CQL: zapytania i operacje na danych

Poziom: podstawy **Dotyczy: Apache Cassandra 3.x / 4.x** **Czas: ~35 min**

CQL wygląda jak SQL i w dużej mierze czyta się jak SQL, ale kilka jego ograniczeń — brak `JOIN`, ostrożne traktowanie `ALLOW FILTERING`, `BATCH`, który nie jest transakcją — wynika wprost z architektury opisanej na stronach Architektura Cassandry: pierścień, węzeł, replikacja i Model danych: keyspace, tabela, klucz partycji, klucz klastrujący.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
CQL	Cassandra Query Language: język zapytań Cassandry, podobny do SQL
DC	Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników
RF	Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

ZMIENIA SYSTEM

Utworzenie keyspace z prostą strategią replikacji: CREATE KEYSPACE

```
CREATE KEYSPACE NazwaKeyspace WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 3};
```

Tworzymy keyspace z jedną liczbą replikacji dla całego klastra, bez podziału na datacenter — sensowne wyłącznie dla klastra testowego w jednym DC. Różnicę między `SimpleStrategy` a `NetworkTopologyStrategy` i konsekwencje tego wyboru w klastrze produkcyjnym opisuje strona Architektura Cassandry: pierścień, węzeł, replikacja.

DC = Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników

ZMIENIA SYSTEM

Utworzenie keyspace ze strategią świadomą topologii: CREATE KEYSPACE

```
CREATE KEYSPACE NazwaKeyspace WITH replication = {'class': 'NetworkTopologyStrategy',
'NazwaDatacenter': 3};
```

Deklarujemy replication factor osobno dla każdego datacenter — przy kilku DC dopisujemy kolejne pary `'nazwa_dc': liczba` w tym samym słowniku. To jedyna strategia, po którą sięgamy w środowisku produkcyjnym.

> **Pojęcie: DURABLE_WRITES.** Flaga keyspace (domyślnie `true`) mówiąca, czy zapisy do tabel tego keyspace przechodzą przez commitlog opisany na stronie Utrzymanie klastra: repair, compaction, czyszczenie, zanim zostaną potwierdzone. Ustawienie `false` przyspiesza zapis kosztem utraty danych z memtable przy awarii węzła przed jego zrzutem na dysk — sensowne wyłącznie dla danych, których utratę akceptujemy (np. tymczasowe dane sesji), nigdy dla danych rozliczeniowych.

```
ALTER KEYSPACE NazwaKeyspace WITH REPLICATION = {'class': 'SimpleStrategy',
'replication_factor': 3} AND DURABLE_WRITES = false;
```

DC = Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników

ZMIENIA SYSTEM

Zmiana replication factor lub strategii keyspace: ALTER KEYSPACE

```
ALTER KEYSPACE NazwaKeyspace WITH replication = {'class': 'NetworkTopologyStrategy',
'NazwaDatacenter': 3};
```

Podnosimy albo obniżamy liczbę replik, ewentualnie zmieniamy samą strategię — operacja zmienia tylko definicję keyspace, nie przenosi automatycznie żadnych danych. Po podniesieniu replication factor nowe kopie danych trzeba dociągnąć ręcznie komendą `nodetool repair`, opisaną na stronie nodetool: stan klastra i operacje administracyjne — dopóki repair się nie wykona, nowa replika jest pusta, mimo że definicja keyspace już mówi inaczej.

ZMIENIA SYSTEM

Utworzenie tabeli ze złożonym kluczem i kolejnością klastrowania: CREATE TABLE

```
CREATE TABLE NazwaKeyspace.NazwaTabeli (
    klient_id uuid,
    data_zamowienia timestamp,
    id_zamowienia uuid,
    wartosc decimal,
    status text,
    PRIMARY KEY ((klient_id), data_zamowienia, id_zamowienia)
) WITH CLUSTERING ORDER BY (data_zamowienia DESC, id_zamowienia ASC);
```

Klucz partycji i kolumny klastrujące — kto z kim tworzy jedną fizyczną partycję i w jakiej kolejności wiersze tej partycji leżą na dysku — omawia strona Model danych: keyspace, tabela, klucz partycji, klucz klastrujący. `CLUSTERING ORDER BY` z `DESC` na pierwszej kolumnie klastrującej sprawia, że najnowsze zamówienia klienta leżą fizycznie na początku partycji — dokładnie w kolejności, w jakiej najczęściej chcemy je czytać.

