Utrzymanie klastra

nodetool, naprawy, kompaktowanie i czyszczenie.

- nodetool: stan klastra i operacje administracyjne
- Utrzymanie klastra: repair, compaction, czyszczenie

nodetool: stan klastra i operacje administracyjne

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

`nodetool` *jest narzędziem administracyjnym, którym sprawdzamy stan klastra i wykonujemy operacje, których nie da się poprosić CQL-em — bo dotyczą samego węzła, nie danych w nim zapisanych.*

Skróty użyte na tej stronie (8)

Skrót	Co znaczy
CQL	Cassandra Query Language: język zapytań Cassandry, podobny do SQL
DC	Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników
DN	Distinguished Name: pełny, jednoznaczny adres jednego obiektu w katalogu LDAP, na przykład konta albo grupy
GC	Garbage Collector: mechanizm Javy sprzątający nieużywaną pamięć
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
JMX	Java Management Extensions: mechanizm Javy do odpytywania i sterowania działającą aplikacją z zewnątrz, tędy Cassandra wystawia swoje metryki
JVM	Java Virtual Machine: maszyna wirtualna, w której działają programy w Javie
RF	Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry

Fiszka

TYLKO ODCZYT

Sprawdzenie stanu wszystkich węzłów klastra: nodetool status

```
sudo nodetool status
```

Zwracamy stan każdego węzła widziany z perspektywy tego, na którym uruchomiliśmy komendę — pierwsza komenda, po którą sięgamy przy jakimkolwiek podejrzeniu problemu z klastrem.

Przykład:

```
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load       Tokens  Owns (effective)  Host
ID                               Rack
UN  10.20.10.36  36.05 GiB  256     34.7%             4f8a1e2d-9b3c-4a11-8e77-
1a2b3c4d5e01  rack1
UN  10.20.10.37  36.18 GiB  256     33.5%             4f8a1e2d-9b3c-4a11-8e77-
1a2b3c4d5e02  rack1
UN  10.20.10.38  33.85 GiB  256     31.8%             4f8a1e2d-9b3c-4a11-8e77-
1a2b3c4d5e03  rack1
```

Interpretacja kolumn. Pierwsza litera (`U`/`D`) mówi, czy węzeł jest widoczny jako aktywny (Up) czy niedostępny (Down) w gossipie; druga litera (`N`/`L`/`J`/`M`) opisuje jego stan — `Normal` (pracuje normalnie), `Leaving` (opuszcza klaster, w trakcie `decommission`), `Joining` (dołącza, streamuje dane), `Moving` (zmienia przypisane tokeny). `UN` — Up/Normal — jest jedynym stanem, który uznajemy za w pełni prawidłowy. `Load` to ilość danych fizycznie przechowywanych przez węzeł. `Owns (effective)` to procent przestrzeni tokenów, za który węzeł odpowiada, uwzględniając replikację. `Host ID` jednoznacznie identyfikuje węzeł niezależnie od jego adresu IP, który mógłby się zmienić. `Rack` pokazuje przynależność deklarowaną w `cassandra-rackdc.properties`, opisanym na stronie Konfiguracja i diagnostyka.

Węzeł widoczny jako `DN` (Down/Normal) sam proces `cassandra` może mieć uruchomiony, ale nie komunikuje się przez gossip z resztą klastra — pierwsze, co sprawdzamy w takiej sytuacji, to właściwość `Gossip active` w wyniku `nodetool info` poniżej.

IP = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **DN** = Distinguished Name: pełny, jednoznaczny adres jednego obiektu w katalogu LDAP, na przykład konta albo grupy · **nodetool** = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie szczegółów danego węzła: nodetool info

```
sudo nodetool info
```

Zwracamy szczegółowy stan węzła, na którym komenda została wykonana — zużycie pamięci, cache'e, procent naprawionych danych.

