Model danych: keyspace, tabela, klucz partycji, klucz klastrujący

Poziom: podstawy **Dotyczy: Apache Cassandra 3.x / 4.x** **Czas: ~25 min**

W Cassandrze klucz główny nie tylko identyfikuje wiersz — decyduje, na który węzeł ten wiersz trafi i w jakiej kolejności zostanie zapisany obok innych wierszy tej samej partycji.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
CQL	Cassandra Query Language: język zapytań Cassandry, podobny do SQL
RF	Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry
UDT	Urząd Dozoru Technicznego: polska instytucja, która według informacji od autora książki wydaje certyfikaty osobom i firmom pracującym przy czynnikach F-gazowych

Z życia wzięte

Z ŻYCIA WZIĘTE

Ola projektuje tabelę telemetrii tak, żeby dało się szybko sprawdzić, gdzie w danej chwili jest konkretny wózek, i osobno, jakie zdarzenia miał w ostatniej godzinie dany skaner. Zamiast jednej uniwersalnej tabeli i JOIN-a, jak zrobiłaby to w PostgreSQL, projektuje dwie osobne tabele pod dwa różne zapytania, każda z innym kluczem partycji. Klucz partycji po identyfikatorze wózka trzyma wszystkie jego zdarzenia fizycznie razem na tych samych węzłach, a kolumny klastrujące po czasie sortują je od razu w kolejności, w jakiej Marek chce je zobaczyć na dashboardzie.

Keyspace i tabela

Pojęcie: keyspace. Najwyższy poziom organizacji danych w Cassandrze, odpowiednik bazy danych w silniku relacyjnym — z jedną istotną różnicą: to na poziomie keyspace, nie tabeli, deklarujemy replication factor i strategię replikacji opisane na stronie Architektura Cassandry: pierścień, węzeł, replikacja. Wszystkie tabele w danym keyspace dziedziczą ten sam sposób replikacji, chyba że jawnie go nadpiszemy. Składnię `CREATE KEYSPACE` pokazujemy na stronie CQL: zapytania i operacje na danych.

Pojęcie: tabela (w CQL czasem nazywana column family). Zbiór wierszy o tym samym zdefiniowanym z góry kluczu głównym i zestawie kolumn — pod tym względem bliżej jej do tabeli relacyjnej niż do dokumentu, bo schemat jest zadeklarowany, nie dowolny. Różnica zaczyna się dopiero w tym, jak ten klucz główny działa.

Klucz główny: partycja i klastrowanie

Pojęcie: klucz partycji (partition key). Część klucza głównego, która wchodzi do funkcji haszującej partycjonera opisanej na stronie Architektura Cassandry: pierścień, węzeł, replikacja i wyznacza, na który węzeł (a właściwie na który zakres tokenów, czyli na którą trójkę węzłów przy RF = 3) trafi cały zestaw wierszy o tej wartości klucza partycji. Wszystkie wiersze o tym samym kluczu partycji leżą fizycznie razem, na tych samych węzłach, w jednym ciągłym fragmencie na dysku — to jest partycja w sensie dosłownym, nie tylko logicznym.

Pojęcie: kolumny klastrujące (clustering columns). Pozostała część klucza głównego, która nie wpływa na to, gdzie wiersz leży, tylko na kolejność wierszy wewnątrz jednej partycji. Wiersze tej samej partycji są fizycznie posortowane na dysku według kolumn klastrujących, więc odczyt zakresu po kolumnie klastrującej (na przykład „zdarzenia tego urządzenia z ostatniej godziny") jest sekwencyjnym odczytem z dysku, a nie przeszukiwaniem rozrzuconych wierszy.

Deklarujemy to rozróżnienie wprost w `PRIMARY KEY`:

```
PRIMARY KEY ((klient_id), data_zamowienia, id_zamowienia)
```

