Architektura i model danych

Pierścień, replikacja, spójność oraz projektowanie tabel pod zapytanie.

- Cassandra: baza z Facebooka, która nie ma szefa
- Architektura Cassandry: pierścień, węzeł, replikacja
- Model danych: keyspace, tabela, klucz partycji, klucz klastrujący

Cassandra: baza z Facebooka, która nie ma szefa

Poziom: podstawy **Dotyczy: Apache Cassandra 4.1** **Czas: ~10 min**

Ta strona otwiera książkę „Cassandra": skąd wziął się silnik bez węzła głównego, jak to wygląda w jednym obrazku i dlaczego w Zielonej trzy takie węzły zbierają dane z magazynowych skanerów i wózków.

Skąd się wzięło

> **Pojęcie: baza kolumnowa (szeroko-kolumnowa).** Model, w którym dane grupowane są wierszami, ale w przeciwieństwie do tabeli relacyjnej każdy wiersz może mieć inny zestaw kolumn, a dane fizycznie trzymane są kolumna po kolumnie, nie wiersz po wierszu.

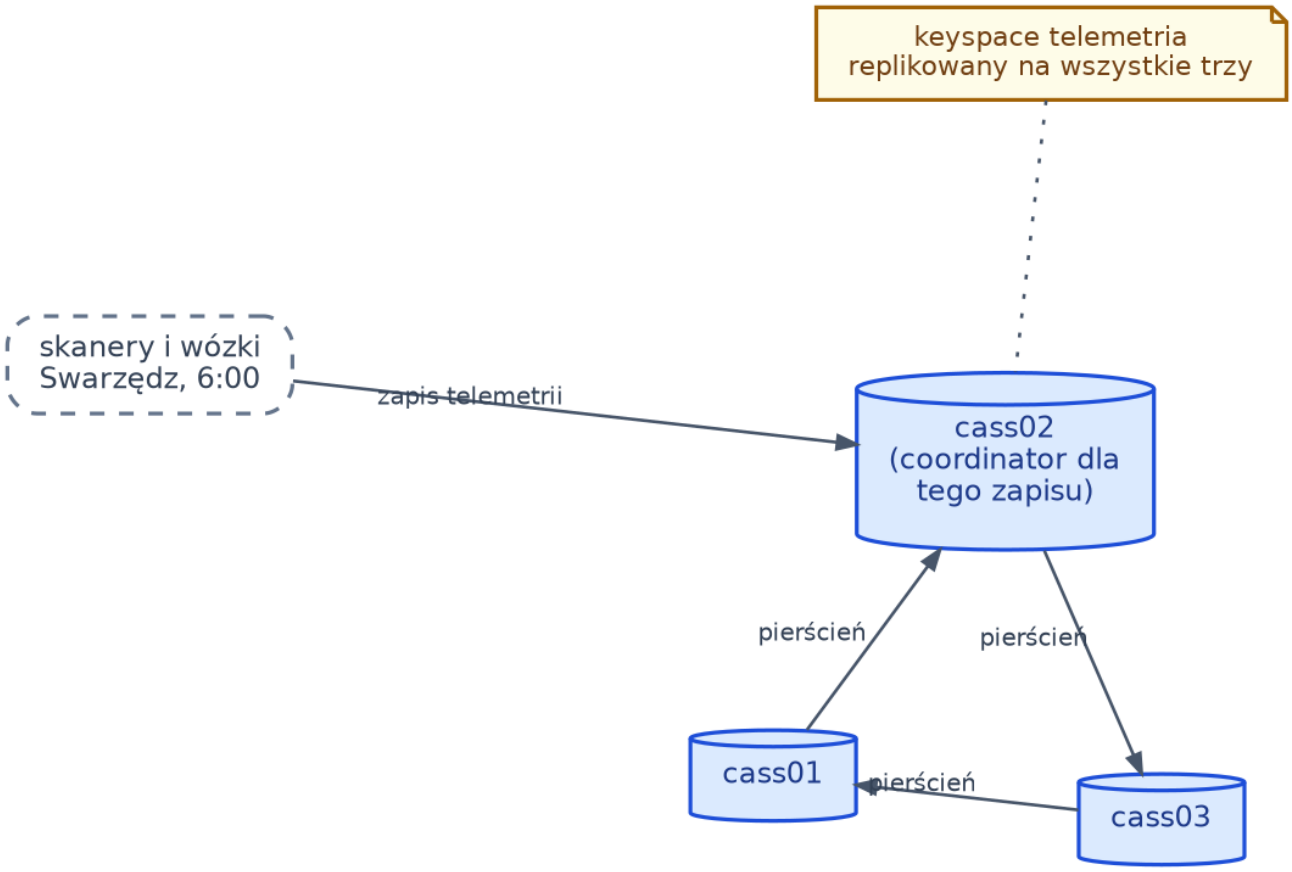
Cassandrę zbudowali w 2008 roku Avinash Lakshman i Prashant Malik, inżynierowie z zespołu wyszukiwania w Facebooku, żeby rozwiązać bardzo konkretny problem: wyszukiwanie w skrzynkach odbiorczych użytkowników wymagało przechowywania odwróconych indeksów wiadomości na setkach zwykłych serwerów naraz, z gwarancją, że awaria pojedynczej maszyny nie zablokuje wyszukiwania nikomu. Lakshman był wcześniej współautorem systemu Dynamo w Amazonie, więc do Cassandry przeniósł jego pomysł na rozproszenie danych bez jednego centralnego punktu. Drugim źródłem inspiracji był opisany przez Google w 2006 roku system Bigtable, od którego Cassandra przejęła sposób organizowania danych w szerokie kolumny.