ASC = Automatic Self-Calibration: automatyczna samokalibracja czujnika CO2, która zakłada, że co tydzień czujnik zobaczy czyste powietrze o stężeniu około 400 ppm

ZMIENIA SYSTEM

Dodanie kolumny do istniejącej tabeli: ALTER TABLE ADD

```
ALTER TABLE NazwaTabeli ADD kanal_sprzedazy text;
```

Dodajemy nową kolumnę do już istniejącej tabeli — wiersze zapisane przed tą zmianą po prostu nie mają wartości w nowej kolumnie (`null`), bez konieczności migracji danych wstecz.

ZMIENIA SYSTEM

Zmiana typu istniejącej kolumny: ALTER TABLE ALTER

```
ALTER TABLE NazwaTabeli ALTER kanal_sprzedazy TYPE text;
```

Zmieniamy typ kolumny na kompatybilny — Cassandra dopuszcza to wyłącznie między typami o zgodnej reprezentacji binarnej (np. `int` na `varint`), nie między dowolną parą

typów; nowsze wersje silnika ograniczają tę operację jeszcze mocniej niż starsze, więc przed wykonaniem warto sprawdzić dokumentację dokładnie dla używanej wersji.

◫◫ ZMIENIA SYSTEM

Zmiana parametrów cache'owania tabeli: ALTER TABLE WITH caching

```
ALTER TABLE NazwaTabeli WITH caching = {'keys': 'NONE', 'rows_per_partition': '1'};
```

Sterujemy tym, ile danych tabeli Cassandra trzyma w pamięci podręcznej węzła (key cache i row cache, widoczne też w wyniku `nodetool info` opisanym na stronie nodetool: stan klastra i operacje administracyjne) — `rows_per_partition: 1` cache'uje tylko pierwszy wiersz każdej partycji, sensowne dla tabel odpytywanych zawsze o najnowszy wpis danej partycji.

◫◫ OSTROŻNIE · NIEODWRACALNE

Usunięcie tabeli: DROP TABLE

```
DROP TABLE NazwaTabeli;
```

Usuwamy tabelę wraz z całą jej zawartością i definicją, nieodwracalnie — operacja propaguje się przez gossip do całego klastra, więc bezpośrednio po niej różne węzły mogą przez chwilę widzieć różną wersję schematu, co widać w wyniku `DESCRIBE CLUSTER` opisanym na stronie cqlsh: klient wiersza poleceń.

◫◫ TYLKO ODCZYT

Usunięcie keyspace: DROP KEYSPACE

```
DROP KEYSPACE NazwaKeyspace;
```

Usuwamy keyspace wraz ze wszystkimi tabelami, które zawiera — operacja o zasięgu całego klastra i bez możliwości cofnięcia, poza odtworzeniem z backupu opisanym na stronie Backup i odtwarzanie.

◫◫ ZMIENIA SYSTEM

Wstawienie wiersza: INSERT

```
INSERT INTO NazwaTabeli (klient_id, data_zamowienia, id_zamowienia, wartosc, status)
VALUES (klient_id, toTimestamp(now()), uuid(), 199.90, 'nowe');
```

Wstawiamy wiersz, podając wprost wszystkie kolumny klucza głównego — bez klucza partycji Cassandra nie wie, na który węzeł ten wiersz w ogóle skierować. `INSERT` na istniejącym już kluczu głównym nie zwraca błędu — nadpisuje istniejący wiersz, tak jakby to był `UPDATE`; w CQL nie ma standardowego rozróżnienia na „wstaw nowy" i „zaktualizuj istniejący" poza lekkimi transakcjami omówionymi niżej.

CQL = Cassandra Query Language: język zapytań Cassandry, podobny do SQL

ZMIENIA SYSTEM

Wstawienie wiersza z czasem życia: INSERT USING TTL

```
INSERT INTO NazwaTabeli (klient_id, data_zamowienia, id_zamowienia, wartosc, status)
VALUES (klient_id, toTimestamp(now()), uuid(), 199.90, 'nowe') USING TTL 3600;
```

> **Pojęcie: TTL (time to live).** Czas w sekundach, po którym Cassandra sama usunie wiersz (a właściwie zapisze dla niego tombstone, opisany na stronie Model danych: keyspace, tabela, klucz partycji, klucz klastrujący) bez żadnej dodatkowej akcji ze strony aplikacji. Sensowne dla danych z natury tymczasowych — tokenów sesji, kolejek zdarzeń do przetworzenia — gdzie ręczne czyszczenie starych wierszy zapytaniem `DELETE` byłoby dodatkowym zadaniem generującym dokładnie te same tombstone'y.

