Лабораторна робота №5

Робота з базовими функціями БД типу column family на прикладі Cassandra

Мета роботи

Ознайомитися з принципами роботи колоночних (column family) баз даних на прикладі Apache Cassandra. Навчитися моделювати дані в CQL, налаштовувати кластер з кількох вузлів за допомогою Docker, дослідити механізми реплікації, рівні узгодженості (Consistency Levels) та їх вплив на доступність і продуктивність системи (CAP-теорема).

Технічне забезпечення та інструменти

1. **Операційна система:** Kali Linux (Virtual Environment).
2. **Середовище розгортання:** Docker, Docker Compose.
3. **СКБД:** Apache Cassandra (image: cassandra:latest).
4. **Мова запитів:** CQL (Cassandra Query Language).
5. **Мова програмування (для тестів):** Python 3 + cassandra-driver.

Частина 1. Робота зі структурами даних у Cassandra

Створення та структура таблиці items

Завдання: Створити таблицю для товарів, де категорія є ключем партиціювання (для швидкого пошуку), а ціна — кластерним ключем (для сортування).

```
cqlsh:shop> DESCRIBE TABLE orders;

CREATE TABLE shop.orders (
    customer_name text,
    order_time timestamp,
    order_id uuid,
    total_price decimal,
    item_ids list<uuid>,
    PRIMARY KEY (customer_name, order_time, order_id)
) WITH CLUSTERING ORDER BY (order_time DESC, order_id ASC)
    AND additional_write_policy = '99p'
    AND allow_auto_snapshot = true
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND incremental_backups = true
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
```

На скріншоті відображено результат команди DESCRIBE TABLE items.

- **Primary Key:** ((category), price, id). Тут category виступає Partition Key (забезпечує групування товарів на вузлах), а price та id — Clustering Keys.
- **Clustering Order:** Вказано ORDER BY (price ASC), що забезпечує фізичне зберігання даних на диску у відсортованому вигляді від найдешевшого до найдорожчого в межах категорії.
- **Map:** Поле properties має тип map<text, text>, що дозволяє зберігати динамічний набір характеристик для різних типів товарів. Також створено індекси для поля name та ключів мапи properties.

Сортування даних (Clustering Key)

Завдання: Вивести всі товари в категорії, відсортовані за ціною.

```
cqlsh:shop> SELECT * FROM items WHERE category = 'Electronics';

 category    | price | id                                   | name       | producer | properties
-------------+-------+--------------------------------------+------------+----------+-----------------------------------------------
 Electronics |   500 | cb1a47b3-2eb3-4868-90f0-5a2542886eee |    Phone Y |  Samsung | {'color': 'black', 'storage': '128GB'}
 Electronics |  1000 | 30982499-71c8-4c5f-b4fc-213ad931766f |   Laptop X |     Dell |          {'cpu': 'i7', 'ram': '16GB'}
 Electronics |  1200 | c409ae76-f27a-4387-8b6e-62b3dde71792 | Laptop Pro |    Apple |         {'color': 'gray', 'cpu': 'M1'}
```

Виконано запит SELECT * FROM items WHERE category = 'Electronics'. Результат показує три товари. Важливо зазначити, що вони вивелися автоматично відсортованими за ціною: 500 -> 1000 -> 1200.

Це підтверджує коректну роботу налаштування CLUSTERING ORDER BY (price ASC), заданого при створенні таблиці. Сортування відбулося без накладних витрат на час виконання запиту (in-place sorting).

Пошук за критеріями (Secondary Index & Slice Query)

Завдання: Вибрати товари за назвою та за діапазоном ціни.

```
cqlsh:shop> SELECT * FROM items WHERE category = 'Electronics' AND name = 'Laptop X';

 category    | price | id                                   | name     | producer | properties
-------------+-------+--------------------------------------+----------+----------+-----------------------------------
 Electronics |  1000 | 30982499-71c8-4c5f-b4fc-213ad931766f | Laptop X |     Dell | {'cpu': 'i7', 'ram': '16GB'}

(1 rows)
cqlsh:shop> SELECT * FROM items WHERE category = 'Electronics' AND price > 600;

 category    | price | id                                   | name       | producer | properties
-------------+-------+--------------------------------------+------------+----------+-----------------------------------
 Electronics |  1000 | 30982499-71c8-4c5f-b4fc-213ad931766f |   Laptop X |     Dell |   {'cpu': 'i7', 'ram': '16GB'}
 Electronics |  1200 | c409ae76-f27a-4387-8b6e-62b3dde71792 | Laptop Pro |    Apple | {'color': 'gray', 'cpu': 'M1'}
```

Пошук за назвою: Виконано запит з фільтром name = 'Laptop X'. Це стало можливим завдяки створеному вторинному індексу (CREATE INDEX item_name_idx). Без індексу Cassandra вимагала б ALLOW FILTERING, що є неефективним.