Przykład:

```
ID                     : 4f8a1e2d-9b3c-4a11-8e77-1a2b3c4d5e01
Gossip active          : true
Thrift active          : false
Native Transport active: true
Load                   : 36.05 GiB
Generation No          : 1708075904
Uptime (seconds)       : 1165490
Heap Memory (MB)       : 4856.17 / 7978.00
Off Heap Memory (MB)   : 35.00
Data Center            : dc1
Rack                   : rack1
Exceptions             : 0
Key Cache              : entries 7595, size 1.99 MiB, capacity 100 MiB, 19517401 hits,
19848210 requests, 0.983 recent hit rate, 14400 save period in seconds
Row Cache              : entries 0, size 0 bytes, capacity 0 bytes, 0 hits, 0 requests,
NaN recent hit rate, 0 save period in seconds
Counter Cache          : entries 0, size 0 bytes, capacity 50 MiB, 0 hits, 0 requests,
NaN recent hit rate, 7200 save period in seconds
Chunk Cache            : entries 15872, size 992 MiB, capacity 992 MiB, 108263412
misses, 138979929 requests, 0.221 recent hit rate, 47.396 microseconds miss latency
Percent Repaired       : 72.27%
Token                  : (invoke with -T/--tokens to see all 256 tokens)
```

Zużycie `Heap Memory` warto sprawdzać systematycznie — węzeł stale pracujący blisko górnej granicy sterty jest kandydatem do długich pauz GC opisanych w tabeli objawów na stronie Konfiguracja i diagnostyka. `Gossip active: true` jest warunkiem koniecznym, żeby węzeł był widoczny w `nodetool status` jako `UN`; `false` oznacza rozłączenie od reszty klastra mimo działającego procesu.

□□ **GC** = Garbage Collector: mechanizm Javy sprzątający nieużywaną pamięć · **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Wyświetlenie przypisania tokenów do węzłów: nodetool ring

```
sudo nodetool ring
```

Zwracamy pełną listę tokenów w pierścieniu opisanym na stronie Architektura Cassandry: pierścień, węzeł, replikacja wraz z węzłem, do którego każdy należy — przy 256 vnodes na węzeł wynik liczy setki wierszy, więc w praktyce sięgamy po tę komendę rzadziej niż po `status`, głównie przy diagnozowaniu nierównego rozłożenia danych.

Przykład:

```
Datacenter: dc1
==========
Address      Rack   Status State   Load       Owns    Token
                                                       -9223372036854775808
10.20.10.36  rack1  Up     Normal  36.05 GiB  34.7%   -9187343239835811840
10.20.10.37  rack1  Up     Normal  36.18 GiB  33.5%   -9143872360128475648
10.20.10.38  rack1  Up     Normal  33.85 GiB  31.8%   -9098274652910384192
...
```

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Wyświetlenie informacji o topologii klastra: nodetool describecluster

```
sudo nodetool describecluster
```

Zwracamy nazwę klastra, użyty snitch i partycjoner oraz wersje schematu widziane przez poszczególne węzły — odpowiednik `DESCRIBE CLUSTER` z `cqlsh`, opisanego na stronie cqlsh: klient wiersza poleceń, wywoływany z poziomu powłoki zamiast sesji CQL.

Przykład:

```
Cluster Information:
    Name: produkcja
    Snitch: org.apache.cassandra.locator.GossipingPropertyFileSnitch
    DynamicEndPointSnitch: enabled
    Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
    Schema versions:
        a1b2c3d4-e5f6-3789-a123-456789abcdef: [10.20.10.36, 10.20.10.37, 10.20.10.38]
```

Więcej niż jedna wersja schematu w wyniku oznacza, że ostatnia zmiana struktury (`CREATE TABLE`, `ALTER TABLE`) nie zdążyła się jeszcze rozpropagować do wszystkich węzłów przez gossip — zwykle kwestia kilku sekund, ale przy dużym klastrze warto to sprawdzić przed kolejną zmianą schematu.