TTL = Time To Live: czas życia rekordu lub danych

ZMIENIA SYSTEM

Aktualizacja wiersza: UPDATE

```
UPDATE NazwaTabeli SET status = 'zrealizowane' WHERE klient_id = klient_id AND
data_zamowienia = data_zamowienia AND id_zamowienia = id_zamowienia;
```

Aktualizujemy kolumny wiersza wskazanego pełnym kluczem głównym — `WHERE` w `UPDATE` musi identyfikować dokładnie jeden wiersz (albo jedną partycję, jeśli aktualizujemy zakres kolumn klastrujących), nigdy dowolny warunek na zwykłej kolumnie, z tego samego powodu,

dla którego `SELECT` po zwykłej kolumnie wymaga `ALLOW FILTERING`.

SET = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

OSTROŻNIE · NIEODWRACALNE

Usunięcie wiersza lub kolumny: DELETE

```
DELETE FROM NazwaTabeli WHERE klient_id = klient_id AND data_zamowienia =
data_zamowienia AND id_zamowienia = id_zamowienia;
```

Usuwamy cały wiersz wskazany kluczem głównym — albo, podając nazwę kolumny przed `FROM` (`DELETE status FROM ...`), tylko jedną jej wartość, zostawiając resztę wiersza nietkniętą. Każde `DELETE`, niezależnie od zasięgu, zapisuje tombstone zamiast fizycznie zwalniać miejsce od razu — mechanikę i koszt tego wyjaśnia strona Model danych: keyspace, tabela, klucz partycji, klucz klastrujący.

TYLKO ODCZYT

Odczyt po kluczu partycji: SELECT

```
SELECT * FROM NazwaTabeli WHERE klient_id = klient_id;
```

Odczytujemy wszystkie wiersze jednej partycji — zapytanie trafia do dokładnie jednej trójki węzłów (przy RF = 3), bez przeszukiwania reszty klastra, bo klucz partycji jednoznacznie wskazuje, gdzie te dane leżą.

RF = Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry

TYLKO ODCZYT

Ograniczenie liczby zwracanych wierszy: SELECT LIMIT

```
SELECT * FROM NazwaTabeli WHERE klient_id = klient_id LIMIT 10;
```

Ograniczamy wynik do zadanej liczby wierszy z partycji — przydatne przy dużych partycjach, gdzie interesuje nas tylko kilka pierwszych wierszy zgodnie z `CLUSTERING ORDER BY` zadeklarowanym przy tworzeniu tabeli.

Przykład:

```
 klient_id                            | data_zamowienia                 |
id_zamowienia                         | status
---------------------------------------+---------------------------------
+--------------------------------------+--------------
 3c1f6e2a-1111-4b3a-9c2d-000000000001 | 2026-03-14 09:12:00.000000+0000 | 9f2e1a3b-4444-
5555-6666-777788889999 | zrealizowane
 3c1f6e2a-1111-4b3a-9c2d-000000000001 | 2026-03-10 14:03:00.000000+0000 | 7b8c9d0e-5555-
6666-7777-888899990000 | zrealizowane


(2 rows)
```

TYLKO ODCZYT

Odczyt po wielu wartościach klucza partycji: SELECT IN

```
SELECT * FROM NazwaTabeli WHERE klient_id IN (klient_id_1, klient_id_2);
```

Odpytujemy kilka konkretnych partycji jednym zapytaniem — coordinator rozsyła je do wszystkich wymienionych partycji naraz i scala wyniki. Warto ograniczać liczbę wartości w `IN` do niewielkiej garści (pojedyncze dziesiątki) — każda dodatkowa wartość to kolejna partycja, którą trzeba odpytać, więc korzyść ze zrównoleglenia szybko zamienia się w koszt obciążenia wielu węzłów jednym zapytaniem klienta.

TYLKO ODCZYT

Odczyt z pominięciem ograniczeń klucza partycji: SELECT ALLOW FILTERING

```
SELECT * FROM NazwaTabeli WHERE status = 'nowe' ALLOW FILTERING;
```

Wymuszamy wykonanie zapytania filtrującego po kolumnie spoza klucza partycji — Cassandra bez tej klauzuli takie zapytanie odrzuci, żeby nie wykonać niekontrolowanego skanu klastra po cichu. `ALLOW FILTERING` nie sprawia, że zapytanie staje się tanie — sprawia tylko, że silnik pozwala je uruchomić, mimo że w najgorszym razie przejrzy wszystkie partycje tabeli, odrzucając po drodze te, które nie pasują do warunku. W kodzie aplikacji obecność tej klauzuli jest sygnałem do przeprojektowania tabeli pod to zapytanie, nie do zaakceptowania jej jako rozwiązania — pełne rozwinięcie na stronie Model danych: keyspace,

tabela, klucz partycji, klucz klastrujący.

