodas de uso diario",  
  "precio": 34990,  
  "categorias": ["calzado", "urbano", "oferta"],  
  "variantes": [  
    { "color": "negro", "talle": 38 },  
    { "color": "negro", "talle": 40 },  
    { "color": "blanco", "talle": 39 }  
  ],  
  "disponibilidad": [  
    { "sucursal": "Santiago Centro", "stock": 12 },  
    { "sucursal": "Providencia", "stock": 5 }  
  ],  
  "reviews": [  
    {  
      "usuario": "carla123",  
      "comentario": "Muy cómodas y livianas.",  
      "calificacion": 5,  
      "fecha": "2024-11-02"  
    },  
    {  
      "usuario": "jorge_m",  
      "comentario": "El talle corre un poco chico.",  
      "calificacion": 4,  
      "fecha": "2024-11-05"  
    }  
  ]  
}
```

Consultas útiles para la tienda online (código MongoDB)

1. Buscar productos por color específico

Ejemplo: 1. Busca productos que tengan al menos una variante con color "negro".

```
db.productos.find({ "variantes.color": "negro" })
```

2. Buscar productos con stock en una sucursal específica Ejemplo: Devuelve productos disponibles en la sucursal "Providencia" con stock mayor a 0.

```
db.productos.find({ "disponibilidad.sucursal": "Providencia", "disponibilidad.stock": { $gt: 0 } })
```

3. Buscar productos con reseñas de calificación baja Ejemplo:

Muestra productos que tengan al menos una reseña con calificación 2 o menor.

```
db.productos.find({ "reviews.calificacion": { $lte: 2 } })
```

4. Mostrar solo productos con cierta categoría

Ejemplo: Filtra productos que contengan "oferta" en el array de categorías.

```
db.productos.find({ categorias: "oferta" })
```

5. Consultar productos con variantes de cierto tamaño

Ejemplo: Devuelve productos que tengan una variante en tamaño 40.

Copiar código

```
db.productos.find({ "variantes.talle": 40 })
```

✓ Ejercicio 2

Actividad: Consultas avanzadas con operadores MongoDB

Contexto:

MongoDB permite realizar consultas muy específicas con operadores como `$gt`, `$in`, `$regex`, `$and` y `$or`. Esta actividad entrena el uso práctico de estas herramientas para resolver necesidades reales de negocio sobre una colección de usuarios.

Consigna:

A partir de una colección de usuarios con datos de edad, ciudad, intereses y fecha de registro, redactá consultas que respondan preguntas del negocio usando operadores avanzados.

Tiempo estimado: 30 minutos

Puntaje: 40 puntos

Paso a paso:

1. El docente entregará un esquema de colección ejemplo.
2. En parejas, escriban consultas para responder las siguientes preguntas:

- ¿Qué usuarios son mayores de 30 y viven en Rosario?
 - ¿Qué usuarios se registraron después de 2023-01-01 y tienen "data" como interés?
 - ¿Cuántos usuarios tienen nombre que comienza con "A" o viven en Córdoba?
3. Utilicen los operadores correctos y apliquen proyección para limitar los campos devueltos.
4. Validar las consultas entre grupos y revisar los resultados esperados.
-

Documento de ejemplo (entregado por el docente):

```
{  
  "_id": ObjectId,  
  "nombre": "Martina Soto",  
  "edad": 34,  
  "direccion": {  
    "ciudad": "Curicó",  
    "region": "Maule"  
  },  
  "intereses": [  
    { "nombre": "data", "nivel": "alto" },  
    { "nombre": "jardinería", "nivel": "medio" }  
  ],  
  "fecha_registro": ISODate("2023-04-12"),  
  "activo": true  
}
```

Solución propuesta:

Consultas avanzadas con operadores MongoDB

Consultas pedidas por el enunciado:

1. ¿Qué usuarios son mayores de 30 y viven en Rosario?

```
db.usuarios.find(  
  {  
    $and: [  
      { edad: { $gt: 30 } },  
      { "direccion.ciudad": "Rosario" }  
    ]  
  },  
  {  
    nombre: 1,  
    edad: 1,  
    "direccion.ciudad": 1,  
    _id: 0  
  }  
)
```

2. ¿Qué usuarios se registraron después del 1 de enero de 2023 y tienen "data" como interés?

```
db.usuarios.find(  
  {
```

```

$and: [
  { fecha_registro: { $gt: ISODate("2023-01-01") } },
  { "intereses.nombre": "data" }
]
},
{
  nombre: 1,
  intereses: 1,
  fecha_registro: 1,
  _id: 0
}
)

```

3. ¿Cuántos usuarios tienen nombre que comienza con "A" o viven en Córdoba?