CQL = Cassandra Query Language: język zapytań Cassandry, podobny do SQL · **nodetool** = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie ruchu sieciowego węzła: nodetool netstats

```
sudo nodetool netstats
```

Zwracamy informacje o trwających transferach danych między węzłami (streaming — na przykład w trakcie `rebuild` czy dołączania nowego węzła) oraz statystyki read repair.

Przykład:

```
Mode: NORMAL
Not sending any streams.
Read Repair Statistics:
Attempted: 128
Mismatch (Blocking): 2
Mismatch (Background): 14
Pool Name                    Active   Pending      Completed   Dropped
Large messages                  n/a         0              0         0
Small messages                  n/a         0        4519823         0
Gossip messages                 n/a         0         849213         0
```

W trakcie działania `nodetool rebuild` albo `nodetool repair` ta sama komenda, wywołana z innego węzła, pokazuje aktywne strumienie danych i pozwala śledzić postęp operacji, która sama w sobie nie wypisuje niczego w trakcie działania.

nodetool = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie obciążenia puli wątków: nodetool tpstats

```
sudo nodetool tpstats
```

Zwracamy liczbę zadań aktywnych, oczekujących i odrzuconych w poszczególnych pulach wątków silnika — pierwsze miejsce, w którym szukamy przyczyny, gdy węzeł odpowiada wolniej niż zwykle.

Przykład:

```
Pool Name                    Active   Pending      Completed   Blocked  All time blocked
ReadStage                         2         0        8321904         0                 0
MutationStage                     0         0       15234771         0                 0
CompactionExecutor                1         3            412         0
0
ValidationExecutor                0         0             28         0
0
GossipStage                       0         0         849213         0
0
HintsDispatcher                   0         0            842         0
0

Message type           Dropped
READ                         0
MUTATION                     0
RANGE_SLICE                  0
```

Stale rosnąca kolumna `Pending` przy `MutationStage` albo `ReadStage` oznacza, że węzeł nie nadąża przyjmować obciążenia w bieżącym tempie — wartości w kolumnie `Dropped` większe od zera oznaczają, że część zapytań została odrzucona, zanim w ogóle doszła do przetworzenia.

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Sprawdzenie stanu kompaktowania: nodetool compactionstats

```
sudo nodetool compactionstats
```

Zwracamy listę aktualnie trwających kompaktowań oraz liczbę zadań oczekujących w kolejce — omówienie samego mechanizmu kompaktowania na stronie Utrzymanie klastra: repair, compaction, czyszczenie.

Przykład:

```
pending tasks: 3
- aplikacja.klienci: 2
- aplikacja.zamowienia_wg_klienta: 1
id                                   compaction type keyspace   table
completed   total       unit  progress
a1b2c3d4-1111-2222-3333-444455556666 Compaction      aplikacja  zamowienia_wg_klienta
1073741824  2147483648  bytes 50.00%
Active compaction remaining time: 0h00m12s
```

Trwale rosnąca liczba `pending tasks` (a nie chwilowy skok po dużym imporcie danych) jest sygnałem, że węzeł nie nadąża kompaktować danych w tempie, w jakim je zapisuje — omówione w tabeli objawów na stronie Konfiguracja i diagnostyka.

nodetool = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie statystyk pracy odśmiecacza pamięci: nodetool gcstats

```
sudo nodetool gcstats
```

Zwracamy skumulowane statystyki działania Garbage Collectora JVM od ostatniego zapytania tą komendą — długość i częstotliwość pauz GC.

Przykład:

```
Interval (ms) Max GC Elapsed (ms) Total GC Elapsed (ms) Stdev GC Elapsed (ms) GC
Reclaimed (MB) Collections Direct Memory Bytes
       600174                  45                   892                   6.21
485293.0          18                -1
```

`Max GC Elapsed` rzędu pojedynczych sekund albo więcej jest sygnałem długich pauz GC — w tym czasie węzeł nie odpowiada na żadne zapytania, co z perspektywy reszty klastra bywa nie do odróżnienia od chwilowej awarii sieci.