□□ TYLKO ODCZYT

Grupowanie wyników w obrębie partycji: SELECT GROUP BY

```
SELECT status, count(*) FROM NazwaTabeli WHERE klient_id = klient_id GROUP BY status;
```

`GROUP BY` w CQL grupuje wyłącznie w obrębie pojedynczej partycji, po kolumnach klucza głównego tej partycji — nie istnieje odpowiednik grupowania po dowolnej kolumnie w całej tabeli, z tego samego powodu, dla którego nie istnieją agregaty po wielu partycjach, opisanego na stronie Model danych: keyspace, tabela, klucz partycji, klucz klastrujący.

□□ **CQL** = Cassandra Query Language: język zapytań Cassandry, podobny do SQL

□□ OSTROŻNIE · NIEODWRACALNE

Usunięcie wszystkich danych z tabeli: TRUNCATE

```
TRUNCATE NazwaTabeli;
```

Usuwamy całą zawartość tabeli, zachowując jej strukturę — w przeciwieństwie do `DELETE` po wierszu, `TRUNCATE` nie zapisuje tombstone'ów dla każdego usuniętego wiersza, tylko każe węzłom porzucić bieżące pliki danych tej tabeli, więc jest znacznie tańszą operacją przy dużych tabelach, ale też bez możliwości odtworzenia inaczej niż z backupu.

□□ ZMIENIA SYSTEM

Utworzenie indeksu wtórnego: CREATE INDEX

```
CREATE INDEX NazwaIndeksu ON NazwaTabeli (status);
```

Pojęcie: indeks wtórny w Cassandrze. W przeciwieństwie do indeksu w bazie relacyjnej, opisanego na stronie Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli", indeks wtórny w Cassandrze jest budowany osobno na każdym węźle, wyłącznie z danych, które ten węzeł już przechowuje — więc zapytanie po indeksowanej kolumnie i tak musi trafić do wszystkich węzłów klastra, bo żaden pojedynczy węzeł nie ma globalnego obrazu tego, gdzie leżą pasujące wiersze. Indeks

wtórny ma sens wyłącznie dla kolumn o niskiej selektywności skoncentrowanej w niewielkiej liczbie partycji (np. status zamówienia w tabeli, w której zapytania i tak zwykle podają też `klient_id`) — nigdy jako ogólne rozwiązanie na „chcemy filtrować po dowolnej kolumnie", bo do tego służy zaprojektowanie osobnej tabeli pod to zapytanie, nie indeks.

⚠ OSTROŻNIE · NIEODWRACALNE

Usunięcie indeksu: DROP INDEX

```
DROP INDEX IF EXISTS NazwaIndeksu;
```

Usuwamy indeks wtórny — `IF EXISTS` zapobiega błędowi, jeśli indeks o tej nazwie już nie istnieje, co bywa wygodne w skryptach wykonywanych bez pewności co do aktualnego stanu schematu.

⚠ ZMIENIA SYSTEM

Utworzenie widoku zmaterializowanego: CREATE MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW NazwaKeyspace.zamowienia_wg_statusu AS
    SELECT * FROM NazwaKeyspace.zamowienia_wg_klienta
    WHERE status IS NOT NULL AND klient_id IS NOT NULL AND data_zamowienia IS NOT NULL
AND id_zamowienia IS NOT NULL
    PRIMARY KEY ((status), klient_id, data_zamowienia, id_zamowienia);
```

Pojęcie: widok zmaterializowany (materialized view). Automatycznie utrzymywana kopia danych z tabeli źródłowej, przepisana pod inny klucz partycji — Cassandra sama dopisuje wiersz do widoku przy każdym zapisie do tabeli źródłowej, oszczędzając aplikacji ręcznego pisania do dwóch tabel naraz, tak jak robilibyśmy to przy ręcznej denormalizacji opisanej na stronie Model danych: keyspace, tabela, klucz partycji, klucz klastrujący. Używamy ich ostrożnie: mechanizm utrzymywania widoków bywał źródłem poważnych błędów spójności we wcześniejszych wersjach Cassandry (dokumentacja projektu przez długi czas oznaczała tę funkcję jako eksperymentalną), a diagnostyka rozjazdu między tabelą źródłową a widokiem jest trudniejsza niż diagnostyka dwóch osobnych tabel zapisywanych jawnie przez aplikację. W środowiskach produkcyjnych ręczna denormalizacja po stronie aplikacji bywa

wybierana świadomie zamiast widoków, mimo dodatkowej pracy przy każdym zapisie.

ZMIENIA SYSTEM

Wykonanie wielu operacji naraz: BATCH