Facebook opublikował Cassandrę jako otwarte oprogramowanie w sierpniu 2008 roku, oddając społeczności kod, który wcześniej działał tylko wewnątrz jednej firmy. W marcu 2009 roku projekt trafił pod skrzydła Apache Software Foundation jako projekt inkubowany, a 17 lutego 2010 roku awansował do statusu w pełni samodzielnego projektu najwyższego poziomu Apache. Nazwa nawiązuje do postaci z mitologii greckiej, kapłanki Troi, która widziała nadchodzącą katastrofę, ale nikt jej nie wierzył, co pasowało do przekonania twórców, że rozproszone systemy bez pojedynczego punktu awarii to nieunikniona przyszłość dla danych na skalę internetu.

Od tamtej pory Cassandra stała się jednym z podstawowych wyborów dla firm, które muszą zapisywać ogromne ilości danych z bardzo dużą częstotliwością, częściej niż odpytywać je w skomplikowany sposób: telemetria z urządzeń, logi zdarzeń, historia aktywności użytkowników.

Jak to działa w jednym obrazku

Zapis trafia do dowolnego węzła klastra, bo żaden z nich nie jest ważniejszy od pozostałych, a ten węzeł, jako koordynator tego konkretnego zapytania, sam rozsyła dane do węzłów odpowiedzialnych za dany wiersz. Ile kopii tego wiersza powstanie, decyduje replication factor ustawiony dla danego keyspace.



Skanery magazynowe zapisują telemetrię na dowolny węzeł pierścienia, dane trafiają replikowane na wszystkie trzy (kliknięcie otwiera pełny rozmiar)

Dlaczego Zielona

Z ŻYCIA WZIĘTE

O szóstej rano w Swarzędzu czterdzieści skanerów i kilkanaście wózków widłowych loguje się naraz do systemu magazynowego, a każdy z nich raportuje swoją pozycję i stan co kilka sekund. Marek, kierownik magazynu, kiedyś próbował trzymać te dane w zwykłej bazie relacyjnej, ale przy starcie zmiany baza dławiła się falą jednoczesnych zapisów. Po przejściu na klaster Cassandry `cass01` do `cass03` ten sam poranek przechodzi bez zauważalnego spowolnienia.

Telemetria skanerów i wózków trafia dziś do keyspace `telemetria`, rozłożonego na trzech węzłach `cass01`, `cass02` i `cass03`, dokładnie dlatego, że Cassandra została zaprojektowana pod ten konkretny rodzaj obciążenia: bardzo dużo małych zapisów naraz, z prostym pytaniem o odczyt („pokaż ostatnie odczyty tego skanera"), a nie pod złożone raporty łączące wiele tabel. Żaden z trzech węzłów nie jest ważniejszy od pozostałych, więc awaria jednego z nich w środku zmiany porannej nie zatrzymuje zapisu reszty floty. To dokładnie ten typ decyzji architektonicznej, którą łatwiej zrozumieć, znając historię: Cassandra powstała po to, żeby żaden pojedynczy serwer nie mógł zablokować całego systemu, a magazyn w Swarzędzu o szóstej rano jest właśnie taką chwilą, w której to się liczy najbardziej.

Mapa tej książki

Książka „Cassandra" składa się z siedmiu rozdziałów i zakłada znajomość słownika z książki „Bazy danych: fundamenty".

