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

Practical exercise to apply the concepts learned.

Ejercicio: Diseño de arquitectura híbrida para plataforma de streaming de video

Análisis de requisitos y patrones de datos:

```
# Requisitos para plataforma de streaming
requisitos_streaming = {
    'datos_transaccionales': {
        'suscripciones': 'ACID crítico, joins complejos',
        'pagos': 'Consistencia fuerte requerida',
        'usuarios': 'Datos maestros normalizados'
    },
    'datos_analiticos': {
        'eventos_reproduccion': 'Volumen masivo, time-series',
        'recomendaciones': 'Relaciones complejas entre usuarios/contenido',
        'analytics_contenido': 'Métricas agregadas variables'
    },
    'requisitos_performance': {
        'latencia_vista': '< 100ms para recomendaciones',
        'throughput_eventos': '1M eventos/segundo',
        'almacenamiento': '100PB datos históricos'
    }
}

print("REQUISITOS PLATAFORMA DE STREAMING")
print("=" * 35)

for categoria, detalles in requisitos_streaming.items():
    print(f"\n{categoria.upper().replace('_', ' ')}:")
    if isinstance(detalles, dict):
        for subcat, desc in detalles.items():
            print(f"  {subcat.title()}: {desc}")
    else:
        print(f"  {detalles}")
```

Selección de tecnologías por caso de uso:

```
# Arquitectura híbrida seleccionada
arquitectura_hibrida = {
    'postgresql': {
        'rol': 'Base de datos transaccional principal',
        'casos_uso': ['Suscripciones', 'Pagos', 'Perfiles de usuario'],
        'justificacion': 'ACID para finanzas, joins complejos para billing',
        'escalabilidad': 'Vertical (hasta ~10TB)',
        'limitaciones': 'Escalabilidad horizontal limitada'
    },

    'cassandra': {
        'rol': 'Base de datos de eventos y analytics',
        'casos_uso': ['Eventos de reproducción', 'Métricas de usuario', 'Logs'],
        'justificacion': 'Escalabilidad horizontal masiva, writes de alto throughput',
        'escalabilidad': 'Horizontal ilimitada',
        'limitaciones': 'Queries complejas limitadas'
    },

    'neo4j': {
        'rol': 'Motor de recomendaciones y relaciones',
        'casos_uso': ['Sistema de recomendaciones', 'Análisis de afinidad', 'Detección de fraude'],
        'justificacion': 'Queries de relaciones complejas, algoritmos de grafos',
        'escalabilidad': 'Hasta ~100B nodos/relaciones',
        'limitaciones': 'No optimizado para agregaciones masivas'
    },

    'redis': {
        'rol': 'Caché y sesiones de alto performance',
```

```
            'casos_uso': ['Sesiones de usuario', 'Caché de recomendaciones', 'Leaderboards'],
            'justificacion': 'Latencia < 1ms, estructuras de datos ricas',
            'escalabilidad': 'Cluster horizontal',
            'limitaciones': 'Datos volátiles (reinicio borra datos)'
        },

        'elasticsearch': {
            'rol': 'Búsqueda y analytics de contenido',
            'casos_uso': ['Búsqueda de contenido', 'Analytics de catálogo', 'Logs estructurados'],
            'justificacion': 'Búsqueda full-text, agregaciones complejas, APIs REST',
            'escalabilidad': 'Horizontal con sharding',
            'limitaciones': 'No transaccional, eventual consistency'
        }
    }

    print("
```

ARQUITECTURA HÍBRIDA SELECCIONADA") print("=" * 40)

for tecnologia, detalles in arquitectura_hibrida.items(): print(f"\n{tecnologia.upper()}:") print(f" Rol: {detalles['rol']}") print(f" Casos de uso: {', '.join(detalles['casos_uso'])}") print(f" Escalabilidad: {detalles['escalabilidad']}")