```
BEGIN BATCH
    INSERT INTO zamowienia_wg_klienta (klient_id, data_zamowienia, id_zamowienia,
status) VALUES (klient_id, toTimestamp(now()), uuid(), 'nowe');
    INSERT INTO zamowienia_wg_statusu (status, klient_id, data_zamowienia,
id_zamowienia) VALUES ('nowe', klient_id, toTimestamp(now()), uuid());
APPLY BATCH;
```

Pojęcie: BATCH w Cassandrze. Grupa operacji zapisu wysyłana do klastra jednym poleceniem — celowo nie transakcja w sensie znanym z bazy relacyjnej i opisanym na stronie Czym jest transakcja: ACID rozłożone na cztery obietnice. `BATCH` nie daje izolacji ani atomowości widocznej dla równoległego odczytu — inny klient czytający w trakcie wykonywania batcha może zobaczyć część operacji już zapisaną, a część jeszcze nie. Jedyna gwarancja, jaką daje, to że jeśli batch się nie powiedzie w całości od razu, Cassandra dopisze wszystkie jego operacje do wewnętrznego logu batchowego i doprowadzi je do końca później — więc `BATCH` gwarantuje w końcu wykonanie wszystkiego, nie wykonanie wszystkiego naraz i widoczne dla innych jako jedna spójna operacja. Zastosowanie, dla którego `BATCH` faktycznie ma sens, to dokładnie ten przykład wyżej: zapis tego samego zdarzenia biznesowego do kilku zdenormalizowanych tabel na raz, żeby nie zostawić ich w rozjeździe przy awarii w połowie operacji — nie zastępowanie transakcji relacyjnej.

ACID = Atomicity, Consistency, Isolation, Durability: cztery obietnice transakcji w bazie danych

ZMIENIA SYSTEM

Warunkowe utworzenie wiersza: INSERT IF NOT EXISTS

```
INSERT INTO NazwaTabeli (klient_id, data_zamowienia, id_zamowienia, status) VALUES
(klient_id, data_zamowienia, id_zamowienia, 'nowe') IF NOT EXISTS;
```

> **Pojęcie: lekka transakcja (lightweight transaction, LWT).** Mechanizm oparty o protokół Paxos, który przed wykonaniem zapisu sprawdza warunek na aktualnym stanie wiersza i wykonuje zapis tylko wtedy, gdy warunek jest spełniony — całość jako operacja atomowa mimo rozproszenia danych po wielu replikach. `IF NOT EXISTS` wykonuje wstawienie tylko wtedy, gdy wiersz o tym kluczu głównym jeszcze nie istnieje.

Przykład wyniku, gdy warunek nie jest spełniony:

```
 [applied] | klient_id                              | data_zamowienia
-----------+----------------------------------------+---------------------------------
     False | 3c1f6e2a-1111-4b3a-9c2d-000000000001 | 2026-03-14 09:12:00.000000+0000
```

Kolumna `[applied]` mówi, czy operacja faktycznie się wykonała — `False` zwraca dodatkowo aktualny stan wiersza, który spowodował odrzucenie zapisu.

◫◫ ZMIENIA SYSTEM

Warunkowa aktualizacja: UPDATE IF