Rozdział	Na jakie pytanie odpowiada
Cassandra: baza z Facebooka, która nie ma szefa	Skąd wzięła się Cassandra i dlaczego nie ma węzła głównego
Architektura Cassandry	Co to jest pierścień, węzeł, replikacja i poziom spójności
Model danych	Co to jest keyspace, tabela, klucz partycji i klucz klastrujący
cqlsh	Jak połączyć się z klastrem z linii poleceń
CQL	Jak pisać zapytania i operacje na danych
nodetool	Jak sprawdzić stan klastra i wykonać operacje administracyjne
Utrzymanie klastra	Czym są repair, compaction i czyszczenie
Backup i odtwarzanie	Jak zrobić i odtworzyć kopię zapasową
Konfiguracja i diagnostyka	Gdzie szukać przyczyny problemu
Role, uprawnienia i monitoring JMX	Jak nadać dostęp i podłączyć monitoring
Zmiana nazwy klastra i datacenter	Jak bezpiecznie zmienić nazwę bez przestoju

Powiązane: Architektura Cassandry · Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin

Źródła: Meta Engineering, „Cassandra: A structured storage system on a P2P network" (engineering.fb.com, sierpień 2008) · Wikipedia, „Apache Cassandra" · Stacksync, „The Prophetess Nobody Believed: The Origin Story of Apache Cassandra"

Architektura Cassandry: pierścień, węzeł, replikacja

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

Cassandra nie ma węzła głównego — każdy węzeł w klastrze jest sobie równy, a to, jak dane trafiają na dysk i jak wracają do klienta, wynika z jednej struktury: pierścienia tokenów.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
CAP	Client Access Point: punkt dostępu klienta, czyli nazwa i adres IP roli klastra, pod którymi użytkownicy łączą się z usługą niezależnie od tego, który węzeł ją prowadzi
DC	Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników
RF	Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry
WMS	Warehouse Management System: system zarządzania magazynem

Po co to istnieje

> **Pojęcie: węzeł.** Pojedynczy serwer biorący udział w klastrze bazy danych, przechowujący fragment danych i odpowiadający na zapytania.

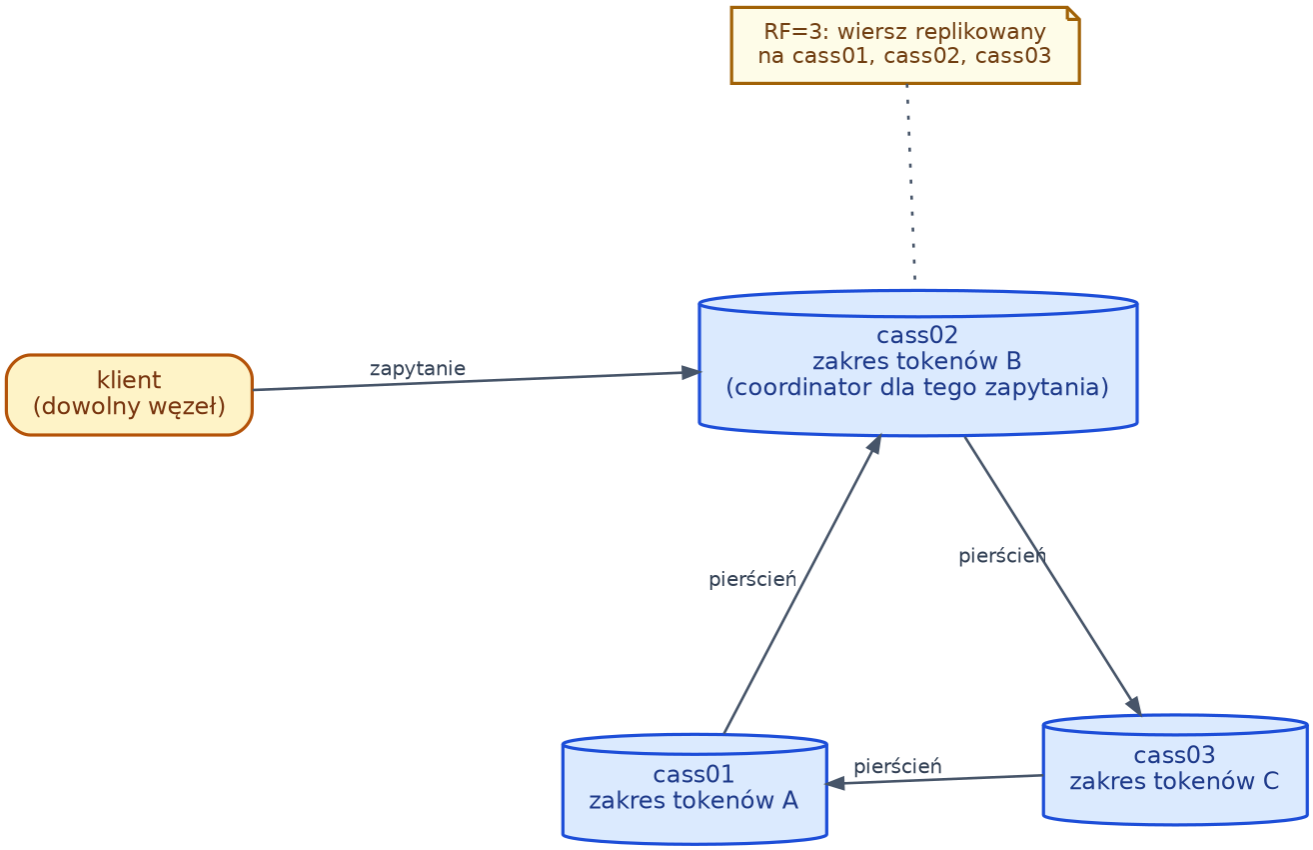
> **Pojęcie: replika.** Kolejna kopia tego samego wiersza danych, trzymana na innym węźle, żeby awaria jednego z nich nie oznaczała utraty dostępu do danych.