Пошук за ціною: Виконано запит price > 600. Оскільки price є частиною Primary Key (Clustering Key), Cassandra дозволяє ефективно виконувати діапазонні запити (Slice queries) в межах однієї партиції (category).

Робота з Materialized View

Завдання: Знайти товари певної категорії за виробником без використання ALLOW FILTERING.

```
cqlsh:shop> SELECT * FROM items_by_producer WHERE category = 'Electronics' AND producer = 'Apple';

 category    | producer | price | id                                   | name       | properties
-------------+----------+-------+--------------------------------------+------------+-----------------------------------
 Electronics |    Apple |  1200 | c409ae76-f27a-4387-8b6e-62b3dde71792 | Laptop Pro | {'color': 'gray', 'cpu': 'M1'}
```

Оскільки поле producer не є ключовим у таблиці items, прямий пошук по ньому неможливий без сканування всієї таблиці. Для вирішення цієї задачі було створено Materialized View items_by_producer, де producer включено до Primary

Key. На скріншоті продемонстровано успішний запит до цього View. Cassandra автоматично підтримує синхронізацію даних між основною таблицею та View, дозволяючи виконувати швидкі запити за виробником.

Структура таблиці orders

Завдання: Створити таблицю замовлень для швидкого пошуку історії клієнта.

```
cqlsh:shop> DESCRIBE TABLE orders;

CREATE TABLE shop.orders (
    customer_name text,
    order_time timestamp,
    order_id uuid,
    total_price decimal,
    item_ids list<uuid>,
    PRIMARY KEY (customer_name, order_time, order_id)
) WITH CLUSTERING ORDER BY (order_time DESC, order_id ASC)
    AND additional_write_policy = '99p'
    AND allow_auto_snapshot = true
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND incremental_backups = true
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
```

Команда DESCRIBE TABLE orders показує структуру таблиці:

- **Primary Key:** ((customer_name), order_time, order_id). Це дозволяє отримати всі замовлення конкретного клієнта одним запитом.
- **Clustering Order:** ORDER BY (order_time DESC). Дані зберігаються так, що найновіші замовлення завжди знаходяться на початку (зверху). Це оптимізує сценарій "показати останні замовлення".

Вибірка та сортування замовлень

Завдання: Вивести замовлення клієнта відсортовані за часом.

```
cqlsh:shop> SELECT * FROM orders WHERE customer_name = 'Ivan';

 customer_name | order_time                      | order_id                             | item_ids                                                                                | total_price
---------------+---------------------------------+--------------------------------------+-----------------------------------------------------------------------------------------+-------------
          Ivan | 2026-01-20 10:19:09.284000+0000 | d945f9cf-7b02-4371-b57b-fefbc117d927 | [cadf860b-eb43-43f4-a87a-7f775f43f142, 9103835a-cd1d-43dc-a037-47a87f2fed40] |        1500
          Ivan | 2023-01-01 10:00:00.000000+0000 | c82f9374-bfe8-4741-be52-547012820030 |                                                                [e76c6c82-38b3-47c6-96ce-752f6f2fb5b9] |         200
```

Виконано запит для клієнта Ivan. У результаті отримано два замовлення. Завдяки налаштуванню кластеризації, замовлення від **2026 року** відображається першим, а від 2023 року — другим. Це підтверджує зворотне сортування (DESC) за часом.

Агрегація та метадані (WriteTime)

Завдання: Підрахувати загальну суму замовлень та перевірити час запису.

```
cqlsh:shop> SELECT sum(total_price) FROM orders WHERE customer_name = 'Ivan';

 system.sum(total_price)
-------------------------
                    1700
```

```
cqlsh:shop> SELECT total_price, WRITETIME(total_price) FROM orders WHERE customer_name = 'Ivan';

 total_price | writetime(total_price)
-------------+------------------------
        1500 |       1768904349279344
         200 |       1768904353256828
```

Агрегація: Функція sum(total_price) коректно підрахувала загальну вартість замовлень клієнта (1500 + 200 = 1700). У Cassandra агрегація виконується в межах партиції.

