Almacenamiento de datos en Apache Cassandra

Objetivo

Aprender qué son las tablas Wide Rows en Apache Cassandra, sus ventajas y desventajas, cómo configurarlas y desarrollar ejemplos prácticos que demuestran su utilidad.

Parte 1: Introducción a Wide Rows

1. **¿Qué son las Wide Rows?**
   - En Cassandra, las Wide Rows permiten almacenar múltiples filas dentro de una partición utilizando una clave de partición (Partition Key) y una o más claves de clustering (Clustering Keys).
   - Las filas dentro de una partición se ordenan por las claves de clustering, lo que las hace ideales para consultas secuenciales o basadas en rangos.
2. **Ventajas de Wide Rows**
   - **Alto rendimiento**: Diseñadas para consultas eficientes sobre grandes volúmenes de datos dentro de una partición.
   - **Flexibilidad**: Permiten ordenar y consultar los datos en función de las claves de clustering.
   - **Eficiencia de almacenamiento**: Los datos relacionados se agrupan en una partición, reduciendo la necesidad de accesos redundantes a disco.
3. **Desventajas**
   - **Tamaño de la partición**: Si las particiones son demasiado grandes, puede haber problemas de rendimiento.
   - **Limitaciones de escalabilidad**: Las consultas pueden sobrecargar un nodo si una partición se convierte en un "hotspot".

Parte 2: Configuración del entorno con Docker

1. **Instalar Docker**
   - Asegúrate de tener Docker instalado en tu máquina.
2. **Crear un contenedor Cassandra**
   - Ejecuta el siguiente comando para levantar un contenedor con Cassandra:

     ```
     docker run --name lab_cassandra -d -p 9042:9042 cassandra:latest
     ```

3. **Verificar el estado del contenedor**
   - Usa el comando:

     ```
     docker logs cassandra-widerows
     ```

- Asegúrate de que Cassandra esté ejecutándose correctamente.

4. **Acceder al cliente CQLSH**
   - Conéctate al contenedor:

```
docker exec -it cassandra-widerows cqlsh
```

Parte 3: Uso de Wide Rows

Caso 1 - Registro de logs de un sistema

1. **Crear la tabla**
   - Diseñamos una tabla para almacenar logs de eventos por dispositivo, ordenados por tiempo:

```
CREATE TABLE logs_by_device (
    device_id UUID,
    log_time TIMESTAMP,
    log_level TEXT,
    message TEXT,
    PRIMARY KEY (device_id, log_time)
) WITH CLUSTERING ORDER BY (log_time DESC);
```

2. **Insertar datos**
   - Insertamos algunos registros:

```
INSERT INTO logs_by_device (device_id, log_time, log_level,
message)
VALUES (uuid(), toTimestamp(now()), 'INFO', 'System
initialized');

INSERT INTO logs_by_device (device_id, log_time, log_level,
message)
VALUES (uuid(), toTimestamp(now()), 'ERROR', 'Disk failure
detected');
```

3. **Consultar los logs**
   - Consultamos los logs para un dispositivo específico:

```
SELECT * FROM logs_by_device WHERE device_id = <device_id>
ORDER BY log_time DESC;
```

Recuerda: la estructura Wide Rows permite ordenar los logs por tiempo para una rápida recuperación, ideal para análisis de eventos cronológicos.

Caso 2 - Ventas por usuario

1. **Crear la tabla**

- Registramos las compras de cada usuario, ordenadas por fecha:

```
CREATE TABLE sales_by_user (
    user_id UUID,
    purchase_time TIMESTAMP,
    product_id UUID,
    amount DECIMAL,
    PRIMARY KEY (user_id, purchase_time)
) WITH CLUSTERING ORDER BY (purchase_time DESC);
```

2. **Insertar datos**
   - Insertamos algunos registros:

```
INSERT INTO sales_by_user (user_id, purchase_time,
product_id, amount)
VALUES (uuid(), toTimestamp(now()), uuid(), 99.99);

INSERT INTO sales_by_user (user_id, purchase_time,
product_id, amount)
VALUES (uuid(), toTimestamp(now()), uuid(), 49.99);
```

3. **Consultar las ventas**
   - Consultamos las ventas de un usuario específico:

```
SELECT * FROM sales_by_user WHERE user_id = <user_id> ORDER
BY purchase_time DESC;
```

La estructura Wide Rows permite consultar todas las compras de un usuario en orden cronológico, útil para personalización y análisis de tendencias.