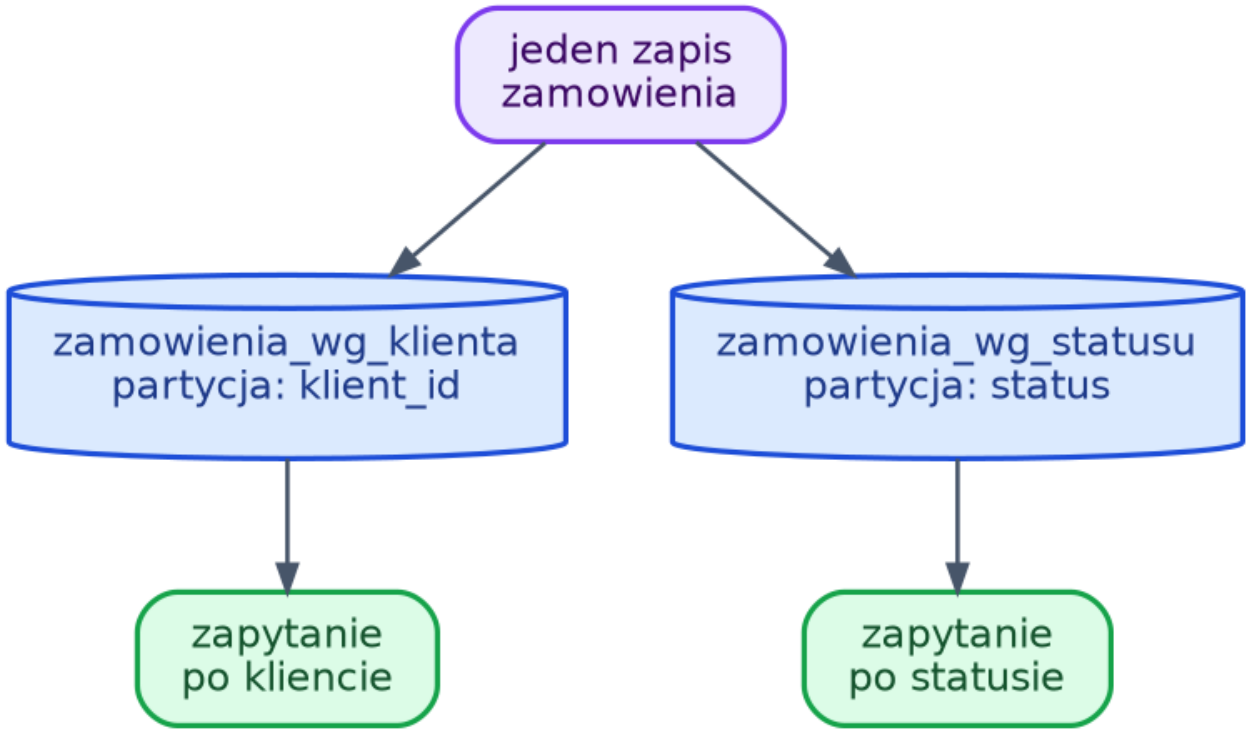
Nawias wewnętrzny obejmuje klucz partycji — tu jednoelementowy, `klient_id` — a `data_zamowienia` i `id_zamowienia` to kolumny klastrujące, według których wiersze tej samej partycji są posortowane, w tej kolejności. Pełną składnię `CREATE TABLE` z kluczem złożonym i `CLUSTERING ORDER BY` pokazujemy na stronie CQL: zapytania i operacje na danych.

Klucz partycji może się też składać z wielu kolumn — `PRIMARY KEY ((klient_id, rok), data_zamowienia)` — wtedy do jednej partycji trafiają tylko wiersze o identycznej parze `(klient_id, rok)`, co bywa świadomym zabiegiem, żeby partycja nie rosła bez końca (partycja obejmująca wszystkie zamówienia klienta od zawsze robi się z czasem nieporęcznie duża).

Dlaczego projektujemy tabelę pod zapytanie, a nie pod encję

W modelu relacyjnym zaczynamy od encji — klient, zamówienie, produkt — i dopiero potem piszemy zapytania, które łączą je w locie przez `JOIN`. W Cassandrze ta kolejność jest odwrócona: zanim napiszemy `CREATE TABLE`, musimy wiedzieć, jakim dokładnie zapytaniem będziemy tę tabelę odpytywać, bo klucz partycji i kolumny klastrujące trzeba dobrać tak, żeby to zapytanie w ogóle dało się wykonać wydajnie.

