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

Activities and Learning

Task 1: Los Desafíos del Big Data: Volumen, Velocidad, Variedad (10 minutos)

Big Data representa un cambio paradigmático en cómo pensamos sobre datos, procesamiento y arquitectura de sistemas.

Las 5 V del Big Data

Volumen: Cantidad masiva de datos que excede capacidades tradicionales.

Escala: De gigabytes a petabytes
Crecimiento: Datos generados por minuto: 2.5 quintillones de bytes
Almacenamiento: Costos, durabilidad, acceso eficiente

Velocidad: Rapidez de generación e ingesta de datos.

Streaming: Datos en tiempo real (sensores, logs, transacciones)
Batch processing: Procesamiento por lotes periódicos
Latencia: Desde milisegundos hasta horas

Variedad: Diversidad de tipos y formatos de datos.

Estructurados: Bases de datos relacionales
Semi-estructurados: JSON, XML, logs
No estructurados: Texto, imágenes, video, audio

Veracidad: Calidad y confiabilidad de los datos.

Calidad: Completitud, exactitud, consistencia
Limpieza: Identificación y corrección de errores
Governance: Políticas de calidad y seguridad

Valor: Capacidad de extraer insights útiles de los datos.

Arquitecturas Distribuidas: Sharding y Replication

Sharding: Dividir datos horizontalmente entre múltiples servidores.

Estrategias de sharding:

Hash-based: Distribución uniforme usando funciones hash
Range-based: Divisiones por rangos de valores
Directory-based: Tabla de mapeo centralizada

```
# Ejemplo conceptual de sharding
def shard_key(user_id, num_shards):
    """Distribuir usuarios uniformemente entre shards"""
    return hash(user_id) % num_shards

# Ventajas: Escalabilidad horizontal, carga balanceada
# Desventajas: Joins complejos entre shards, rebalancing costoso
```

Replication: Copiar datos entre múltiples nodos para disponibilidad y performance.

Tipos de replicación:

Master-Slave: Escrituras solo en master, lecturas desde slaves
Multi-Master: Escrituras en múltiples nodos (conflictos posibles)
Peer-to-Peer: Todos los nodos iguales
Modelos de Consistencia: CAP Theorem

Teorema CAP: Solo se pueden garantizar 2 de 3 propiedades en sistemas distribuidos.

Consistency (Consistencia): Todos los nodos ven los mismos datos simultáneamente. **Availability (Disponibilidad)**: Sistema responde a todas las consultas válidas. **Partition Tolerance (Tolerancia a particiones)**: Sistema funciona despite fallos de red.

Implicaciones prácticas:

CP Systems: Consistencia fuerte, disponibilidad reducida (ej: bancos)
AP Systems: Alta disponibilidad, consistencia eventual (ej: redes sociales)
CA Systems: Consistencia y disponibilidad, sin tolerancia a particiones (sistemas únicos)

Task 2: Procesamiento por Lotes vs Streaming (10 minutos)

Dos paradigmas fundamentales para procesar datos a escala, cada uno optimizado para diferentes casos de uso.

Batch Processing: Procesamiento por Lotes

Características:

Procesamiento periódico: Horas, días, semanas
Datasets completos: Todos los datos históricos
Análisis complejo: Machine learning, reportes complejos
Latencia alta: Resultados no inmediatos

Tecnologías clave:

Hadoop MapReduce: Framework distribuido para batch processing
Apache Spark: Motor unificado para procesamiento distribuido
Airflow: Orquestación de pipelines de datos

Ejemplo de pipeline batch:

```
# Pseudocódigo de pipeline batch diario
def daily_batch_pipeline():
    # 1. Ingestión: Recopilar datos del día anterior
    raw_data = collect_yesterday_data()

    # 2. Validación: Verificar calidad y completitud
    validated_data = validate_data_quality(raw_data)

    # 3. Transformación: Limpiar y enriquecer
    transformed_data = transform_business_logic(validated_data)

    # 4. Agregación: Calcular métricas diarias
    daily_metrics = calculate_daily_aggregates(transformed_data)

    # 5. Almacenamiento: Guardar en data warehouse
    save_to_data_warehouse(daily_metrics)

    # 6. Reportes: Generar dashboards y alertas
    generate_reports(daily_metrics)
```

Stream Processing: Procesamiento en Tiempo Real

Características:

Procesamiento continuo: Milisegundos a segundos
Eventos individuales: Procesamiento uno por uno
Análisis simple: Agregaciones básicas, alertas, recomendaciones
Latencia baja: Resultados inmediatos

Tecnologías clave:

Apache Kafka: Platform para streaming de eventos
Apache Flink: Procesamiento de streams complejo
Apache Storm: Computación distribuida en tiempo real

Ejemplo de procesamiento streaming:

```
# Pseudocódigo de procesamiento streaming
def streaming_pipeline():
    # 1. Consumir eventos en tiempo real
    stream = kafka_consumer.subscribe('user_events')

    # 2. Procesamiento por ventana temporal
    windowed_stream = stream.window(TumblingEventTimeWindows.of(Time.minutes(5)))

    # 3. Agregaciones en tiempo real
    real_time_metrics = windowed_stream.groupBy('user_id').aggregate(
        count('clicks'),
        sum('revenue'),
        max('session_duration')
    )

    # 4. Detección de anomalías
    anomalies = real_time_metrics.filter(lambda x: detect_anomaly(x))

    # 5. Acciones automáticas
    anomalies.foreach(lambda anomaly: send_alert(anomaly))
```

```
    # 6. Almacenamiento para análisis posterior
    real_time_metrics.addSink(elasticsearch_sink)
```

Arquitecturas Lambda y Kappa

Arquitectura Lambda: Procesamiento híbrido batch + streaming.

Componentes:

Capa batch: Procesamiento completo, alta latencia, resultados precisos
Capa streaming: Procesamiento rápido, baja latencia, resultados aproximados
Capa serving: Combinar resultados de ambas capas

Arquitectura Kappa: Solo streaming, simplificación radical.

Ventajas Kappa:

Simplicidad: Una sola pipeline para todos los casos
Mantenimiento: Menos componentes, menos complejidad
Reprocesamiento: Fácil recalcular con datos históricos

Task 3: Estrategias de Particionamiento para Datasets Masivos (10 minutos)

El particionamiento inteligente es crucial para mantener performance en sistemas de Big Data.

Particionamiento por Tiempo (Time-based Partitioning)

Estrategia más común: Particionar por tiempo para datos temporales.

```
-- Ejemplo en sistema distribuido (ClickHouse)
CREATE TABLE events (
    timestamp DateTime,
    user_id UInt64,
    event_type String,
    properties String
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)  -- Partición mensual
ORDER BY (timestamp, user_id);

-- Ventajas:
-- - Eliminación eficiente de datos antiguos
-- - Consultas temporales usan solo particiones relevantes
-- - Compresión eficiente por temporalidad
```

Particionamiento por Clave (Key-based Partitioning)

Hash partitioning: Distribución uniforme para escalabilidad.

```
# Ejemplo en Apache Cassandra
def get_partition_key(user_id, num_partitions):
    """Distribuir usuarios uniformemente"""
    return hash(user_id) % num_partitions

# Ventajas:
-- - Carga balanceada entre nodos
-- - Escalabilidad horizontal sencilla
-- - Consultas por usuario van a una partición
```

Particionamiento Jerárquico

Estrategia híbrida: Múltiples niveles de particionamiento.

```
-- Particionamiento jerárquico en Hive/Spark
CREATE TABLE sales (
    sale_date DATE,
    region STRING,
    product_id INT,
    amount DECIMAL(10,2)
)
PARTITIONED BY (
    year INT,      -- Primer nivel: año
    month INT,     -- Segundo nivel: mes
    region STRING  -- Tercer nivel: región
);

-- Estructura resultante:
/*
sales/
├── year=2024/
│   ├── month=01/
│   │   ├── region=europe/
│   │   └── region=america/
│   └── month=02/
│       ├── region=europe/
│       └── region=america/
*/
```

Estrategias de Compresión y Optimización

Compresión columnar: Eficiente para consultas analíticas.

Formatos optimizados:

Parquet: Compresión columnar excelente para analytics
ORC: Optimizado para Hive, buena compresión
Delta Lake: Formato transaccional con optimizaciones

Técnicas de optimización:

Predicate pushdown: Filtrar temprano en la pipeline
Projection pushdown: Seleccionar solo columnas necesarias
Bloom filters: Acelerar filtros de existencia