WriteTime: Функція WRITETIME(total_price) повернула час запису даних у мікросекундах (Unix timestamp). Ці мітки часу використовуються Cassandra для вирішення конфліктів (стратегія Last Write Wins) при реплікації даних між вузлами кластера.

Частина 2. Налаштування реплікації та кластеризація

Перевірка статусу кластера

Завдання: Сконфігурувати кластер з 3-х нод та перевірити його статус.

```
┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 nodetool status
Datacenter: DC1
═══════════════
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load        Tokens  Owns (effective)  Host ID                               Rack
UN  172.24.0.2  194.39 KiB  16      65.9%             dd17e5db-6d59-4f61-a184-899bf9bf8641  rack1
UN  172.24.0.3  80.08 KiB   16      64.7%             ac611352-2b9d-4a84-b4df-afdbfdf2e263  rack1
UN  172.24.0.4  85.23 KiB   16      69.5%             8c2fbd1a-d941-4076-af96-8cb2bd4e14a4  rack1
```

Команда nodetool status підтверджує, що кластер успішно створено та він складається з трьох вузлів (IP-адреси: 172.24.0.2, 172.24.0.3, 172.24.0.4). Статус UN (Up/Normal) навпроти кожного вузла свідчить про те, що всі ноди активні, бачать одна одну через протокол Gossip та готові до обробки запитів. Вони знаходяться в одному дата-центрі (DC1) та одній стійці (rack1).

Розподіл даних (Replication Factor)

Завдання: Створити Keyspaces з різними факторами реплікації (RF=1, 2, 3) та проаналізувати розподіл даних.

```
┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 nodetool status ks_rf1
Datacenter: DC1
═══════════════
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load        Tokens  Owns (effective)  Host ID                               Rack
UN  172.24.0.2  145.81 KiB  16      44.3%             dd17e5db-6d59-4f61-a184-899bf9bf8641  rack1
UN  172.24.0.3  102.7 KiB   16      24.9%             ac611352-2b9d-4a84-b4df-afdbfdf2e263  rack1
UN  172.24.0.4  107.84 KiB  16      30.7%             8c2fbd1a-d941-4076-af96-8cb2bd4e14a4  rack1


┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 nodetool status ks_rf2
Datacenter: DC1
═══════════════
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load        Tokens  Owns (effective)  Host ID                               Rack
UN  172.24.0.2  145.81 KiB  16      65.9%             dd17e5db-6d59-4f61-a184-899bf9bf8641  rack1
UN  172.24.0.3  102.7 KiB   16      64.7%             ac611352-2b9d-4a84-b4df-afdbfdf2e263  rack1
UN  172.24.0.4  107.84 KiB  16      69.5%             8c2fbd1a-d941-4076-af96-8cb2bd4e14a4  rack1


┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 nodetool status ks_rf3
Datacenter: DC1
═══════════════
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load        Tokens  Owns (effective)  Host ID                               Rack
UN  172.24.0.2  145.81 KiB  16      100.0%            dd17e5db-6d59-4f61-a184-899bf9bf8641  rack1
UN  172.24.0.3  102.7 KiB   16      100.0%            ac611352-2b9d-4a84-b4df-afdbfdf2e263  rack1
UN  172.24.0.4  107.84 KiB  16      100.0%            8c2fbd1a-d941-4076-af96-8cb2bd4e14a4  rack1
```

На скріншоті показано вивід команди nodetool status <keyspace>, яка відображає відсоток даних, за які відповідає кожен вузол (колонка Owns):

- **ks_rf1 (RF=1):** Дані розподілені між нодами (сумарно ~100%), але кожна конкретна порція даних зберігається лише на одному вузлі. В разі падіння вузла частина даних стане недоступною.
- **ks_rf2 (RF=2):** Кожен вузол відповідає приблизно за 66% кільця токенів.
- **ks_rf3 (RF=3):** У колонці **Owns** ми бачимо **100.0%** для кожного вузла. Це означає, що **кожен** рядок даних, записаний у цей Keyspace, фізично копіюється на всі три вузли кластера. Це забезпечує максимальну відмовостійкість.