Konsekwencją jest to, że ta sama encja bywa zapisana w kilku tabelach naraz, każda pod inny wzorzec dostępu. Zamówienia potrzebujemy odczytywać zarówno „po kliencie", jak i „po statusie realizacji" — w Cassandrze to dwie osobne tabele, `zamowienia_wg_klienta` (partycja: `klient_id`) i `zamowienia_wg_statusu` (partycja: `status`), każda z tym samym zestawem kolumn, zapisywane oba naraz przy każdym zapisie zamówienia. Duplikacja danych, która w modelu relacyjnym byłaby błędem projektowym psującym integralność, w Cassandrze jest normą — bo tu nie ma mechanizmu, który połączyłby dwie tabele w locie równie tanio, jak robi to `JOIN` w silniku relacyjnym.



Jeden zapis zamówienia trafia od razu do dwóch tabel, każdej zaprojektowanej pod inny wzorzec odczytu (kliknięcie otwiera pełny rozmiar)

Ograniczenia modelu

Brak JOIN. Cassandra nie ma odpowiednika złączenia opisanego na stronie Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza — nie dlatego, że o nim zapomniano, tylko dlatego, że złączenie w locie danych leżących na różnych węzłach byłoby operacją kosztowną i nieprzewidywalną w czasie, a Cassandra celowo nie udostępnia operacji, których koszt rośnie z rozmiarem danych w sposób, którego administrator nie kontroluje. Złączenie, którego potrzebujemy, wykonujemy raz, przy zapisie, zapisując zdenormalizowane dane do tabeli zaprojektowanej pod to konkretne zapytanie.

Brak agregatów po wielu partycjach. Funkcje takie jak `COUNT`, `SUM` czy `AVG` działają w Cassandrze wyłącznie w obrębie pojedynczego zapytania obejmującego ograniczony zakres danych — nie istnieje odpowiednik `GROUP BY klient_id` liczącego sumę zamówień każdego klienta w całej tabeli naraz, bo to wymagałoby przejrzenia wszystkich partycji na wszystkich węzłach. `COUNT(*)` na całej tabeli technicznie się wykona, ale skanuje cały klaster i w praktyce jest operacją diagnostyczną, nie czymś wywoływanym z aplikacji na gorącej ścieżce.

> **Pojęcie: `ALLOW FILTERING` jako sygnał złego modelu.** Klauzula, którą Cassandra wymusza, gdy zapytanie filtruje po kolumnie, która nie jest częścią klucza partycji ani nie jest indeksowana — bez niej taki `SELECT` zostaje odrzucony, zanim w ogóle spróbuje coś odczytać. `ALLOW FILTERING` nie przyspiesza zapytania — pozwala je wykonać, każąc silnikowi przejrzeć potencjalnie wszystkie partycje i odrzucić te, które nie pasują do warunku, dopiero po ich odczytaniu. Potrzeba użycia tej klauzuli w kodzie aplikacji, a nie w zapytaniu ad hoc do diagnostyki, jest sygnałem, że tabela została zaprojektowana pod inne zapytanie niż to, które faktycznie wykonujemy — rozwiązaniem jest zwykle dodatkowa tabela pod ten wzorzec dostępu, nie dopisanie `ALLOW FILTERING` do zapytania produkcyjnego. Samą składnię i ostrzeżenie przy jej użyciu pokazujemy na stronie CQL: zapytania i operacje na danych.

Typy danych CQL

Typ	Znaczenie
`text` / `varchar`	ciąg znaków UTF-8 o dowolnej długości
`int`, `bigint`, `smallint`, `tinyint`	liczby całkowite różnej szerokości
`float`, `double`, `decimal`	liczby zmiennoprzecinkowe i stałoprzecinkowe
`boolean`	wartość logiczna
`uuid`	uniwersalny identyfikator, zwykle klucz partycji generowany po stronie aplikacji
`timeuuid`	UUID zawierający znacznik czasu — sortuje się chronologicznie, przydatny jako kolumna klastrująca
`timestamp`	data i czas z dokładnością do milisekund
`date`, `time`	data albo czas bez drugiej składowej
`blob`	dane binarne w postaci surowej
`inet`	adres IP, w wersji 4 albo 6
`counter`	licznik — omówiony osobno niżej

Kolekcje: set, list, map

> **Pojęcie: kolekcja w CQL.** Kolumna, która zamiast pojedynczej wartości trzyma zbiór wartości tego samego typu — Cassandra udostępnia trzy rodzaje, każdy z inną semantyką porządku i unikalności.

