Manejo de Grandes Volúmenes de Datos (Big Data Basics) - Día 3

Change Status

pending 40 min

Learning Objectives

1. Comprender los desafíos únicos del procesamiento de datos a gran escala
2. Diseñar arquitecturas que escalen horizontalmente para crecimiento futuro
3. Evaluar trade-offs entre consistencia, disponibilidad y particionamiento

Practical exercise to apply the concepts learned.

Ejercicio: Diseño de arquitectura Big Data para análisis de e-commerce

Análisis de requisitos y patrones de datos:

```
# Análisis de volúmenes y patrones para e-commerce
import pandas as pd
import numpy as np

# Estimación de volúmenes para plataforma e-commerce
estimaciones_mensuales = {
    'eventos_usuario': 50000000,    # 50M eventos (clicks, views, etc.)
    'ordenes': 1000000,             # 1M órdenes
    'productos': 100000,            # 100K productos
    'clientes_activos': 5000000,    # 5M clientes activos
    'reviews': 500000,              # 500K reviews
    'logs_sistema': 100000000       # 100M logs diarios
}

print("ESTIMACIONES DE VOLUMEN - E-COMMERCE MENSUAL")
print("=" * 50)
for componente, volumen in estimaciones_mensuales.items():
    print("25")

# Patrones de consulta identificados
patrones_consulta = {
    'tiempo_real': [
        '¿Cuántos usuarios activos ahora?',
        '¿Cuál es la conversión actual?',
        '¿Hay anomalías en ventas?'
    ],
    'batch_diario': [
        'Reportes de ventas por categoría',
        'Análisis de comportamiento de cliente',
        'Optimización de inventario'
    ],
    'batch_semanal': [
        'Tendencias de productos',
        'Segmentación de clientes',
        'Análisis de campañas de marketing'
    ]
}

print("
```

PATRONES DE CONSULTA IDENTIFICADOS") print("=" * 40) for frecuencia, consultas in patrones_consulta.items(): print(f"\n{frecuencia.upper()}:") for consulta in consultas: print(f" • {consulta}")

2. **Diseño de arquitectura híbrida Lambda**:

```python
# Arquitectura Lambda para e-commerce
arquitectura_lambda = {
    'capa_streaming': {
        'tecnologias': ['Apache Kafka', 'Apache Flink', 'Redis'],
        'casos_uso': [
            'Monitoreo en tiempo real de ventas',
            'Detección de fraudes',
            'Personalización de recomendaciones',
            'Alertas de inventario bajo'
        ],
        'latencia': 'milisegundos-segundos',
        'datos': 'eventos individuales'
    },
    'capa_batch': {
        'tecnologias': ['Apache Spark', 'Hadoop MapReduce', 'Apache Airflow'],
        'casos_uso': [
            'Reportes de performance mensual',
            'Modelos de machine learning',
            'Análisis de cohortes de clientes',
            'Optimización de precios'
        ],
        'latencia': 'horas-días',
        'datos': 'datasets completos'
    },
    'capa_serving': {
        'tecnologias': ['Apache Druid', 'ClickHouse', 'Elasticsearch'],
        'funciones': [
            'Unificar resultados batch + streaming',
            'Servir consultas analíticas rápidas',
            'Dashboards en tiempo real',
            'APIs para aplicaciones'
        ]
    }
}

print("
ARQUITECTURA LAMBDA PROPUESTA")
print("=" * 35)

for capa, detalles in arquitectura_lambda.items():
    print(f"\n{capa.upper().replace('_', ' ')}:")
    print(f"  Tecnologías: {', '.join(detalles['tecnologias'])}")
    if 'latencia' in detalles:
        print(f"  Latencia: {detalles['latencia']}")
    if 'casos_uso' in detalles:
        print("  Casos de uso:")
        for caso in detalles['casos_uso']:
            print(f"    • {caso}")
```