Фізичне розташування даних

Завдання: Визначити IP-адреси вузлів, де зберігається конкретний запис.

```
┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 nodetool getendpoints ks_rf3 data 1
172.24.0.3
172.24.0.4
172.24.0.2
```

Команда nodetool getendpoints ks_rf3 data 1 показує, на яких саме машинах лежить рядок з Primary Key 1. Оскільки для ks_rf3 встановлено Replication Factor = 3, у результаті ми бачимо три IP-адреси (всі вузли кластера). Це підтверджує, що реплікація працює коректно і дані дублюються.

Вплив Consistency Level на доступність (CAP-теорема)

Завдання: Перевірити можливість читання при відключеній ноді з високим рівнем консистентності (THREE).

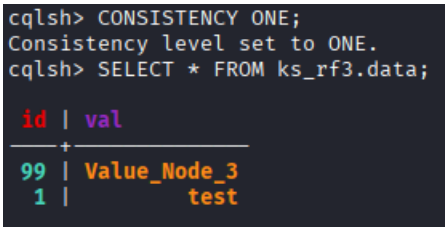
```
┌──(kali㉿kali)-[~]
└─$ docker exec -it cassandra-1 cqlsh
Connected to MyCluster at 127.0.0.1:9042
[cqlsh 6.2.0 | Cassandra 5.0.6 | CQL spec 3.4.7 | Native protocol v5]
Use HELP for help.
cqlsh> CONSISTENCY THREE;
Consistency level set to THREE.
cqlsh> SELECT * FROM ks_rf3.data;
NoHostAvailable: ('Unable to complete the operation against any hosts', {<Host: 127.0.0.1:9042 DC1>: Unavailable('Error from server: code=1000 [Unavailable exception] message="Cannot achieve consistency level THREE" info={\'consistency\': \'THREE\', \'required_replicas\': 3, \'alive_replicas\': 2}')})
```

Після відключення однієї з нод (у кластері залишилося 2 активні ноди), було виконано спробу читання з рівнем узгодженості CONSISTENCY THREE. Система повернула помилку UnavailableException.

Причина: Рівень THREE вимагає підтвердження від 3-х реплік. Оскільки в наявності тільки 2 (alive_replicas: 2), Cassandra відмовилася виконувати запит, щоб не порушити вимогу консистентності.

Відновлення доступності (Зниження Consistency Level)

Завдання: Перевірити можливість читання при відключеній ноді з низьким рівнем консистентності (ONE).

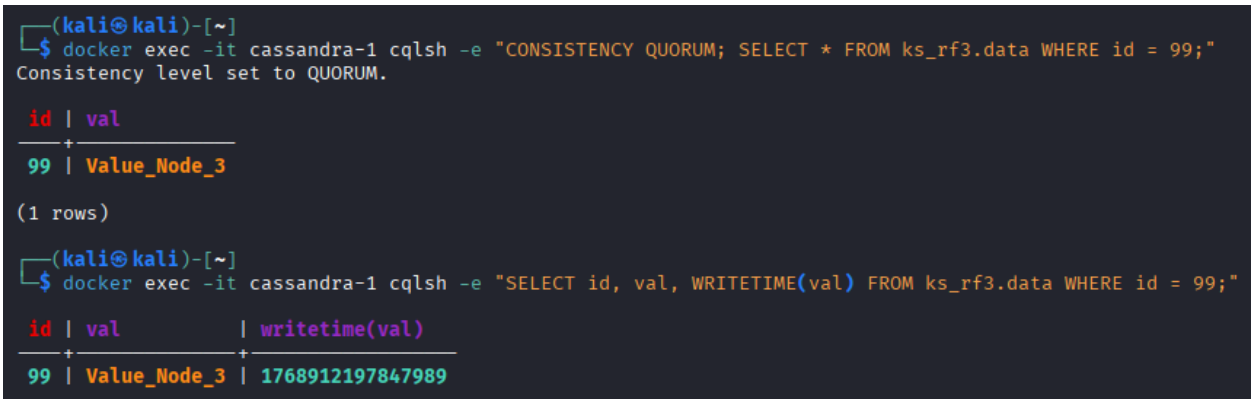


При тих самих умовах (одна нода відключена) рівень узгодженості було знижено до CONSISTENCY ONE. Запит виконався успішно.