3. **Diseño de esquemas y patrones de consulta**:

```sql
-- PostgreSQL: Datos transaccionales críticos
CREATE TABLE suscripciones (
    id SERIAL PRIMARY KEY,
    usuario_id INTEGER REFERENCES usuarios(id),
    plan_id INTEGER REFERENCES planes(id),
    fecha_inicio DATE,
    fecha_fin DATE,
    estado VARCHAR(20),
    precio_mensual DECIMAL(8,2),
    metodo_pago VARCHAR(50)
);

-- Cassandra: Eventos de reproducción (time-series)
CREATE KEYSPACE streaming WITH REPLICATION = {
    'class': 'NetworkTopologyStrategy',
    'datacenter1': 3
};

CREATE TABLE eventos_reproduccion (
    usuario_id UUID,
    contenido_id UUID,
    timestamp TIMESTAMP,
    duracion_reproducida INT,
    posicion_actual INT,
    dispositivo VARCHAR,
    calidad VARCHAR,
    PRIMARY KEY ((usuario_id, contenido_id), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- Neo4j: Grafo de relaciones usuario-contenido
// Nodos principales
CREATE (u:Usuario {id: 1, nombre: "Ana"})
CREATE (c:Contenido {id: 100, titulo: "Serie Drama", genero: "Drama"})

// Relaciones
CREATE (u)-[:VIO {rating: 5, tiempo_completo: true}]->(c)
CREATE (u)-[:BUSCO_GENERO]->(:Genero {nombre: "Drama"})

// Query de recomendaciones
MATCH (u:Usuario {id: 1})-[:VIO]->(c1:Contenido)-[:DEL_GENERO]->(g:Genero)<-[:DEL_GENERO]-(c2:Contenido)
WHERE NOT (u)-[:VIO]->(c2)
RETURN c2.titulo, COUNT(*) as afinidad
ORDER BY afinidad DESC
LIMIT 10;

-- Redis: Caché de recomendaciones
// Hashes para recomendaciones por usuario
HSET recomendaciones:usuario:1 serie:100 0.95 serie:200 0.87 serie:150 0.82

// Sorted sets para trending
ZADD trending:series 154 serie:100
ZADD trending:series 128 serie:200

-- Elasticsearch: Búsqueda de contenido
PUT /contenido/_doc/100
{
  "titulo": "Serie Drama Completa",
  "genero": ["Drama", "Suspenso"],
  "actores": ["Actor A", "Actor B"],
  "descripcion": "Serie de drama intenso...",
  "rating_promedio": 4.5,
  "temporadas": 3
}

// Query de búsqueda
GET /contenido/_search
{
  "query": {
    "bool": {
```

```
      "must": [
        {"multi_match": {"query": "drama", "fields": ["titulo", "descripcion"]}},
        {"terms": {"genero": ["Drama", "Suspenso"]}}
      ],
      "filter": {"range": {"rating_promedio": {"gte": 4.0}}}
    }
  }
}
```

Implementación de patrón CQRS:

```python
# Implementación CQRS para plataforma de streaming
class StreamingCQRS:
    def __init__(self):
        self.command_db = PostgreSQL()  # Writes normalizados
        self.query_db = Cassandra()     # Reads optimizados
        self.cache = Redis()
        self.search = Elasticsearch()

    # Command side: Validación estricta, consistencia
    def create_subscription(self, user_id, plan_id, payment_info):
        """Crear suscripción - lado comando"""
        # Validar usuario existe
        if not self.command_db.user_exists(user_id):
            raise ValueError("Usuario no existe")

        # Validar plan disponible
        if not self.command_db.plan_available(plan_id):
            raise ValueError("Plan no disponible")

        # Procesar pago (simulado)
        payment_result = self.process_payment(payment_info)

        if payment_result['success']:
            # Crear suscripción en BD transaccional
            subscription = self.command_db.create_subscription({
                'user_id': user_id,
                'plan_id': plan_id,
                'payment_id': payment_result['id']
            })

            # Publicar evento para actualizar read models
            self.publish_event('SubscriptionCreated', subscription)

            return subscription
        else:
            raise ValueError("Pago fallido")

    # Query side: Optimizado para lecturas rápidas
    def get_user_recommendations(self, user_id):
        """Obtener recomendaciones - lado query"""
        # Primero intentar caché
        cache_key = f"recommendations:{user_id}"
        cached = self.cache.get(cache_key)

        if cached:
            return json.loads(cached)

        # Si no está en caché, calcular desde query model
        recommendations = self.query_db.get_user_recommendations(user_id)

        # Almacenar en caché para futuras consultas
        self.cache.setex(cache_key, 3600, json.dumps(recommendations))  # 1 hora

        return recommendations

    # Event handling: Mantener consistencia eventual
    def handle_subscription_created(self, event):
        """Actualizar read models cuando se crea suscripción"""
        # Actualizar perfil de usuario en query model
        self.query_db.update_user_profile(event['user_id'], {
            'subscription_active': True,
            'plan_id': event['plan_id'],
            'subscription_date': event['created_at']
        })

        # Invalidar caché relacionado
        self.cache.delete(f"user_profile:{event['user_id']}")
        self.cache.delete(f"recommendations:{event['user_id']}")

# Uso del sistema
cqrs = StreamingCQRS()

# Crear suscripción (command)
subscription = cqrs.create_subscription(user_id=123, plan_id=1, payment_info={...})

# Obtener recomendaciones (query)
recommendations = cqrs.get_user_recommendations(user_id=123)
```

Verificación: ¿Por qué elegirías esta arquitectura híbrida sobre un sistema SQL puro o NoSQL puro? ¿Qué desafíos introduciría esta complejidad adicional y cómo los mitigarías?

Requerimientos:

Conocimientos básicos de SQL

Familiaridad con conceptos de bases de datos
Jupyter para ejemplos conceptuales
Opcional: Conexiones a servicios reales para pruebas