Laboratorio 15: Cassandra para Datos IoT en Tiempo Real

Prof. Heider Sanchez
ACLs: Ana María Accilio, Sebastián Loza

Problema:

El Internet de las Cosas (IoT) es un campo donde los dispositivos conectados generan y envían datos de manera continua. Estos dispositivos incluyen desde sensores en fábricas y automóviles hasta dispositivos médicos y electrodomésticos inteligentes. Todos estos dispositivos generan **flujos masivos de datos en tiempo real** que necesitan ser capturados, almacenados y analizados rápidamente para extraer información valiosa y permitir respuestas inmediatas. Este tipo de carga de trabajo plantea serios desafíos para las bases de datos tradicionales debido a la **velocidad**, **volumen** y **variedad** de los datos involucrados.

Imaginemos una **red de sensores IoT** desplegada en una planta industrial, donde se monitorean variables críticas como la temperatura, la presión, la vibración de las máquinas y la humedad. Estos sensores generan miles de puntos de datos por segundo, y los administradores necesitan acceder a estos datos en tiempo real para detectar anomalías, prever fallos y realizar ajustes de manera proactiva. Los datos recientes pueden ser críticos para el análisis en tiempo real, **mientras que los datos más antiguos pueden volverse irrelevantes con el tiempo**.

Solución con el Modelo Relacional de Tablas:

Las bases de datos relacionales tradicionales (RDBMS), aunque útiles para muchos casos de uso, no están diseñadas para manejar este tipo de flujos de datos masivos de manera eficiente. Las bases de datos relacionales están optimizadas para **consultas ACID** (transacciones atómicas, consistentes, aisladas y duraderas) y para datos que no cambian con tanta frecuencia. En un escenario de IoT, donde los datos llegan constantemente y por grandes cantidades, el *rendimiento de una base de datos relacional puede degradarse rápidamente*.

Ejemplo: supongamos que la planta industrial necesita gestionar la información generada por sus 1000 sensores de temperatura, los cuales están distribuidos en diferentes equipos y áreas del establecimiento. Si modelamos bajo un esquema relacional tendríamos la siguiente tabla y consulta:

```
Temperatura(sensorId, horaEvento, valor)

Select * from Temperatura where sensorId = "123ADC"
and horaEvento > '2018-04-03 07:01:00'
and horaEvento < '2018-04-03 07:04:00';
```

Dado el volumen de datos y las constantes consultas para el análisis y modelamiento matemático, se decide indexar por (sensorId, horaEvento). Pero esto a su vez **eleva la complejidad de inserción** de transmisión de datos en un BD relacional.

Ahora, supongamos que la empresa desea integrar la información de todas sus plantas industriales, tanto las ubicadas en el país como las del extranjero, por lo que es necesario ampliar la cantidad de servidores y replicación de datos. En otras palabras, un sistema distribuido de datos. La tabla quedaría expresada de la siguiente manera:

```
Temperatura(sedeId, sensorId, horaEvento, valor)
```

Lo cual agrega desafíos de almacenamiento replicado, indexación distribuida, consultas distribuidas, etc.

Por lo tanto: Un sistema de base de datos relacional tradicional (RDBMS) no es suficiente cuando se trata de recuperar datos de series de tiempo de manera eficiente en ambientes distribuidos. La sobrecarga generada por los optimizadores de consulta y administración de transacciones no utilizados, junto con la recuperación fila por fila forzada por esquemas en estrella, no permite tiempos de respuesta eficientes. Y nuevamente, escalar un RDBMS es casi imposible.

- La solución sería utilizar bases de datos especializadas para series de tiempo (TSDB), basados en tecnologías de código abierto NoSQL, y un modelo de datos flexible para superar dichas deficiencias.
- **Apache Cassandra** se considera como la base de datos de elección cuando se recopilan eventos de series de tiempo. Estos pueden ser mensajes, eventos o transacciones similares que tienen un elemento de tiempo asociado. Cassandra destaca sobre HBase por su fácil desarrollo (CQL parecido a SQL), arquitectura simple (sin-maestro) con fácil instalación y pocos requerimientos, multi-datacenter, y mayor desempeño.