`set<text>` przechowuje zbiór unikalnych wartości bez zdefiniowanej kolejności — pasuje do tagów albo uprawnień, gdzie liczy się tylko przynależność, nie kolejność ani powtórzenia.

`list<text>` przechowuje wartości w kolejności wstawienia, z dopuszczonymi powtórzeniami — kosztem tego, że modyfikacja elementu w środku listy jest w Cassandrze operacją droższą niż dopisanie na końcu.

`map<text, int>` przechowuje pary klucz-wartość w obrębie jednej kolumny jednego wiersza — wygodne dla danych o zmiennej, niewielkiej liczbie atrybutów, których nie opłaca się rozbijać na osobne kolumny.

Kolekcje mają sens dla niewielkiej liczby elementów (dokumentacja silnika odradza więcej niż kilkaset) — cała kolekcja jest odczytywana i zapisywana w całości przy każdym dostępie do wiersza, więc duża kolekcja robi z pojedynczego odczytu operację kosztowną.

Typy zdefiniowane przez użytkownika (UDT)

> **Pojęcie: UDT (user-defined type).** Własny typ złożony z nazwanych pól, definiowany raz i używany jako typ kolumny w wielu tabelach — odpowiednik struktury (`struct`) znanej z języków programowania, osadzonej wewnątrz wiersza Cassandry. Adres klienta złożony z ulicy, miasta i kodu pocztowego, użyty w tabeli `klienci` i w `zamowienia_wg_klienta` naraz bez przepisywania tych samych trzech kolumn dwa razy, jest typowym zastosowaniem UDT — porządkuje duplikację danych, która i tak jest normą w tym modelu, zamiast jej zapobiegać.

Counter

> **Pojęcie: typ counter.** Specjalny typ kolumny do atomowego zwiększania i zmniejszania wartości liczbowej bez odczytu bieżącego stanu przed zapisem — `UPDATE licznik SET wartosc = wartosc + 1 WHERE ...` jest operacją bezpieczną przy wielu równoległych zapisach z różnych węzłów naraz, bo Cassandra sama scala przyrosty, zamiast wymagać od aplikacji odczytania starej wartości i obliczenia nowej. Ograniczenie: tabela z kolumną typu `counter` może zawierać wyłącznie kolumny klucza głównego i inne kolumny typu `counter` — nie da się w tym samym wierszu trzymać licznika obok zwykłej kolumny tekstowej.

Tombstone: czym jest usuwanie w bazie rozproszonej

> **Pojęcie: tombstone (nagrobek).** Cassandra nigdy nie usuwa danych z dysku w miejscu — `DELETE` zapisuje w tym samym miejscu, w którym leżałaby zwykła wartość, specjalny znacznik mówiący „ta wartość została usunięta o tym czasie". Powód jest ten sam, co przy każdym zapisie w tym modelu: dane leżą w wielu kopiach na różnych węzłach, a węzeł, który był offline w chwili `DELETE`, musi po powrocie dowiedzieć się, że dana wartość zniknęła, a nie tylko że przestała być aktualizowana — sam brak wiersza w odpowiedzi jednej repliki nie odróżnia „nigdy nie istniał" od "został usunięty".

Usuwanie jest w tym modelu operacyjnie droższe niż zapis z tego samego powodu, dla którego jest w ogóle możliwe: tombstone zajmuje miejsce, tak samo jak zwykła wartość, i musi zostać odczytany razem z resztą wiersza przy każdym kolejnym `SELECT` na tej partycji, dopóki nie zostanie fizycznie usunięty przez kompaktowanie opisane na stronie Utrzymanie klastra: repair, compaction, czyszczenie. Partycja z dużą liczbą tombstone'ów (typowy przypadek: kolejka, z której elementy są usuwane zaraz po przetworzeniu) potrafi spowolnić odczyt na tyle, że Cassandra ostrzega o tym w logu — sygnał opisany w tabeli objawów na stronie Konfiguracja i diagnostyka.