JVM = Java Virtual Machine: maszyna wirtualna, w której działają programy w Javie · **GC** = Garbage Collector: mechanizm Javy sprzątający nieużywaną pamięć · **nodetool** = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie statystyk tabel: nodetool tablestats

```
sudo nodetool tablestats NazwaKeyspace
```

Zwracamy szczegółowe statystyki każdej tabeli danego keyspace — rozmiar na dysku, liczbę partycji, statystyki bloom filtra. `nodetool cfstats` jest starszym aliasem tej samej komendy, zachowanym dla zgodności wstecznej ze starszymi wersjami Cassandry.

Przykład:

```
Keyspace: aplikacja
    Read Count: 8452193
    Read Latency: 0.842 ms
    Write Count: 15234771
    Write Latency: 0.113 ms
    Pending Flushes: 0
        Table: klienci
        SSTable count: 4
        Space used (live): 512.34 MiB
        Number of partitions (estimate): 1204832
        Memtable cell count: 8213
        Memtable data size: 4.21 MiB
        Local read latency: 0.612 ms
        Local write latency: 0.098 ms
        Bloom filter false positives: 12
        Bloom filter false ratio: 0.00001
        Compacted partition mean bytes: 421
```

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Sprawdzenie tego samego przez starszy alias: nodetool cfstats

```
sudo nodetool cfstats NazwaKeyspace.NazwaTabeli
```

Zwraca identyczne dane, co `tablestats`, dla pojedynczej wskazanej tabeli — nazwa `cfstats` (column family stats) pochodzi z czasów, gdy Cassandra jeszcze nie używała terminologii CQL-owej „tabela".

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Sprawdzenie, które węzły przechowują dany wiersz: nodetool getendpoints

```
sudo nodetool getendpoints NazwaKeyspace NazwaTabeli WartoscKluczaPartycji
```

Zwracamy listę adresów IP węzłów faktycznie przechowujących repliki wiersza o podanym kluczu partycji — użyteczne przy diagnozowaniu, dlaczego konkretny odczyt zwraca nieoczekiwany wynik: sprawdzamy bezpośrednio na tych właśnie węzłach, czy dane są zgodne.

Przykład:

```
10.20.10.37
10.20.10.38
10.20.10.36
```

Trzy adresy przy RF = 3 — kolejność nie jest znacząca, to po prostu lista replik.

□□ **IP** = Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection) · **RF** = Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry · **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Sprawdzenie stanu gossipu: nodetool gossipinfo

```
sudo nodetool gossipinfo
```

Zwracamy surowy stan gossipu opisanego na stronie Architektura Cassandry: pierścień, węzeł, replikacja — wszystko, co dany węzeł wie o sobie i o innych węzłach klastra.

Przykład:

```
/10.20.10.36
  generation:1708075904
  heartbeat:2456789
  STATUS:16:NORMAL,-9223372036854775808
  LOAD:2456780:3.868832768E10
  SCHEMA:12:a1b2c3d4-e5f6-3789-a123-456789abcdef
```

```
DC:8:dc1
RACK:10:rack1
RELEASE_VERSION:4:4.1.3
RPC_ADDRESS:3:10.20.10.36
HOST_ID:2:4f8a1e2d-9b3c-4a11-8e77-1a2b3c4d5e01
RPC_READY:22:true
NATIVE_ADDRESS_AND_PORT:33:10.20.10.36:9042
```

nodetool = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Wypięcie węzła z przyjmowania zapisów: nodetool drain

```
sudo nodetool drain
```

Tzw. wypięcie węzła oznacza w praktyce zablokowanie możliwości zapisu — węzeł rozłącza obecne połączenia klienckie, blokuje nowe i przechodzi w stan, w którym nie przyjmuje już żadnych mutacji, zrzucając przy okazji wszystkie dane z memtable do SSTable na dysku. Szczególnie przydatne przed aktualizacją systemu operacyjnego danego węzła albo przed jego kontrolowanym restartem — po `drain` możemy bezpiecznie zatrzymać usługę, wiedząc, że wszystkie dane z pamięci trafiły już na dysk.