Solución con Cassandra

Cassandra está diseñada específicamente para manejar **escrituras y lecturas masivas con baja latencia** y para **escalar horizontalmente**, lo que permite añadir servidores fácilmente a medida que crece el volumen de datos. A continuación, explicamos las características que hacen que Cassandra sea una opción recomendada para el análisis de datos IoT en tiempo real:

Como habiamos mencionado, la planta industrial está monitoreada por una red de sensores IoT que capturan datos de variables clave como la **temperatura** de diferentes máquinas. Cada sensor envía miles de datos por segundo a un servidor central donde estos datos se almacenan en una base de datos Cassandra. Un modelo típico de almacenamiento en Cassandra solo para la **temperatura** podría ser de la siguiente manera:


```
CREATE TABLE temperature (
    sensor_id TEXT,
    event_time TIMESTAMP,
    temperature DOUBLE,
    PRIMARY KEY (sensor_id, event_time)
);
```

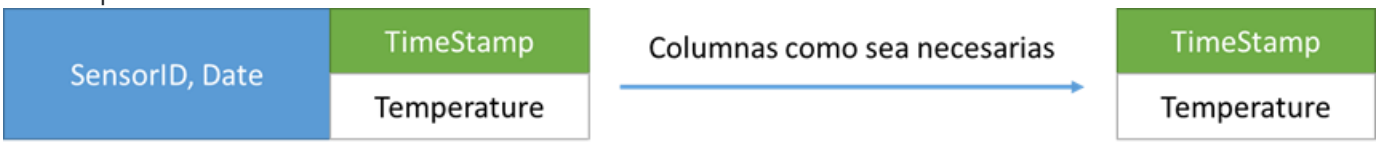
Insertamos datos:

```
INSERT INTO temperature(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:01:00', 25);
INSERT INTO temperature(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:02:00',24);
INSERT INTO temperature(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:03:00',24);
INSERT INTO temperature(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:04:00',25);
```

Aplicamos consulta:

```
SELECT temperature FROM temperature
WHERE sensor_id='1234ABCD'
AND event_time > '2018-04-03 07:01:00'
AND event_time < '2018-04-03 07:04:00';
```

Este modelo puede llevar tal vez a tener demasiadas columnas para una sola fila. Podemos entonces usar el siguiente modelo para distribuir la data de un mismo sensor en diferentes filas.




```
CREATE TABLE temperature_day (
   sensor_id TEXT,
   date TEXT,
   event_time TIMESTAMP,
   temperature DOUBLE,
   PRIMARY KEY ((sensor_id, date), event_time)
);
```

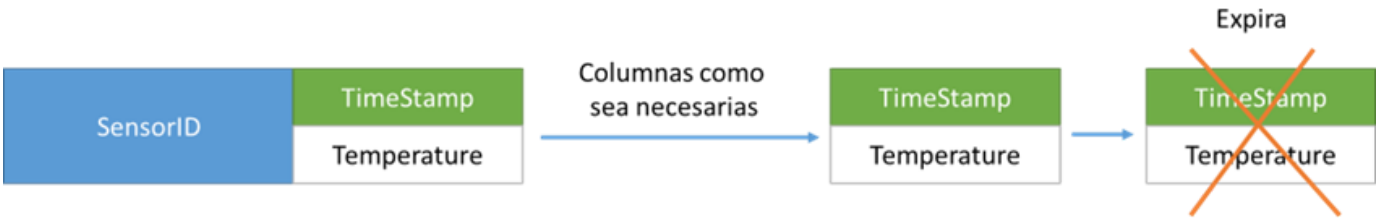
Insertamos datos:

```
INSERT INTO temperature_day(sensor_id, date, event_time,temperature)
VALUES ('1234ABCD','2018-04-03','2018-04-03 07:01:00', 24);
INSERT INTO temperature_day(sensor_id, date, event_time,temperature)
VALUES ('1234ABCD','2018-04-03','2018-04-03 07:02:00', 25);
INSERT INTO temperature_day(sensor_id, date, event_time,temperature)
VALUES ('1234ABCD','2018-04-04','2018-04-04 07:01:00',23);
INSERT INTO temperature_day(sensor_id, date, event_time,temperature)
VALUES ('1234ABCD','2018-04-04','2018-04-04 07:02:00', 24);
```