Model relacyjny, opisany na stronach fundamentów, zakłada zwykle jeden serwer albo klaster z jasno wydzieloną rolą — węzeł podstawowy przyjmuje zapisy, repliki tylko czytają. Cassandra powstała z przeciwnego założenia: żaden pojedynczy węzeł nie może być punktem awarii ani wąskim gardłem, więc architektura eliminuje pojęcie węzła głównego całkowicie. Każdy węzeł potrafi przyjąć zapis, każdy potrafi odpowiedzieć na odczyt, a decyzję, który węzeł faktycznie przechowuje dany wiersz, podejmuje deterministyczna funkcja skrótu, nie konfiguracja administratora.

Konsekwencją tego wyboru jest to, że pytania, które w bazie relacyjnej są oczywiste — „gdzie jest ten wiersz?", „czy odczytam najświeższą wersję?" — w Cassandrze wymagają zrozumienia mechaniki pierścienia, replikacji i poziomów spójności. To, czym różni się model kolumnowy od relacyjnego na poziomie koncepcyjnym, opisuje strona Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin — tutaj zajmujemy się wyłącznie tym, jak Cassandra jest zbudowana w środku.

Pierścień i zakresy tokenów

> **Pojęcie: pierścień.** Logiczna struktura, w której wszystkie węzły klastra są ułożone w okrąg według przypisanych im wartości liczbowych — tokenów. Przestrzeń możliwych tokenów jest skończona i cykliczna (od wartości minimalnej do maksymalnej, a potem z powrotem do minimalnej), więc każdy węzeł odpowiada za pewien wycinek tego okręgu — zakres tokenów między swoim tokenem a tokenem poprzedniego węzła w kierunku wskazówek zegara.



Trzy wezly ulozone w pierscien, kazdy z wlasnym zakresem tokenow, i wiersz replikowany na wszystkie trzy (kliknięcie otwiera pełny rozmiar)

Pojęcie: token. Liczba przypisana węzłowi (albo, przy vnodes, wielu liczb przypisanych temu samemu węzłowi), która wyznacza jego pozycję na pierścieniu. Każdy klucz partycji, zanim trafi do konkretnego węzła, przechodzi przez funkcję haszującą partycjonera (domyślnie `Murmur3Partitioner`), która zamienia go na liczbę z tej samej przestrzeni tokenów. Węzeł odpowiedzialny za dany wiersz to ten, którego zakres tokenów obejmuje wynik tego hasza — nie ten, którego administrator „przypisał" do danej tabeli. Sposób, w jaki klucz partycji trafia do funkcji haszującej, opisuje strona Model danych: keyspace, tabela, klucz partycji, klucz klastrujący.

Pojęcie: vnodes. Zamiast przypisać każdemu fizycznemu węzłowi jeden token (jeden ciągły zakres na pierścieniu), nowoczesna Cassandra przypisuje mu domyślnie 256 mniejszych, rozproszonych po całym pierścieniu tokenów wirtualnych. Efekt jest taki, że dane, za które odpowiadał węzeł, który właśnie odpadł z klastra, są rozłożone na wielu pozostałych węzłach zamiast skupione na jednym sąsiedzie — dołączenie albo usunięcie węzła rozkłada obciążenie równomiernie zamiast przenosić cały ciężar na najbliższego sąsiada w pierścieniu. Liczbę tokenów widzimy wprost w wyniku `nodetool info`, opisanym na stronie nodetool: stan klastra i operacje administracyjne.

Węzeł, rack, datacenter

Pojęcie: węzeł. Pojedynczy proces `cassandra` (jedna instancja bazy) uczestniczący w klastrze, przechowujący swój wycinek danych i biorący udział w gossipie opisanym niżej. W tej książce węzły nazywamy `cass01`, `cass02`, `cass03`.

Pojęcie: rack. Grupa węzłów deklarowana administracyjnie jako fizycznie bliska sobie — ten sam szereg szaf, ta sama strefa dostępności w chmurze. Cassandra unika umieszczania więcej niż jednej repliki tego samego wiersza w tym samym racku, o ile jest to możliwe przy zadanym `replication_factor` — dzięki temu awaria całej szafy nie zabiera jednocześnie dwóch kopii tych samych danych.