```

db.usuarios.countDocuments({
  $or: [
    { nombre: { $regex: /^A/, $options: "i" } },
    { "direccion.ciudad": "Córdoba" }
  ]
})

```

Explicación técnica de operadores usados:

- \$gt : compara si un valor (edad, fecha) es mayor que el dado.
- \$and : combina múltiples condiciones obligatorias.
- \$or : permite cumplir una de varias condiciones.
- \$regex : busca coincidencias por patrones en cadenas (ej: nombres que comienzan con "A").
- "campo": 1 en la proyección muestra solo esos campos; "_id": 0 lo oculta.
- "intereses.nombre": "data" busca dentro del array de objetos en intereses .

Observación:

Este diseño permite realizar consultas complejas sin necesidad de joins, gracias al uso de arrays y documentos embebidos, aprovechando la flexibilidad de MongoDB.

LECCION 6

Escritura y recuperación en lote

1. Consigna

Contexto:

DynamoDB permite insertar y recuperar múltiples ítems en una sola operación mediante batch writing (`BatchWriteItem`) y batch retrieve (`BatchGetItem`). Esta funcionalidad es esencial para aplicaciones de alto volumen.

Objetivo:

Simular la carga y recuperación de múltiples ítems en una tabla llamada `Usuarios` utilizando operaciones por lote (`batch`), con el fin de comprender su funcionamiento y ventajas.

Tiempo estimado: 30 minutos

Instrucciones:

1. Escribe un comando simulado `BatchWriteItem` que incluya tres usuarios con los atributos: `UserID`, `Nombre`, `Edad`, `Ciudad`.
2. Luego, escribe un comando `BatchGetItem` para recuperar dos de esos usuarios.
3. Justifica por qué es útil usar operaciones por lote en este caso.
4. (Opcional) Calcula cuántas operaciones individuales se ahorrarían usando batch en vez de `PutItem` o `GetItem` por ítem.

Solución propuesta:

1. Simulación de escritura en lote (`BatchWriteItem`)

El comando `BatchWriteItem` permite insertar varios ítems (filas) en una sola operación. En este caso, estás insertando 3 usuarios a la vez en una tabla DynamoDB, lo que es mucho más eficiente que insertar uno por uno con `PutItem`.

Ventaja: reduce la cantidad de llamadas a la API y mejora el rendimiento general de la aplicación.

En el ejemplo se usaron campos: `UserID`, `Nombre`, `Edad`, `Ciudad`.

```
$batchWriteRequest = @{
    RequestItems = @{
        Usuarios = @(
            @{
                PutRequest = @{
                    Item = @{
                        UserID = @{ S = "U001" }; Nombre = @{ S = "Ana" }; Edad = @{ N = "29" }
                    }
                }
            },
            @{
                PutRequest = @{
                    Item = @{
                        UserID = @{ S = "U002" }; Nombre = @{ S = "Luis" }; Edad = @{ N = "34" }
                    }
                }
            },
            @{
                PutRequest = @{
                    Item = @{
                        UserID = @{ S = "U003" }; Nombre = @{ S = "Marta" }; Edad = @{ N = "25" }
                    }
                }
            }
        )
    }
}

Write-Output "Simulación de BatchWriteItem:"
$batchWriteRequest
```

2. Simulación de recuperación en lote (`BatchGetItem`)

Este comando permite recuperar múltiples ítems a la vez, siempre que conozcas su clave primaria.

En el ejercicio se usaron `UserID` como clave para obtener 2 de los 3 usuarios previamente cargados.

En lugar de hacer dos llamadas `GetItem`, se hace una sola con los dos IDs.

```
$batchGetRequest = @{
    RequestItems = @{
        Usuarios = @{
            Keys = @(
                @{
                    UserID = @{ S = "U001" } },
                @{
                    UserID = @{ S = "U003" } }
            )
        }
    }
}

Write-Output "Simulación de BatchGetItem:"
$batchGetRequest
```

3. Justificación:

¿Por qué usar batch? Usar operaciones en lote (batch-write-item y batch-get-item) permite:

- Reducir el número total de llamadas a la API, lo cual mejora la eficiencia de red.
- Disminuir la latencia total en comparación con múltiples operaciones individuales.
- Reducir costos operativos, especialmente útil en sistemas con alto volumen de usuarios o datos.
- Es ideal para cargas iniciales, migraciones o sincronización de datos.

Justificación:

Las aplicaciones de mensajería, redes sociales o juegos suelen manejar muchos datos en poco tiempo.

Operaciones batch permiten:

Reducir latencia (menos llamadas).

Aumentar eficiencia (más ítems por operación).

Minimizar costos si se paga por número de llamadas a la API.

Conclusión: Para una app que maneje muchos usuarios o datos de forma simultánea, el uso de BatchWriteItem y BatchGetItem es clave para mantener el rendimiento.

4. Cálculo de ahorro de operaciones

Operaciones individuales necesarias:

3 Llamadas PutItem

2 Llamadas GetItem

Total: 5 operaciones

Operaciones batch utilizadas:

1 BatchWriteItem

1 BatchGetItem

Total: 2 operaciones

Ahorro total: 3 operaciones

Esto representa una mejora notable en eficiencia, sobre todo en escenarios a gran escala.

Reflexión Final

A través de este ejercicio se comprendió la utilidad de las operaciones por lote (BatchWriteItem y BatchGetItem) en DynamoDB. Estas permiten realizar tareas de carga y recuperación de datos de forma más eficiente, minimizando el número de llamadas individuales a la base de datos. En situaciones reales, como el manejo de información de usuarios en una aplicación de gran escala, estas herramientas resultan esenciales para optimizar el rendimiento, reducir costos operativos y aumentar la velocidad de procesamiento.

Este ejercicio permitió aplicar conceptos claves del uso de DynamoDB desde PowerShell, simulando flujos reales de interacción con la base de datos y destacando la importancia de planificar operaciones en función de la eficiencia.