Po wykonaniu tej komendy węzeł będzie widoczny w `nodetool status` z reszty klastra jako `DN` — w tym konkretnym przypadku jest to zachowanie prawidłowe, nie objaw awarii.

DN = Distinguished Name: pełny, jednoznaczny adres jednego obiektu w katalogu LDAP, na przykład konta albo grupy · **nodetool** = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Wymuszenie zrzutu memtable na dysk: nodetool flush

```
sudo nodetool flush NazwaKeyspace
```

Wymuszamy natychmiastowy zrzut danych z memtable do SSTable dla wskazanego keyspace (albo dla całego węzła, jeśli keyspace pominiemy), bez czekania, aż zrobi to automatyczny mechanizm opisany na stronie Utrzymanie klastra: repair, compaction, czyszczenie. Wykonujemy to przed operacjami wymagającymi spójnego obrazu danych na dysku — na przykład przed snapshotem, opisanym na stronie Backup i odtwarzanie, albo przed planowanym zatrzymaniem węzła.

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Usunięcie danych nienależących już do węzła: nodetool cleanup

```
sudo nodetool cleanup NazwaKeyspace
```

Usuwamy z dysku dane, za które węzeł formalnie już nie odpowiada — sytuacja, która powstaje po dołączeniu nowego węzła do klastra: zakresy tokenów zostają przeliczone, ale stare dane fizycznie zostają na starym węźle, dopóki `cleanup` ich nie usunie. Operacja jest bezpieczna, bo dotyczy wyłącznie danych zreplikowanych już gdzie indziej zgodnie z aktualną topologią.

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Wymuszenie kompaktowania: nodetool compact

```
sudo nodetool compact NazwaKeyspace NazwaTabeli
```

Wymuszamy natychmiastowe pełne kompaktowanie wskazanej tabeli, poza normalnym harmonogramem strategii kompaktowania opisanej na stronie Utrzymanie klastra: repair, compaction, czyszczenie. Operacja bywa kosztowna dla dużych tabel (potrzebuje chwilowo dodatkowego miejsca na dysku, rzędu rozmiaru kompaktowanych danych) i przy strategii `SizeTieredCompactionStrategy` skleja wszystkie SSTable w jedną, co bywa niepożądane — świadomie sięgamy po tę komendę, nie uruchamiamy jej rutynowo.

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Naprawa spójności danych: nodetool repair

```
sudo nodetool repair --full --ignore-unreplicated-keyspaces
```

Uruchamiamy anti-entropy repair opisany na stronie Architektura Cassandry: pierścień, węzeł, replikacja — pełną naprawę porównującą drzewa Merkle tego węzła z jego replikami. Flaga `--ignore-unreplicated-keyspaces` pomija keyspace, które na tym węźle w ogóle nie mają

żadnej repliki (np. keyspace z RF mniejszym niż liczba węzłów w danym DC), zamiast kończyć się dla nich błędem.