Pojęcie: datacenter (DC) w Cassandrze. Największa jednostka topologii — grupa racków, zwykle odpowiadająca fizycznej lokalizacji albo logicznemu podziałowi ruchu (na przykład osobny DC pod obciążenie analityczne, żeby ciężkie zapytania nie odbijały się na latencji ruchu produkcyjnego). Przynależność węzła do racku i DC deklarujemy w pliku `cassandra-rackdc.properties`

, opisanym na stronie Konfiguracja i diagnostyka.

Replication factor i strategia replikacji

Pojęcie: replication factor (RF). Liczba określająca, na ile węzłów będzie replikowany dany keyspace — czyli ile kopii każdego wiersza istnieje jednocześnie w klastrze. RF ustawiany jest per keyspace, nie globalnie dla całego klastra, więc różne keyspace w tym samym klastrze mogą mieć różny poziom replikacji.

To, na jakie konkretnie węzły dany wiersz trafi, decyduje już strategia replikacji, nie samo RF — RF mówi tylko „ile", strategia mówi „gdzie".

Pojęcie: SimpleStrategy. Najprostsza strategia replikacji — rozkłada kopie danych po kolejnych węzłach na pierścieniu, idąc zgodnie z ruchem wskazówek zegara od węzła wskazanego przez token, zupełnie ignorując podział na racki i datacenter. Dla klastra w jednym DC działa poprawnie, ale w klastrze wielo-DC może umieścić wszystkie kopie w tym samym centrum danych, robiąc replikację bezużyteczną w razie awarii całego DC.

Pojęcie: NetworkTopologyStrategy. Strategia świadoma topologii — RF deklarujemy osobno dla każdego datacenter, a strategia w obrębie każdego z nich stara się dodatkowo rozłożyć repliki po różnych rackach. To jedyna strategia sensowna w środowisku produkcyjnym i jedyna, która ma sens przy więcej niż jednym DC. Pełną składnię `CREATE KEYSPACE` i `ALTER KEYSPACE` dla obu strategii pokazujemy na stronie CQL: zapytania i operacje na danych.

Strategia	Świadoma racków i DC	Sensowna w produkcji wielo-DC
SimpleStrategy	nie	nie
NetworkTopologyStrategy	tak	tak

Spójność ostateczna i poziomy spójności

> **Pojęcie: spójność ostateczna (eventual consistency).** Gwarancja słabsza od spójności silnej znanej z transakcji relacyjnych — mówi, że jeśli zapis do repliki się powiedzie, to wszystkie repliki tego wiersza *w końcu* będą miały tę samą wartość, ale nie gwarantuje, że stanie się to natychmiast ani że każdy kolejny odczyt zobaczy już najnowszą wersję. W klastrze z RF = 3 zapis może zostać potwierdzony klientowi, zanim wszystkie trzy repliki faktycznie zapiszą dane na dysk — pozostałe dwie dogonią pierwszą z opóźnieniem rzędu milisekund, w normalnych warunkach sieciowych.

> **Pojęcie: poziom spójności (consistency level).** Parametr ustawiany per zapytanie (nie globalnie dla całej bazy), który mówi, ile replik musi potwierdzić operację, zanim Cassandra uzna ją za udaną i odpowie klientowi. To jest strojenie kompromisu między spójnością a dostępnością i latencją osobno dla każdego zapytania, nie decyzja podejmowana raz na cały klaster.

Przy RF = 3 (trzy kopie każdego wiersza) najczęściej spotykane poziomy zachowują się tak:

Poziom	Ile replik musi odpowiedzieć przy RF = 3	Skutek
`ONE`	1	najniższa latencja, najsłabsza gwarancja świeżości danych
`QUORUM`	2 (większość z 3)	kompromis — odporny na awarię jednej repliki
`LOCAL_QUORUM`	2, ale tylko replik w lokalnym DC	jak `QUORUM`, bez czekania na odpowiedź z odległego datacenter
`ALL`	3 (wszystkie)	najsilniejsza gwarancja, zero tolerancji na awarię pojedynczej repliki

> **Pojęcie: kworum.** Liczba obliczana jako `(RF / 2) + 1`, zaokrąglona w dół przy dzieleniu całkowitym. Dla RF = 3 kworum wynosi 2, dla RF = 5 wynosi 3. Poziom `QUORUM` wymaga odpowiedzi od tylu replik, ile wynosi kworum — nie od większości w sensie potocznym, tylko od tej konkretnej, wyliczonej liczby.

