Arquitecturas NoSQL vs SQL para Diferentes Casos de Uso - Día 4

Change Status

in-progress 40 min

Learning Objectives

1. Evaluar fortalezas y limitaciones de paradigmas SQL vs NoSQL
2. Seleccionar tecnologías basadas en requisitos específicos del análisis
3. Diseñar arquitecturas híbridas que aprovechen lo mejor de ambos mundos

Activities and Learning

Task 1: Paradigmas de Bases de Datos: SQL vs NoSQL (10 minutos)

Dos mundos filosóficos diferentes para almacenar y acceder datos, cada uno optimizado para diferentes patrones de uso.

SQL (Relational): Consistencia y Estructura

Características fundamentales:

Esquemas rígidos: Estructura predefinida con constraints
ACID transactions: Atomicidad, Consistencia, Aislamiento, Durabilidad
Joins complejos: Relaciones entre tablas normalizadas
SQL como interfaz: Lenguaje declarativo estandarizado

Fortalezas para análisis:

Integridad referencial: Datos consistentes y relacionados
Consultas complejas: Joins, subqueries, agregaciones avanzadas
Transacciones: Garantías de consistencia para operaciones críticas
Estandarización: SQL universalmente conocido

Limitaciones:

Escalabilidad horizontal: Difícil distribuir joins complejos
Esquemas rígidos: Cambios estructurales requieren migraciones
Performance en escritura: Normalización puede ser costosa
NoSQL: Flexibilidad y Escalabilidad

Características fundamentales:

Esquemas flexibles: Sin estructura predefinida estricta
Escalabilidad horizontal: Diseñado para distribución
Modelos diversos: Key-value, document, column-family, graph
Eventual consistency: Prioriza disponibilidad sobre consistencia inmediata

Fortalezas para análisis:

Escalabilidad masiva: Maneja petabytes eficientemente
Flexibilidad: Adapta fácilmente a cambios en requisitos
Performance: Optimizado para patrones específicos de consulta
Costo-efectivo: Escalado horizontal más económico

Limitaciones:

Consistencia eventual: Datos pueden estar desactualizados temporalmente
Consultas complejas: Joins no nativos, requieren aplicación
Expertise especializada: Cada sistema tiene su propio lenguaje/query

Task 2: Familias NoSQL y sus Casos de Uso (12 minutos)

Cuatro familias principales de NoSQL, cada una optimizada para diferentes patrones de datos y consulta.

Key-Value Stores: Simplicidad Extrema

Ejemplos: Redis, DynamoDB, Riak **Caso de uso**: Caché, sesiones, configuraciones

```
# Redis para caching de análisis
import redis

r = redis.Redis(host='localhost', port=6379, decode_responses=True)

# Almacenar métricas calculadas
r.set('ventas_diarias:2024-01-15', '125000')
```

```
r.set('usuarios_activos:2024-01-15', '8500')

# Recuperar para dashboards
ventas = r.get('ventas_diarias:2024-01-15')
usuarios = r.get('usuarios_activos:2024-01-15')

# Hashes para estructuras complejas
r.hset('cliente:12345', mapping={
    'nombre': 'Juan Pérez',
    'segmento': 'VIP',
    'ultima_compra': '2024-01-15',
    'total_compras': '45000'
})
```

Ventajas: Performance extrema para lookups simples **Desventajas**: No soporta queries complejas

Document Databases: Flexibilidad JSON

Ejemplos: MongoDB, CouchDB, Elasticsearch **Caso de uso**: Contenido variable, catálogos de productos, analytics

```
// MongoDB para datos de productos variables
db.productos.insertOne({
    _id: ObjectId(),
    nombre: "Laptop Gaming Pro",
    categoria: "Electrónica",
    precio: 1299.99,
    especificaciones: {
        procesador: "Intel i7",
        ram: "16GB",
        almacenamiento: "512GB SSD",
        pantalla: "15.6\" 144Hz"
    },
    reviews: [
        {usuario: "user123", rating: 5, comentario: "Excelente producto"},
        {usuario: "user456", rating: 4, comentario: "Buen rendimiento"}
    ],
    tags: ["gaming", "laptop", "intel"],
    fecha_creacion: new Date()
});

// Query flexible para analytics
db.productos.aggregate([
    {$match: {"categoria": "Electrónica", "precio": {$gte: 1000}}},
    {$group: {
        _id: "$especificaciones.procesador",
        productos: {$sum: 1},
        precio_promedio: {$avg: "$precio"}
    }}
]);
```

Ventajas: Esquemas flexibles, queries ricas en documentos **Desventajas**: Joins complejos, consistencia eventual

Column-Family Stores: Analytics de Alto Rendimiento

Ejemplos: Cassandra, HBase, Bigtable **Caso de uso**: Time-series, logs, analytics masivos

```
-- Cassandra para time-series de eventos
CREATE KEYSPACE analytics WITH REPLICATION = {
    'class': 'SimpleStrategy',
    'replication_factor': 3
};

CREATE TABLE eventos_usuario (
    user_id UUID,
    event_time TIMESTAMP,
    event_type TEXT,
    properties MAP<TEXT, TEXT>,
    PRIMARY KEY ((user_id, event_time), event_time)
) WITH CLUSTERING ORDER BY (event_time DESC);

-- Query optimizada para analytics temporales
SELECT user_id, event_time, event_type
FROM eventos_usuario
WHERE user_id = ?
  AND event_time >= ?
  AND event_time <= ?
ALLOW FILTERING;
```