```
UPDATE NazwaTabeli SET status = 'anulowane' WHERE klient_id = klient_id AND
data_zamowienia = data_zamowienia AND id_zamowienia = id_zamowienia IF status = 'nowe';
```

Wykonujemy aktualizację tylko wtedy, gdy bieżąca wartość kolumny spełnia podany warunek — klasyczne zastosowanie to zmiana statusu zamówienia tylko z jednego konkretnego stanu na drugi, żeby dwa równoległe procesy nie nadpisały sobie nawzajem statusu w niewłaściwej kolejności.

Koszt lekkich transakcji. Protokół Paxos wymaga czterech rund komunikacji między replikami zamiast jednej, jak przy zwykłym zapisie — LWT są więc rzędu wielkości wolniejsze niż zwykły `INSERT` czy `UPDATE` i nie powinny trafiać na gorącą ścieżkę zapisu obsługującą duży wolumen; rezerwujemy je dla operacji, gdzie poprawność warunku faktycznie musi być gwarantowana atomowo, nie dla każdego zapisu z osobna.

◫◫ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

Z życia wzięte

□□ Z ŻYCIA WZIĘTE

Zespół Oli chce podnieść replication factor telemetrii z dwóch do trzech kopii, żeby awaria jednego z węzłów cass01 do cass03 nigdy nie kosztowała utraty dostępu do świeżych danych z magazynu. ALTER KEYSPACE zmienia definicję w mgnieniu oka, ale nowa, trzecia replika na razie stoi pusta, bo sama zmiana definicji nie kopiuje żadnych danych. Dopiero nodetool repair uruchomiony na każdym węźle po kolei faktycznie dociąga brakujące kopie, zanim ktokolwiek zdąży polegać na tej trzeciej replice podczas awarii.

Częste błędy

Pisanie `WHERE` tak jak w SQL, bez uwzględnienia klucza partycji. Warunek na dowolnej kolumnie działa składniowo tak samo jak w bazie relacyjnej, dopóki silnik nie odrzuci go, żądając `ALLOW FILTERING` — a to zwykle sygnał, że tabela nie została zaprojektowana pod to zapytanie, nie że brakuje jednej klauzuli.

Traktowanie `BATCH` jako narzędzia do przyspieszenia wielu niezależnych zapisów. `BATCH` obejmujący zapisy do wielu niepowiązanych partycji nie jest szybszy niż wysłanie tych samych zapisów osobno — jest wolniejszy, bo coordinator musi rozbić go i tak na osobne operacje per partycja, dodatkowo zapisując cały batch do logu batchowego na wypadek awarii w trakcie.

Nadużywanie lekkich transakcji. `IF NOT EXISTS` i `IF` używane przy każdym zapisie „na wszelki wypadek" obciążają klaster kosztem Paxos bez realnej potrzeby — sięgamy po nie tylko tam, gdzie zwykły zapis rzeczywiście dopuszczałby błędny stan (dwa równoległe żądania próbujące zarezerwować tę samą pozycję).

□□ Ćwiczenia

Poniższe zadania sprawdzają dobór właściwej konstrukcji CQL do sytuacji, nie wymagają uruchomionego klastra.

Pytania kontrolne

1. Aplikacja musi wykonać zapis zdarzenia do dwóch zdenormalizowanych tabel naraz i nie może zostawić ich w rozjeździe przy awarii w połowie operacji. Czy to zastosowanie dla `BATCH`, dla lekkiej transakcji, czy dla żadnego z nich?

Odpowiedź

Dla `BATCH` — to dokładnie przypadek, do którego mechanizm został zaprojektowany: dwie operacje zapisu tego samego zdarzenia, które mają w końcu obie się wykonać albo żadna. Lekka transakcja rozwiązuje inny problem — warunkowe wykonanie pojedynczego zapisu w zależności od aktualnego stanu danych, nie grupowanie kilku niezależnych zapisów.

2. Zapytanie `SELECT * FROM zamowienia WHERE klient_id = ... AND status = 'nowe'` wymaga `ALLOW FILTERING`, mimo że `klient_id` jest kluczem partycji. Co najprawdopodobniej jest przyczyną?

Odpowiedź

`status` nie jest kolumną klastrującą tej tabeli ani nie jest zindeksowany — sam warunek na `klient_id` byłby wystarczający i nie wymagałby `ALLOW FILTERING`, ale dodatkowy warunek na zwykłej kolumnie w tym samym zapytaniu uruchamia ten sam mechanizm ochronny, bo silnik musiałby przejrzeć wszystkie wiersze partycji i odfiltrować niepasujące po ich odczytaniu.

Powiązane: cqlsh: klient wiersza poleceń · Model danych: keyspace, tabela, klucz partycji, klucz klastrujący · Architektura Cassandry: pierścień, węzeł, replikacja · Czym jest transakcja: ACID rozłożone na cztery obietnice · Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli"

Źródła: dokumentacja Apache Cassandra, rozdz. „CQL" · notatki własne z eksploatacji klastrów produkcyjnych

---

Revision #43
Created 2026-09-24 23:34:52 UTC by Tomasz Konieczny
Updated 2026-10-26 14:13:09 UTC by Tomasz Konieczny