Wzór R + W > RF. Jeśli poziom spójności odczytu (R, liczba replik odpytywanych przy `SELECT`) i poziom spójności zapisu (W, liczba replik, które muszą potwierdzić `INSERT`/`UPDATE`) spełniają

nierówność R + W > RF, to każdy odczyt musi trafić na przynajmniej jedną replikę, która widziała najnowszy zapis — bo zbiór replik odpytanych przy odczycie i zbiór replik, które potwierdziły zapis, matematycznie muszą się nachodzić. Na liczbach: przy RF = 3, W = QUORUM (2) i R = QUORUM (2), mamy 2 + 2 = 4 > 3 — silna spójność jest zagwarantowana, bo nie da się wybrać dwóch replik do zapisu i dwóch do odczytu spośród trzech bez choć jednej wspólnej. Przy R = ONE (1) i W = ONE (1) mamy 1 + 1 = 2, co nie przekracza RF = 3 — odczyt może trafić na replikę, która jeszcze nie dostała zapisu, i zwrócić dane nieaktualne.

Gossip, snitch i coordinator node

Pojęcie: gossip. Protokół, którym węzły Cassandry wymieniają się informacją o stanie klastra — który węzeł żyje, jaki ma load, jaką wersję schematu — bez centralnego rejestru. Co sekundę każdy węzeł wybiera losowo kilku innych węzłów i wymienia z nimi swoją wiedzę o reszcie klastra, więc informacja o zmianie stanu jednego węzła dociera do wszystkich w kilka sekund, nawet w dużym klastrze. Stan gossipu dla każdego węzła widzimy komendą `nodetool gossipinfo`, opisaną na stronie nodetool: stan klastra i operacje administracyjne. Właściwość `Gossip active` w wyniku `nodetool info` mówi, czy dany węzeł w ogóle bierze udział w tej wymianie — jeśli nie, węzeł widoczny jest w klastrze jako `DN`, mimo że sam proces `cassandra` może działać poprawnie.

Pojęcie: snitch. Komponent, który mówi Cassandrze, do jakiego racku i datacenter należy dany węzeł, i na tej podstawie liczy „odległość" między węzłami przy podejmowaniu decyzji o replikacji i o tym, którą replikę odpytać przy odczycie. `GossipingPropertyFileSnitch` (czytający przynależność z lokalnego pliku `cassandra-rackdc.properties` i propagujący ją dalej przez gossip) jest wyborem właściwym w większości wdrożeń produkcyjnych.

Pojęcie: coordinator node. Węzeł, do którego klient faktycznie się połączył i wysłał zapytanie — niekoniecznie ten, który przechowuje dane. Coordinator na podstawie tokenu wylicza, które węzły są odpowiedzialne za dany wiersz, przekazuje im zapytanie, czeka na tyle odpowiedzi, ile wymaga poziom spójności, i dopiero wtedy odpowiada klientowi. Każdy węzeł w klastrze może pełnić rolę coordinatora dla dowolnego zapytania — to kolejny przejaw braku węzła głównego.

Trzy mechanizmy uzgadniania stanu

Rozproszenie danych po wielu węzłach i dopuszczenie potwierdzania zapisu przed dotarciem do wszystkich replik oznacza, że repliki będą się od siebie chwilowo różnić. Cassandra utrzymuje je w zgodzie trzema niezależnymi mechanizmami, każdym reagującym na inny rodzaj rozjazdu.

Pojęcie: hinted handoff. Mechanizm uruchamiany w chwili zapisu, gdy jedna z docelowych replik jest chwilowo niedostępna. Coordinator zapisuje u siebie „podpowiedź" (hint) — informację, jaki zapis należał się nieobecnej replice — i odtwarza go automatycznie, gdy tamten węzeł wróci do klastra. Hinty mają ograniczony czas życia (domyślnie 3 godziny w `max_hint_window_in_ms`); po jego przekroczeniu odzyskanie spójności zostaje w całości w gestii pozostałych dwóch mechanizmów.

Pojęcie: read repair. Mechanizm uruchamiany przy okazji zwykłego odczytu. Gdy coordinator pyta więcej niż jedną replikę (co dzieje się zawsze przy poziomach silniejszych niż `ONE`, a sporadycznie nawet przy `ONE`, zgodnie z parametrem `read_repair_chance`), porównuje otrzymane wersje wiersza i jeśli się różnią, wysyła najnowszą wersję do repliki, która miała starszą — naprawa dzieje się jako efekt uboczny odczytu, bez dodatkowego zadania administracyjnego.

Pojęcie: anti-entropy repair (naprawa pełna). Jedyny z trzech mechanizmów wymagający ręcznego albo zaplanowanego uruchomienia — komendą `nodetool repair`. Węzły porównują ze sobą drzewa Merkle (skrócone, hierarchiczne sumy kontrolne danych w danym zakresie tokenów) i wymieniają tylko te fragmenty danych, w których drzewa się różnią, zamiast przesyłać wszystko. To jedyny mechanizm, który dociera też do wierszy, których nikt nigdy nie odczytał (więc read repair ich nie dotknął) i do zapisów starszych niż okno hintów. Dlaczego repair musi być wykonywany cyklicznie i co się stanie, jeśli tego zaniedbamy, rozwija strona Utrzymanie klastra: repair, compaction, czyszczenie.