Причина: Для рівня ONE достатньо відповіді від будь-якої однієї репліки. Оскільки у нас живі 2 ноди, система залишається доступною (High Availability), жертвуючи суворою узгодженістю.

Вирішення конфліктів (Split Brain & Last Write Wins)

Завдання: Створити конфлікт даних (мережева ізоляція) і перевірити, як Cassandra його вирішить після відновлення зв'язку.



Було змодельовано ситуацію Split Brain, коли на ізольовані частини кластера записали різні значення для одного ID (99). Після відновлення мережі було виконано запит SELECT з CONSISTENCY QUORUM, що спровокувало процес Read Repair.

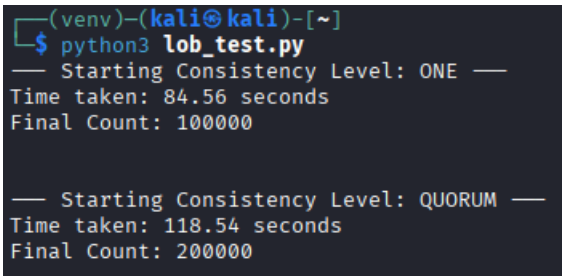
Результат: Кластер повернув значення Value_Node_3.

Пояснення (WriteTime): Команда WRITETIME(val) показує мікросекунди, коли було зроблено запис: 1768912197847989. Cassandra використовує стратегію Last Write Wins (LWW). Вона порівняла мітки часу конфліктуючих записів і залишила той, який мав новіший timestamp (той, що був зроблений на Node_3). Старе значення було перезаписане.

Частина 3. Аналіз продуктивності та перевірка цілісності

Результати навантажувального тестування (Python Script)

Завдання: Запустити 10 клієнтів по 10 000 запитів інкрементації кожен (сумарно 100 000 операцій) з різними рівнями консистентності (ONE та QUORUM) та порівняти час виконання.



Скрипт виконав два прогони тестів, результати яких відображено на скріншоті:

Consistency Level: ONE

Час виконання: 84.56 секунд.

Результат: 100 000.

Цей режим працює найшвидше, оскільки клієнт отримує підтвердження відразу після запису на першу доступну ноду, не чекаючи реплікації на інші. В умовах локального Docker-кластера втрат даних не зафіксовано.

Consistency Level: QUORUM

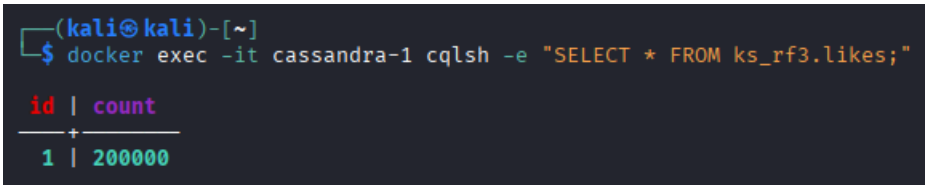
Час виконання: 118.54 секунд.

Результат: 200 000 (100 000 з попереднього тесту + 100 000 нових).

Час виконання збільшився приблизно на 40% (з 84с до 118с). Це очікувана поведінка, оскільки для успішного запису з рівнем QUORUM (при RF=3) координатор повинен отримати підтвердження мінімум від двох вузлів. Це забезпечує сувору узгодженість даних ціною зниження швидкості запису.

Перевірка фінального стану в базі даних

Завдання: Перевірити кінцеве значення лічильника в таблиці likes через консоль cqlsh.



Запит SELECT * FROM ks_rf3.likes показує, що для id=1 значення лічильника count дорівнює 200 000. Це підтверджує, що:

- Усі запити від обох тестів (100к + 100к) були успішно оброблені кластером.
- Тип даних counter у Cassandra коректно обробляє конкурентні запити на інкрементацію від багатьох потоків одночасно без стану гонитви (race condition).

Висновок

У ході виконання роботи було розгорнуто кластер Apache Cassandra з трьох вузлів у середовищі Docker. Було досліджено особливості моделювання даних (Partition keys, Clustering keys, Map, Materialized Views), налаштовано різні стратегії реплікації та перевірено стійкість системи до відмов (Partition Tolerance). Експериментально доведено, що підвищення рівня узгодженості (Consistency Level) гарантує цілісність даних при збоях, але знижує продуктивність запису.