Ventajas: Escalabilidad masiva, performance en writes **Desventajas**: Queries complejas limitadas, aprendizaje empinado

Graph Databases: Relaciones Complejas

Ejemplos: Neo4j, Amazon Neptune, JanusGraph **Caso de uso**: Redes sociales, recomendaciones, fraud detection

```
// Neo4j para análisis de recomendaciones
CREATE (u1:Usuario {id: 1, nombre: "Ana"})
```

```
CREATE (u2:Usuario {id: 2, nombre: "Carlos"})
CREATE (p1:Producto {id: 100, nombre: "Laptop"})
CREATE (p2:Producto {id: 200, nombre: "Mouse"})

// Relaciones de compra
CREATE (u1)-[:COMPRO {fecha: "2024-01-15", precio: 1299.99}]->(p1)
CREATE (u2)-[:COMPRO {fecha: "2024-01-16", precio: 1299.99}]->(p1)
CREATE (u1)-[:COMPRO {fecha: "2024-01-15", precio: 29.99}]->(p2)

// Relaciones de similitud
CREATE (u1)-[:SIMILAR {score: 0.85}]->(u2)

// Query para recomendaciones
MATCH (usuario:Usuario {id: 1})-[:COMPRO]->(producto:Producto)<-[:COMPRO]-(otro:Usuario)-[:COMPRO]->(recomendacion:Producto)
WHERE usuario <> otro
RETURN recomendacion.nombre, COUNT(*) as frecuencia
ORDER BY frecuencia DESC
LIMIT 5;
```

Ventajas: Queries de relaciones complejas, performance en graphs **Desventajas**: No optimizado para agregaciones masivas

Task 3: Guía de Selección y Arquitecturas Híbridas (8 minutos)

Elegir la tecnología correcta requiere entender profundamente los requisitos del análisis.

Decision Framework

Factores críticos de decisión:

Patrón de datos:

Estructurados fijos → SQL (PostgreSQL, MySQL)
Documentos variables → Document (MongoDB)
Time-series → Column-family (Cassandra, InfluxDB)
Relaciones complejas → Graph (Neo4j)

Requisitos de consistencia:

Transacciones críticas → SQL con ACID
Eventual consistency aceptable → NoSQL distribuido
Lectura masiva → NoSQL optimizado para reads

Escalabilidad requerida:

< 1TB, consultas complejas → SQL tradicional
1TB-100TB, analytics → NoSQL columnar
> 100TB, variable → Arquitectura híbrida

Equipo y expertise:

SQL tradicional → Equipos con background relacional
NoSQL moderno → Equipos dispuestos a aprender paradigmas nuevos
Híbrido → Equipos con diversidad tecnológica
Arquitecturas Híbridas: Lo Mejor de Ambos Mundos

Patrón Polystore: Diferentes datos en diferentes sistemas.

```
# Arquitectura polystore para e-commerce
arquitectura_polystore = {
    # SQL para datos transaccionales críticos
    'postgresql': {
        'usos': ['Pedidos', 'Inventario', 'Clientes maestros'],
        'ventajas': ['ACID', 'Integridad referencial', 'Consultas complejas']
    },

    # MongoDB para catálogos flexibles
    'mongodb': {
        'usos': ['Productos con atributos variables', 'Reviews', 'Contenido'],
        'ventajas': ['Esquemas flexibles', 'Consultas en documentos']
    },

    # Cassandra para eventos y analytics
    'cassandra': {
        'usos': ['Logs de usuario', 'Eventos de click', 'Métricas temporales'],
        'ventajas': ['Escalabilidad horizontal', 'Performance en writes']
    },

    # Redis para caching y sesiones
    'redis': {
        'usos': ['Sesiones de usuario', 'Caché de consultas', 'Leaderboards'],
        'ventajas': ['Performance extrema', 'Estructuras de datos ricas']
    }
}
```

Patrón CQRS: Command Query Responsibility Segregation.

```python
# CQRS: Separar writes (comando) de reads (query)
class CQRSArchitecture:
    def __init__(self):
        self.command_side = PostgreSQL()  # Writes normalizados
        self.query_side = MongoDB()       # Reads desnormalizados

    def place_order(self, order):
        """Write side: Validación estricta, normalización"""
        validated_order = self.validate_order(order)
        self.command_side.save_order(validated_order)

        # Publicar evento para actualizar read models
        self.publish_event('OrderPlaced', validated_order)

    def get_customer_orders(self, customer_id):
        """Read side: Optimizado para consultas específicas"""
        return self.query_side.get_customer_order_summary(customer_id)
```

Migración inteligente: Estrategias para transitar entre paradigmas.

```python
# Estrategia de migración gradual
class MigrationStrategy:
    def phase_1_dual_writes(self):
        """Fase 1: Escribir en ambos sistemas"""
        # Nuevo sistema recibe todas las writes
        # Sistema legacy mantiene consistencia
        # Validar consistencia entre sistemas

    def phase_2_backfill(self):
        """Fase 2: Migrar datos históricos"""
        # Batch migration de datos históricos
        # Validación de integridad
        # Switch de reads gradualmente

    def phase_3_cutover(self):
        """Fase 3: Transición completa"""
        # Routing de traffic al nuevo sistema
        # Monitoreo intensivo
        # Plan de rollback si es necesario
```