Mechanizm	Kiedy działa	Wymaga ręcznego uruchomienia
Hinted handoff	w chwili zapisu, gdy replika jest chwilowo niedostępna	nie
Read repair	przy okazji odczytu, gdy odpytane repliki się różnią	nie
Anti-entropy repair	na żądanie albo wg harmonogramu	tak

Twierdzenie CAP i miejsce Cassandry

Ogólne brzmienie twierdzenia CAP i najczęstsze błędne jego odczytanie opisuje strona Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin — tu tylko lokujemy na nim Cassandrę, bez mitologii, która wokół tego narosła.

Cassandra w chwili partycji sieci domyślnie wybiera dostępność — węzeł odcięty od reszty klastra dalej odpowiada na zapytania, potwierdzając zapisy przy poziomie `ONE` nawet bez łączności z pozostałymi replikami, i dogania resztę klastra przez hinted handoff oraz repair, gdy partycja się skończy. To bywa streszczane jako „Cassandra to baza AP", i w konfiguracji domyślnej to trafny skrót. Ale poziomy spójności opisane wyżej pozwalają przesunąć tę wagę w stronę spójności per zapytanie — `ALL` w połączeniu z partycją sieci oznacza, że część replik jest nieosiągalna i zapytanie zwróci błąd zamiast nieaktualnych danych, czyli zachowanie bliższe stronie C. Twierdzenie CAP nie jest więc etykietą przyklejoną raz na zawsze do silnika — jest opisem tego, co silnik *domyślnie* robi w chwili awarii sieci, i Cassandra jest jednym z niewielu silników, które to zachowanie oddają administratorowi jako parametr zapytania, a nie ukrywają za jedną globalną decyzją.

Z życia wzięte

> **Z ŻYCIA WZIĘTE**
>
> O 6:00 w Swarzędzu czterdzieści skanerów loguje się naraz do WMS, a razem z nimi telemetria wózków widłowych zaczyna zapisywać się do klastra cass01, cass02 i cass03. Tomek sprawdza, który węzeł faktycznie odpowiada za dany wózek, i odkrywa, że to nie administrator decyduje o tym przypisaniu, tylko funkcja haszująca identyfikator wózka na pierścieniu tokenów. Kiedy cass02 na chwilę traci łączność sieciową podczas porannego szczytu, reszta klastra dalej przyjmuje zapisy z poziomem spójności ONE, a zaległe dane dogonią go automatycznie przez hinted handoff. Marek z magazynu nawet nie zauważa przerwy, bo system śledzenia wózków działa bez przestoju.

Częste błędy

Mylenie replication factor z liczbą replik biorących udział w konkretnej operacji. RF = 3 mówi, że istnieją trzy kopie wiersza — nie mówi, ile z nich musi odpowiedzieć na dany `SELECT` czy `INSERT`. To reguluje osobno poziom spójności tej konkretnej operacji.

SimpleStrategy w klastrze wielo-DC. Strategia nieświadoma topologii potrafi umieścić wszystkie kopie danych w jednym datacenter, mimo że fizycznie klaster rozciąga się na dwa — awaria tego jednego DC oznacza wtedy utratę dostępu do danych, których replikacja miała temu zapobiec.

Traktowanie `ONE` jako błędu konfiguracyjnego. Niski poziom spójności bywa świadomym wyborem — dla zapisu telemetrii, gdzie pojedynczy nieaktualny odczyt nic nie kosztuje, `ONE` daje realny zysk w latencji i przepustowości, którego `QUORUM` by nie dał.

□□ Ćwiczenia

> Poniższe pytania sprawdzają rozumienie mechanizmu, nie wymagają uruchomionego klastra.

Pytania kontrolne

1. Klaster ma trzy węzły w jednym racku i RF = 3. Dlaczego świadomość racków, którą oferuje NetworkTopologyStrategy, w tym konkretnym przypadku nic nie daje?

Odpowiedź

Bo wszystkie trzy węzły i tak są w jednym racku — strategia świadoma racków rozkłada repliki między różne racki tylko wtedy, gdy więcej niż jeden rack istnieje. Przy jednym racku efekt jest identyczny jak przy SimpleStrategy, mimo innej nazwy strategii w definicji keyspace.

2. Przy RF = 5 zapis wykonano z poziomem `QUORUM`, a odczyt zaraz po nim z poziomem `ONE`. Czy odczyt ma gwarancję zobaczenia tego zapisu? Ile wynosi kworum przy RF = 5?