Aplicamos consulta:

```
SELECT * FROM temperature_day
WHERE sensor_id = '1234ABCD'
AND date ='2018-04-03';
```

En sistemas de series de tiempo el almacenamiento es continuo, pero en muchos casos las aplicaciones de analítica suelen desechar la información histórica después de extraer los patrones relevantes o modelos descriptivos compactos. Por lo que los datos más antiguos ya no son útiles, entonces pueden ser eliminados eventualmente. Cassandra ofrece una característica llamada columnas que caducan para que nuestros datos desaparezcan silenciosamente después de una cantidad establecida de segundos.



Para gestionar este ciclo de vida de los datos en Cassandra, se puede utilizar la característica de TTL (Time to Live) para definir un tiempo de expiración de los datos.

```
CREATE TABLE latest_temperatures (
   sensor_id text,
   event_time timestamp,
   temperature double,
   PRIMARY KEY (sensor_id, event_time)
) WITH default_time_to_live = 2592000;  -- 30 días en segundos
```

default_time_to_live = 2592000: Define que todos los datos en esta tabla tendrán un TTL de 30 días (30 días = 30 * 24 * 60 * 60 segundos).

Insertamos datos:

```
INSERT INTO latest_temperatures(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:03:00',22);
INSERT INTO latest_temperatures(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:02:00',23);
```

Suponiendo que queremos que las lecturas críticas de temperatura expiren después de solo 1 día (86400 segundos).

```
INSERT INTO latest_temperatures(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:01:00', 24) USING TTL 86400;
INSERT INTO latest_temperatures(sensor_id,event_time,temperature)
VALUES ('1234ABCD','2018-04-03 07:04:00', 24) USING TTL 86400;
```

Verificar los datos:

```
SELECT sensor_id,event_time, temperature, TTL(temperature)
FROM latest_temperatures;
```

Parte Evaluada

P1. (7 pts) Medir la temperatura y la humedad:

Expandir el modelo de datos para soportar múltiples tipos de mediciones (temperatura y humedad) y aplicar conceptos de TTL para la gestión del ciclo de vida de los datos.

1. Modificar la estructura de la tabla:

- Diseñar e implementar una nueva tabla que permita almacenar tanto mediciones de temperatura como de humedad

- La tabla debe incluir una columna `measurement_type` que pueda tomar los valores 'temperatura' o 'humedad'
- Considerar el mejor diseño de partition key y clustering key para optimizar las consultas.
- Aplicar un TTL (Time to Live) apropiado para la gestión automática del ciclo de vida de los datos.

2. Insertar datos de prueba:

- Generar al menos 10,000 registros con datos realistas de diferentes sensores capturados periodicamente (ejemplo, cada 30 segundos)
- Incluir mediciones de temperatura (rango: 15-35°C) y humedad (rango: 30-80%)
- Los datos deben cubrir al menos los últimos 7 días.
- Implementar diferentes valores de TTL para simular diferentes políticas de retención

3. Consultas requeridas:

- **Obtener todas las mediciones de humedad del sensor 'SENS001' del último día**. La consulta debe incluir el tiempo restante de vida (TTL) de cada registro.
- **Detectar valores anómalos fuera del rango permitido en la última hora**. Implementar una consulta que identifique mediciones de temperatura o humedad fuera del rango normal. La consulta debe permitir filtrar por sensor específico.
- **Verificar el tiempo restante de vida de los datos usando la función TTL**. Implementar una consulta que muestre el TTL en diferentes unidades (segundos, horas, días). Crear una consulta para identificar datos que están próximos a expirar (ej: en las próximas 24 horas).