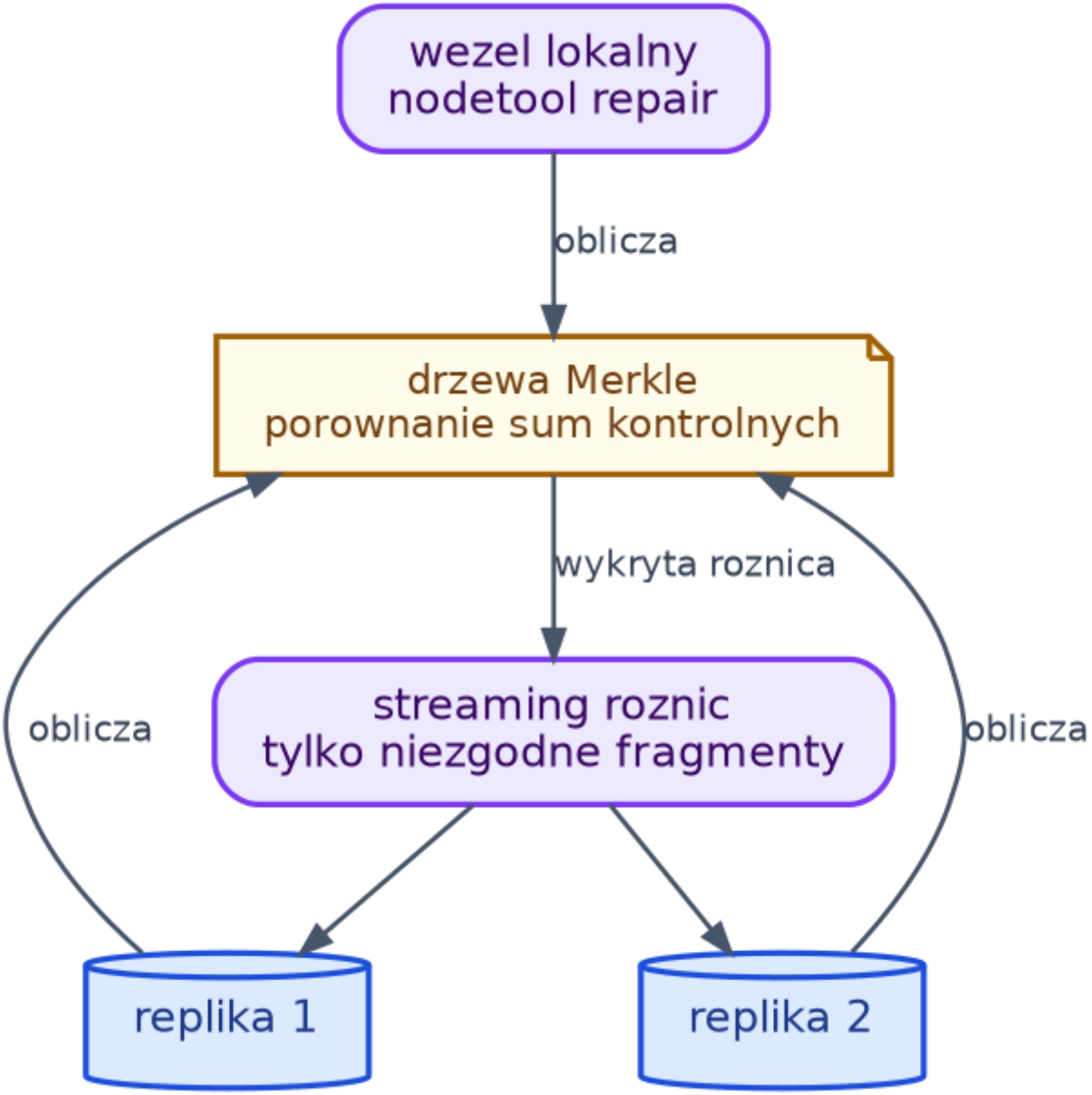
Wariant bez `--full` wykonuje naprawę przyrostową (incremental repair), domyślną od Cassandry 4.0 — naprawia tylko dane zapisane od poprzedniej naprawy, zamiast przeliczać drzewa Merkle dla całego zbioru danych za każdym razem:

```
sudo nodetool repair
```

Wariant z `-pr` (primary range) ogranicza naprawę wyłącznie do zakresu tokenów, za który dany węzeł jest podstawową (pierwszą) repliką, zamiast wszystkich zakresów, które przechowuje jako replika dowolnego rzędu — uruchomiony po kolei na każdym węźle klastra pokrywa cały zbiór danych dokładnie raz, zamiast wielokrotnie:

```
sudo nodetool repair -pr
```

Różnicę między tymi trzema wariantami i harmonogram, w jakim je łączymy, rozwija strona Utrzymanie klastra: repair, compaction, czyszczenie.