Uso de Colecciones (Sets, Lists, Maps) en Apache Cassandra

Objetivo

Aprender a trabajar con colecciones en Apache Cassandra (Sets, Lists, Maps) a través de casos de uso prácticos y ejercicios interactivos. Este taller incluye operadores avanzados como **TTL (Time to Live)** y otros comandos útiles de Cassandra.

Parte 1: Casos de uso y ejercicios prácticos

Caso de uso 1 - Mensajería de textos (Lists)

- **Descripción**: Diseñaremos una tabla que almacena los últimos mensajes enviados por cada usuario utilizando una lista. Los mensajes estarán ordenados por tiempo de envío.

1. **Crear la tabla**:

```
CREATE TABLE user_messages (
    user_id UUID PRIMARY KEY,
    messages LIST<TEXT>
);
```

2. **Ejercicio 1: Insertar mensajes en la lista**:

```
UPDATE user_messages SET messages = messages + ['Hola, ¿cómo estás?'] WHERE user_id = uuid();
UPDATE user_messages SET messages = messages + ['¿Qué tal tu día?'] WHERE user_id = uuid();
```

3. **Ejercicio 2: Recuperar mensajes del usuario**:

```
SELECT messages FROM user_messages WHERE user_id = <user_id>;
```

4. **Ejercicio 3: Limitar la cantidad de mensajes en la lista**:

```
UPDATE user_messages SET messages = messages[-5:] WHERE user_id = <user_id>;
```

En Cassandra, las listas se manejan dentro de una única fila, eliminando la necesidad de múltiples tablas relacionadas.

Caso de uso 2 - Red de sensores (Sets)

- **Descripción**: Diseñaremos una tabla para almacenar los sensores activos en una red. Usaremos un set para garantizar que no haya duplicados.

1. **Crear la tabla**:

```
CREATE TABLE network_sensors (
    network_id UUID PRIMARY KEY,
    active_sensors SET<TEXT>
);
```

2. **Ejercicio 4: Agregar sensores al set**:

```
UPDATE network_sensors SET active_sensors = active_sensors +
{'sensor_1', 'sensor_2'} WHERE network_id = uuid();
```

3. **Ejercicio 5: Remover un sensor del set**:

```
UPDATE network_sensors SET active_sensors = active_sensors -
{'sensor_1'} WHERE network_id = <network_id>;
```

4. **Ejercicio 6: Consultar sensores activos**:

```
SELECT active_sensors FROM network_sensors WHERE network_id =
<network_id>;
```

En bases relacionales, necesitarías una tabla secundaria con una relación uno-a-muchos. Cassandra simplifica esta gestión con sets.

Caso de uso 3: Reportes de información en farmacias (Maps)

- **Descripción**: Diseñaremos una tabla para registrar los medicamentos vendidos por cada farmacia. Usaremos un mapa para asociar nombres de medicamentos con la cantidad vendida.

1. **Crear la tabla**:

```
CREATE TABLE pharmacy_sales (
    pharmacy_id UUID PRIMARY KEY,
    sales MAP<TEXT, INT>
);
```

2. **Ejercicio 7: Registrar ventas de medicamentos**:

```
UPDATE pharmacy_sales SET sales['Paracetamol'] = 50,
sales['Ibuprofeno'] = 30 WHERE pharmacy_id = uuid();
```

3. **Ejercicio 8: Actualizar la cantidad de un medicamento**:

```
UPDATE pharmacy_sales SET sales['Paracetamol'] = 60 WHERE
pharmacy_id = <pharmacy_id>;
```

4. **Ejercicio 9: Consultar las ventas por farmacia**:

```
SELECT sales FROM pharmacy_sales WHERE pharmacy_id =
<pharmacy_id>;
```

En bases relacionales, necesitarías varias filas o una tabla adicional para representar estas relaciones, mientras que Cassandra lo resuelve dentro de una sola fila.

Caso de uso 4: Expiración automática de datos con TTL

- **Descripción**: Diseñaremos una tabla para almacenar reportes temporales de calidad de aire por ciudad. Usaremos **TTL** (Time to Live) para que los reportes se eliminen automáticamente después de 24 horas.