4. Entregable:

- Documentar el diseño de la consulta con su respectivo comentario
- Medir y reportar los tiempos de respuesta de cada consulta

P2. (13 pts) Evaluación Experimental

Realizar una comparación experimental del rendimiento entre un cluster de Cassandra y PostgreSQL para cargas de trabajo de series de tiempo, evaluando tanto la escalabilidad de escritura como la eficiencia de consultas en un entorno distribuido.

1. Configuración del Entorno:

a) Cluster de Cassandra con Docker Compose:

- Configurar un cluster de Cassandra de al menos **3 nodos** utilizando Docker Compose
- Configurar el keyspace con estrategia de replicación apropiada y factor de replicación = 3
- Verificar el estado del cluster y la distribución de datos. Evidenciar.

b) PostgreSQL:

- Configurar una instancia de PostgreSQL (puede ser local o en contenedor)

c) Jupyter Notebook:

- Implementar la conexión a ambas bases de datos desde un **Jupyter Notebook**
- Utilizar las mismas especificaciones de hardware para ambas pruebas

2. Diseño de Esquemas Equivalentes:

Cassandra (en el cluster):

```
CREATE TABLE temperature_measurements (
   sensor_id TEXT,
   date TEXT,
   event_time TIMESTAMP,
   temperature DOUBLE,
   humidity DOUBLE,
   PRIMARY KEY ((sensor_id, date), event_time)
);
```

PostgreSQL:

```
CREATE TABLE temperature_measurements (
    sensor_id varchar(20),
    date varchar(10),
    event_time TIMESTAMP,
    temperature DOUBLE PRECISION,
    humidity DOUBLE PRECISION,
    PRIMARY KEY (sensor_id, date, event_time)
);
```

3. Generación de Datos de Prueba:

- Generador de datos sintéticos que produzca series de tiempo realistas
- Parámetros de datos:
  - Sensores: 100
  - Frecuencia: 1 lectura por minuto por sensor
  - Rango temporal: 7, 15, 30, 60 días

4. Experimentos de Rendimiento:

a) Pruebas de Escritura (INSERT):

- Medir tiempo de inserción para diferentes volúmenes: 7, 15, 30, 60 días.
- Evaluar inserción individual vs. inserción en lotes (batch)
- Para Cassandra: Medir el impacto de la replicación en 3 nodos
- Capturar el tiempo total

b) Pruebas de Lectura (SELECT): Implementar las siguientes consultas de prueba:

- **Consulta por sensor y rango temporal:** Obtener datos de un sensor específico en un día
- **Consulta agregada:** Calcular temperatura promedio de la última hora para múltiples sensores
- **Consulta de rango de valores:** Encontrar lecturas anómalas (fuera de rango normal)

c) Pruebas de Escalabilidad y Distribución:

- Evaluar el rendimiento con diferentes tamaños de dataset
- Para Cassandra: Analizar cómo se distribuyen los datos entre los 3 nodos
- Evaluar la tolerancia a fallos (simular la caída de un nodo)

5. Análisis Comparativo:

- Crear gráficos comparativos de rendimiento entre el cluster de Cassandra y PostgreSQL

- Identificar escenarios donde la arquitectura distribuida de Cassandra ofrece ventajas

6. Entregable:

- **Archivo docker-compose.yml** para el cluster de Cassandra y evidencia de ejecución.
- **Notebook completo** con código documentado y ejecutable
- **Gráficos experimentales** que muestren:
  - Tiempo de inserción en ambas bases de datos
  - Tiempo de consulta en ambas bases de datos
  - Distribución de datos entre nodos del cluster
- **Análisis escrito** con conclusiones sobre:
  - Ventajas de la arquitectura distribuida vs. centralizada
  - Escenarios donde Cassandra cluster supera a PostgreSQL