Estrategias de particionamiento y escalabilidad:

```
-- Estrategias de particionamiento para diferentes componentes

-- 1. Eventos de usuario (streaming + histórico)
-- Kafka topics particionados por tipo de evento
CREATE TABLE eventos_usuario (
    timestamp TIMESTAMP,
    user_id BIGINT,
    event_type VARCHAR(50),
    session_id VARCHAR(100),
    properties JSONB,
    -- Particionamiento por tiempo + hash para distribución
    PARTITION BY RANGE (timestamp) SUBPARTITION BY HASH (user_id)
);

-- 2. Órdenes de compra (transaccional + analítico)
CREATE TABLE ordenes (
    order_id BIGINT PRIMARY KEY,
    user_id BIGINT,
    order_date TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(20),
    -- Particionamiento mensual para optimización temporal
    PARTITION BY RANGE (EXTRACT(YEAR_MONTH FROM order_date))
);

-- 3. Datos de productos (relacional + búsqueda)
-- Elasticsearch para búsqueda, PostgreSQL para datos maestros
CREATE TABLE productos (
    product_id BIGINT PRIMARY KEY,
    category_id INTEGER,
    name VARCHAR(200),
    price DECIMAL(10,2),
    stock_quantity INTEGER,
    -- Índices para diferentes patrones de consulta
    INDEX idx_category_price (category_id, price),
    INDEX idx_name_fts (name) USING GIN,  -- Full-text search
    INDEX idx_stock (stock_quantity) WHERE stock_quantity > 0
);

-- 4. Métricas agregadas (data warehouse columnar)
-- ClickHouse para analytics de alto rendimiento
CREATE TABLE metricas_diarias (
    fecha DATE,
    categoria VARCHAR(50),
    region VARCHAR(50),
    ventas_total DECIMAL(10,2),
    ordenes_total INTEGER,
    clientes_unicos INTEGER,
    conversion_rate DECIMAL(5,4)
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(fecha)
ORDER BY (fecha, categoria, region);
```

Implementación de pipeline de procesamiento:

```
# Pipeline de procesamiento para arquitectura Lambda
def lambda_pipeline_arquitecture():
    """
    Arquitectura Lambda simplificada para e-commerce
    """

    # CAPA DE STREAMING (velocidad)
    def capa_streaming():
        """Procesamiento en tiempo real"""
        eventos_stream = kafka_consumer.consume('user_events')

        # Procesamiento con Flink
        eventos_procesados = eventos_stream \
            .filter(lambda x: x['event_type'] == 'purchase') \
            .key_by(lambda x: x['user_id']) \
            .window(TumblingEventTimeWindows.of(Time.minutes(5))) \
            .aggregate(AggregationFunction())

        # Resultados a Redis para consultas rápidas
        eventos_procesados.addSink(redis_sink)

        # También a storage duradero para batch layer
        eventos_procesados.addSink(s3_sink)

    # CAPA DE BATCH (precisión)
    def capa_batch():
        """Procesamiento completo y preciso"""
        # Leer todos los datos históricos
        datos_completos = spark.read.parquet('s3://data-lake/events/')

        # Procesamiento completo con Spark
        metricas_batch = datos_completos \
            .groupBy('fecha', 'categoria') \
            .agg(
                sum('revenue').alias('ventas_total'),
                countDistinct('user_id').alias('clientes_unicos'),
                (sum('purchases') / countDistinct('user_id')).alias('conversion_rate')
            )

        # Guardar resultados batch
        metricas_batch.write.mode('overwrite').parquet('s3://data-lake/batch-metrics/')

    # CAPA DE SERVING (unificación)
    def capa_serving():
        """Unificar y servir resultados"""
        # Combinar resultados streaming + batch
        resultados_streaming = redis_cluster.get_recent_metrics()
        resultados_batch = spark.read.parquet('s3://data-lake/batch-metrics/')

        # Unificar en ClickHouse para consultas analíticas
        resultados_combinados = merge_results(resultados_streaming, resultados_batch)
        clickhouse_client.insert('metricas_unificadas', resultados_combinados)

    return {
        'streaming': capa_streaming,
        'batch': capa_batch,
        'serving': capa_serving
    }

# Demostración de escalabilidad
escalabilidad = {
    'volumen_actual': '10TB datos/mes',
    'proyeccion_2_años': '100TB datos/mes',
    'estrategias_escalabilidad': [
        'Auto-scaling de clusters Spark/Flink',
        'Particionamiento horizontal adicional',
```

```
            'Compresión columnar avanzada',
            'Cache distribuido (Redis Cluster)',
            'CDN para datos estáticos'
        ]
    }

    print("
```

ESTRATEGIA DE ESCALABILIDAD") print("=" * 30) print(f"Volumen actual: {escalabilidad['volumen_actual']}") print(f"Proyección 2 años: {escalabilidad['proyeccion_2_años']}") print("Estrategias:") for estrategia in escalabilidad['estrategias_escalabilidad']: print(f" • {estrategia}")

```
**Verificación**: Explica cómo la arquitectura Lambda resuelve el trade-off entre velocidad (streaming) y precisión (batch), y describe escenarios donde elegirías Kappa sobre Lambda para simplificar la arquitectura.
```

Requerimientos:

Conceptos básicos de sistemas distribuidos
Familiaridad con Apache Spark o Hadoop
Conocimientos de SQL y bases de datos
Jupyter para experimentación conceptual