1. **Crear la tabla**:

```
CREATE TABLE air_quality (
    city TEXT PRIMARY KEY,
    report TEXT,
    created_at TIMESTAMP
);
```

2. **Ejercicio 10: Insertar un reporte con TTL**:

```
INSERT INTO air_quality (city, report, created_at)
VALUES ('Madrid', 'Calidad del aire: Buena', toTimestamp(now()))
USING TTL 86400;
```

3. **Ejercicio 11: Verificar el tiempo restante del TTL**:

```
SELECT TTL(report) FROM air_quality WHERE city = 'Madrid';
```

4. **Ejercicio 12: Consultar los reportes activos**:

```
SELECT * FROM air_quality;
```

En bases relacionales, la eliminación automática requeriría programar un script o tarea cron; Cassandra lo maneja de forma nativa con TTL.

Clustering Columns y Storage Attached Indexes (SAI) en Apache Cassandra

Objetivo

Aprender a trabajar con **Clustering Columns** y **Storage Attached Indexes (SAI)** en Apache Cassandra.

Clustering Columns

¿Qué son las Clustering Columns?

- Son columnas en la clave primaria que determinan el orden de los datos dentro de una partición.
- Permiten organizar y consultar los datos de forma eficiente dentro de una partición.
- Utilizan la cláusula WITH CLUSTERING ORDER BY para especificar el orden.

Caso de Uso: Registro de actividad de usuarios

- **Descripción**: Diseñaremos una tabla para almacenar el historial de actividades de usuarios en una aplicación, ordenando las actividades por fecha y hora en orden descendente.

1. **Crear la tabla**:

```
CREATE TABLE user_activity (
    user_id UUID,
    activity_time TIMESTAMP,
    activity_type TEXT,
    description TEXT,
    PRIMARY KEY (user_id, activity_time)
) WITH CLUSTERING ORDER BY (activity_time DESC);
```

2. **Ejercicio 1: Insertar datos de actividad**:

```
INSERT INTO user_activity (user_id, activity_time, activity_type,
description)
VALUES (uuid(), toTimestamp(now()), 'LOGIN', 'User logged in');

INSERT INTO user_activity (user_id, activity_time, activity_type,
description)
VALUES (uuid(), toTimestamp(now()), 'UPLOAD', 'Uploaded a
document');
```

3. **Ejercicio 2: Consultar las últimas actividades de un usuario**:

```
SELECT * FROM user_activity WHERE user_id = <user_id> LIMIT 5;
```

4. **Ejercicio 3: Consultar actividades dentro de un rango de tiempo**:

```
SELECT * FROM user_activity
WHERE user_id = <user_id>
AND activity_time >= '2025-01-01T00:00:00'
AND activity_time <= '2025-01-05T23:59:59';
```

Caso de Uso: Transacciones bancarias

- **Descripción**: Diseñaremos una tabla para almacenar transacciones bancarias, ordenadas por fecha en orden ascendente.

1. **Crear la tabla**:

```
CREATE TABLE bank_transactions (
    account_id UUID,
    transaction_time TIMESTAMP,
    amount DECIMAL,
    description TEXT,
    PRIMARY KEY (account_id, transaction_time)
) WITH CLUSTERING ORDER BY (transaction_time ASC);
```

2. **Ejercicio 4: Insertar transacciones**:

```
INSERT INTO bank_transactions (account_id, transaction_time,
amount, description)
VALUES (uuid(), toTimestamp(now()), 500.00, 'Deposit');

INSERT INTO bank_transactions (account_id, transaction_time,
amount, description)
VALUES (uuid(), toTimestamp(now()), -200.00, 'Withdrawal');
```

3. **Ejercicio 5: Consultar transacciones en orden cronológico**:

```
SELECT * FROM bank_transactions WHERE account_id = <account_id>;
```

Storage Attached Indexes (SAI)

¿Qué son los Storage Attached Indexes?

- Introducidos en Cassandra 4.0 y mejorados en 5.0.
- Permiten crear índices eficientes en columnas no clave para consultas avanzadas.
- Mejoran el rendimiento en comparación con índices secundarios tradicionales.

Caso de Uso: Catálogo de productos

- **Descripción**: Diseñaremos una tabla para almacenar información de productos en un catálogo. Queremos buscar productos por nombre o descripción.

1. **Crear la tabla**:

```
CREATE TABLE product_catalog (
    product_id UUID PRIMARY KEY,
    name TEXT,
    description TEXT,
    price DECIMAL
);
```

2. **Ejercicio 6: Crear un índice SAI en la columna name**:

```
CREATE CUSTOM INDEX ON product_catalog (name)
USING 'StorageAttachedIndex';
```

3. **Ejercicio 7: Insertar productos en el catálogo**:

```
INSERT INTO product_catalog (product_id, name, description,
price)
VALUES (uuid(), 'Laptop', 'High-performance laptop', 1200.00);

INSERT INTO product_catalog (product_id, name, description,
price)
VALUES (uuid(), 'Smartphone', 'Latest model smartphone', 800.00);
```

4. **Ejercicio 8: Consultar productos por nombre**:

```
SELECT * FROM product_catalog WHERE name = 'Laptop';
```

Caso de Uso: Reportes de sensores

- **Descripción**: Diseñaremos una tabla para almacenar reportes de sensores en diferentes ubicaciones, permitiendo búsquedas rápidas por ubicación y tipo de sensor.