Odpowiedź

Kworum przy RF = 5 wynosi 3 (`(5/2)+1` = 3, przy dzieleniu całkowitym). W = 3, R = 1, więc R + W = 4, co nie przekracza RF = 5 — nierówność R + W > RF nie jest spełniona, więc odczyt nie ma matematycznej gwarancji trafienia na replikę, która już widziała ten zapis.

3. Węzeł jest widoczny w `nodetool status` jako `DN`, ale proces `cassandra` na nim działa i przyjmuje połączenia lokalnie. Który z trzech mechanizmów uzgadniania — hinted handoff, read repair, anti-entropy repair — najprędzej ucierpi na tej sytuacji, i dlaczego akurat ten?

Odpowiedź

Hinted handoff — bo hinty mają ograniczony czas życia (domyślnie 3 godziny). Jeśli węzeł jest odcięty od gossipu dłużej niż to okno, nagromadzone dla niego podpowiedzi wygasają i nie zostaną odtworzone automatycznie po jego powrocie — odzyskanie spójności zostanie wtedy wyłącznie w gestii read repair i uruchomionego ręcznie `nodetool repair`.

Powiązane: Relacyjne, dokumentowe, kolumnowe, klucz-wartość: mapa rodzin · Model danych: keyspace, tabela, klucz partycji, klucz klastrujący · Utrzymanie klastra: repair, compaction, czyszczenie · nodetool: stan klastra i operacje administracyjne

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

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)
```

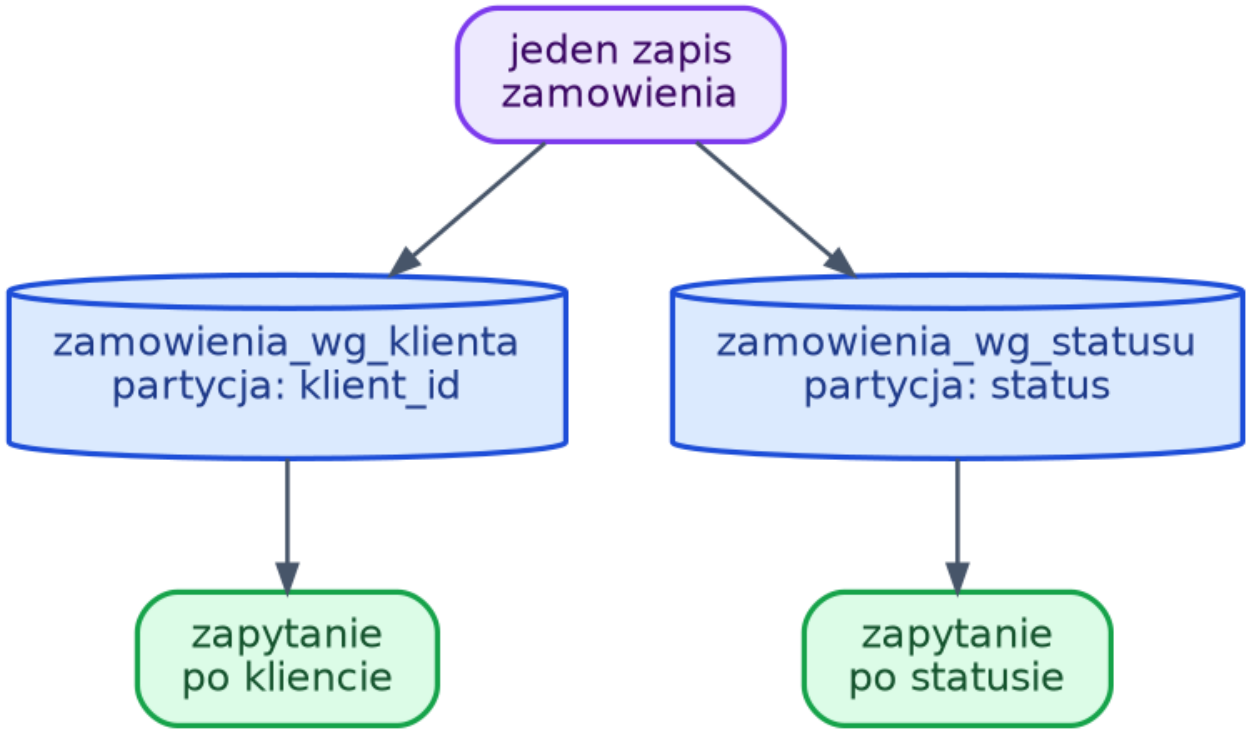
Nawias wewnętrzny obejmuje klucz partycji — tu jednoelementowy, `klient_id` — a `data_zamowienia` i `id_zamowienia` to kolumny klastrujące, według których wiersze tej samej partycji są posortowane, w tej kolejności. Pełną składnię `CREATE TABLE` z kluczem złożonym i `CLUSTERING ORDER BY` pokazujemy na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