> **Pojęcie: gc_grace_seconds.** Parametr tabeli określający, jak długo tombstone musi przeleżeć na dysku, zanim kompaktowanie pozwoli go trwale usunąć. Domyślna wartość to 10 dni (864000 sekund) — czas dobrany tak, żeby węzeł, który był offline dłużej niż to okno, zdążył zostać naprawiony komendą `nodetool repair` zanim tombstone zniknie; w przeciwnym razie węzeł wracający po dłuższej nieobecności mógłby nie zobaczyć informacji o usunięciu i „wskrzesić" dawno skasowany wiersz przy najbliższym read repair. Związek między `gc_grace_seconds` a harmonogramem napraw rozwija strona Utrzymanie klastra: repair, compaction, czyszczenie.

Częste błędy

Projektowanie tabeli od encji, nie od zapytania. Klasyczny odruch osoby przychodzącej z modelu relacyjnego — stworzenie jednej tabeli `zamowienia` z kluczem `id_zamowienia` i próba filtrowania po `klient_id` w `WHERE`. Bez `klient_id` jako klucza partycji takie zapytanie wymaga `ALLOW FILTERING` i skanuje cały klaster.

Nieograniczony wzrost pojedynczej partycji. Klucz partycji zbyt ogólny (na przykład sam `rok` dla tabeli zdarzeń zapisywanych tysiące razy na sekundę) prowadzi do partycji, które rosną bez końca i w końcu stają się „gorącymi partycjami" — całe obciążenie zapisu i odczytu tej partycji spada na jedną trójkę węzłów (przy RF = 3), zamiast rozkładać się po całym klastrze.

Traktowanie kolekcji jak małej, prywatnej tabeli. Kolekcja z tysiącami elementów, aktualizowana często przez wielu klientów naraz, generuje dużo tombstone'ów przy każdej modyfikacji (usunięcie elementu z `set` czy `map` to osobny tombstone na element) — objaw dokładnie taki sam, jak przy partycji z dużą liczbą usuniętych wierszy.

□□ Ćwiczenia

> Pytania sprawdzają rozumienie modelu, nie wymagają uruchomionego klastra.

Pytania kontrolne

1. Tabela ma `PRIMARY KEY ((sensor_id), zmierzono_o)`. Jakie zapytanie wykona się szybko, a jakie będzie wymagało `ALLOW FILTERING`?

Odpowiedź

`SELECT * FROM odczyty WHERE sensor_id = ... AND zmierzono_o > ...` wykona się szybko — filtruje po kluczu partycji i zakresie na kolumnie klastrującej, czyli dokładnie zgodnie z tym, jak dane są fizycznie ułożone na dysku. `SELECT * FROM odczyty WHERE zmierzono_o > ...` bez `sensor_id` będzie wymagało `ALLOW FILTERING`, bo `zmierzono_o` samo w sobie nie wyznacza żadnej konkretnej partycji — trzeba by przejrzeć wszystkie.

2. Dlaczego dwie tabele z tymi samymi kolumnami, różniące się wyłącznie kluczem partycji, są w Cassandrze normalnym rozwiązaniem, a nie oznaką złego projektu?

Odpowiedź

Bo Cassandra nie ma taniego mechanizmu złączenia danych leżących w dwóch różnych miejscach klastra w czasie odczytu — jeśli te same dane trzeba odpytywać dwoma różnymi kluczami, jedyny sposób, żeby oba zapytania były szybkie, to zapisać dane dwa razy, raz pod każdy klucz partycji, w chwili zapisu, gdy koszt tej operacji jest znany i kontrolowany.

3. Tabela `zdarzenia` z kolumną typu `counter` ma dodatkowo zawierać nazwę zdarzenia jako zwykłą kolumnę tekstową. Dlaczego to się nie uda i jak to obejść?

Odpowiedź

Tabela z kolumną typu `counter` może zawierać wyłącznie kolumny klucza głównego i inne kolumny typu `counter` — nie da się w niej trzymać zwykłej kolumny tekstowej obok licznika. Obejściem jest osobna tabela na metadane zdarzenia (nazwa, opis) i osobna tabela wyłącznie z licznikiem, powiązane tym samym kluczem partycji.

Powiązane: Architektura Cassandry: pierścień, węzeł, replikacja · CQL: zapytania i operacje na danych · Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza · Utrzymanie klastra: repair, compaction, czyszczenie

Źródła: dokumentacja Apache Cassandra, rozdz. „Data Modeling" i „CQL"

Revision #43
Created 2026-09-24 23:34:49 UTC by Tomasz Konieczny
Updated 2026-10-26 14:13:05 UTC by Tomasz Konieczny