1. **Crear la tabla**:

```
CREATE TABLE sensor_reports (
    report_id UUID PRIMARY KEY,
    location TEXT,
    sensor_type TEXT,
    report_data TEXT,
    timestamp TIMESTAMP
```

```
);
```

2. **Ejercicio 9: Crear un índice SAI en la columna location**:

```
CREATE CUSTOM INDEX ON sensor_reports (location)
USING 'StorageAttachedIndex';
```

3. **Ejercicio 10: Insertar reportes de sensores**:

```
INSERT INTO sensor_reports (report_id, location, sensor_type,
report_data, timestamp)
VALUES (uuid(), 'Warehouse A', 'Temperature', '25°C',
toTimestamp(now()));

INSERT INTO sensor_reports (report_id, location, sensor_type,
report_data, timestamp)
VALUES (uuid(), 'Warehouse B', 'Humidity', '60%',
toTimestamp(now()));
```

4. **Ejercicio 11: Consultar reportes por ubicación**:

```
SELECT * FROM sensor_reports WHERE location = 'Warehouse A';
```

Importante: **no es posible usar Storage Attached Index (SAI)** sobre campos de tipo **SET**, **LIST** o **MAP** en Apache Cassandra. Los **SAI** están diseñados para trabajar únicamente con columnas de datos simples y algunos tipos de datos compuestos (como TEXT, INT, UUID, etc.), pero no soportan colecciones.

Razones por las que no se puede usar SAI en colecciones

1. **Complejidad del índice**:
   - Los índices en colecciones como SET, LIST o MAP requerirían manejar múltiples valores por fila, lo que incrementa significativamente la complejidad y el costo computacional de mantener el índice.
   - Esto iría en contra del diseño eficiente de Cassandra, que busca minimizar la sobrecarga.
2. **Alternativas existentes**:
   - Para búsquedas dentro de colecciones, Cassandra ya proporciona operadores como **CONTAINS** que permiten realizar consultas directas en colecciones sin necesidad de índices adicionales.

Alternativa: Usar operadores en colecciones

En lugar de un SAI, puedes usar operadores específicos para realizar búsquedas en colecciones. Aquí hay algunos ejemplos prácticos:

Ejemplo 1: Uso de SET con el operador CONTAINS

1. **Crear la tabla**:

```
CREATE TABLE user_tags (
    user_id UUID PRIMARY KEY,
    tags SET<TEXT>
);
```

2. **Insertar datos**:

```
INSERT INTO user_tags (user_id, tags) VALUES (uuid(), {'sports',
'music', 'travel'});
```

3. **Consultar usuarios que tienen un tag específico**:

```
SELECT * FROM user_tags WHERE tags CONTAINS 'sports';
```

Ejemplo 2: Uso de LIST con el operador CONTAINS

1. **Crear la tabla**:

```
CREATE TABLE user_messages (
    user_id UUID PRIMARY KEY,
    messages LIST<TEXT>
);
```

2. **Insertar datos**:

```
UPDATE user_messages SET messages = messages + ['Hello, how are
you?'] WHERE user_id = uuid();
```

3. **Consultar usuarios que tienen un mensaje específico en la lista**:

```
SELECT * FROM user_messages WHERE messages CONTAINS 'Hello, how
are you?';
```

Ejemplo 3: Uso de MAP con el operador CONTAINS

1. **Crear la tabla**:

```
CREATE TABLE user_prefs (
    user_id UUID PRIMARY KEY,
    preferences MAP<TEXT, TEXT>
);
```

2. **Insertar datos**:

```
INSERT INTO user_prefs (user_id, preferences) VALUES (uuid(),
{'theme': 'dark', 'language': 'en'});
```

3. **Consultar usuarios que tienen un valor específico en el mapa**:

```
SELECT * FROM user_prefs WHERE preferences CONTAINS 'dark';
```

4. **Consultar usuarios que tienen una clave específica en el mapa**:

```
SELECT * FROM user_prefs WHERE preferences CONTAINS KEY 'theme';
```