Anti-entropy repair: porównanie drzew Merkle i naprawa rozjazdów między replikami (kliknięcie otwiera pełny rozmiar)

RF = Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry · **DC** = Domain Controller: serwer, który trzyma Active Directory i loguje użytkowników · **nodetool** = narzędzie administracyjne węzła Cassandry

ZMIENIA SYSTEM

Odbudowanie węzła z danych innych węzłów: nodetool rebuild

```
sudo nodetool rebuild -- NazwaDatacenter
```

W przypadku uszkodzenia danych węzła — zwykle po wielu błędach związanych ze storage'em — możemy wykonać jego odbudowę, streamując dane od podstaw z innych węzłów klastra, zamiast próbować naprawiać uszkodzone pliki punktowo. Operacja jest ryzykowna i niesie realne obciążenie sieci i dysków pozostałych węzłów — wykonujemy ją tylko w uzasadnionych przypadkach.

Przed jej wykonaniem zatrzymujemy usługę `cassandra.service` na odbudowywanym węźle i usuwamy całą zawartość jego katalogów danych — `rebuild` zakłada, że zaczyna od pustego węzła, nie że scala nowe dane z istniejącymi.

Postęp możemy śledzić z innego węzła komendą `nodetool netstats` opisaną wyżej. Przykład fragmentu logu streamowania w trakcie działania:

```
[Stream #a1b2c3d4-1111-2222-3333-444455556666] Sending 91 files, 21082895050 bytes
total. Already sent 24 files, 395406967 bytes total
/var/lib/cassandra/data/aplikacja/zamowienia_wg_klienta-3b7ea380.../md-1047-big-Data.db
497636/497636 bytes (100%) sent to /10.20.10.37
/var/lib/cassandra/data/aplikacja/klienci-3c77b560.../md-115-big-Data.db 83138/83138
bytes (100%) sent to /10.20.10.37
```

Po zakończonym procesie zalecane jest wykonanie pełnej naprawy całego klastra komendą `nodetool repair --full`, żeby domknąć ewentualne rozjazdy powstałe w trakcie odbudowy.

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Wycofanie węzła z klastra: nodetool decommission

```
sudo nodetool decommission
```

Wykonywana na samym usuwanym węźle — przekazuje jego dane pozostałym węzłom klastra zgodnie z nową topologią (bez tego węzła), a dopiero po zakończeniu streamingu pozwala go bezpiecznie zatrzymać. Operacja może trwać długo przy dużym wolumenie danych; węzeł w trakcie jej wykonywania jest widoczny w `nodetool status` w stanie `Leaving` (`UL`).

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ **TYLKO ODCZYT**

Wymuszone usunięcie martwego węzła: nodetool removenode

```
sudo nodetool removenode NazwaHostId
```

Wykonywana z dowolnego innego, żyjącego węzła klastra, gdy węzeł do usunięcia jest już trwale niedostępny i nie może sam wykonać `decommission` — pozostałe węzły przejmują odpowiedzialność za jego zakresy tokenów i odtwarzają brakujące repliki między sobą. Host ID usuwanego węzła znajdujemy w `nodetool status` sprzed jego awarii.

Postęp sprawdzamy tą samą komendą z argumentem `status`:

```
sudo nodetool removenode status
```

Przykład:

```
RemovalStatus: Removing token (256 tokens). Waiting for replication confirmation from
[10.20.10.37,10.20.10.38].
```

nodetool = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Ograniczenie przepustowości kompaktowania: nodetool setcompactionthroughput

```
sudo nodetool setcompactionthroughput 64
```

Ustawiamy górny limit przepustowości dysku, jaki kompaktowanie może zająć, w MB/s — sięgamy po to tymczasowo, gdy kompaktowanie w tle zaczyna zauważalnie odbijać się na latencji zapytań klienckich. Wartość `0` oznacza brak limitu. Ustawienie działa tylko do restartu usługi — trwałą zmianę wprowadzamy w `cassandra.yaml`, opisanym na stronie Konfiguracja i diagnostyka.

nodetool = narzędzie administracyjne węzła Cassandry

TYLKO ODCZYT

Sprawdzenie histogramów opóźnień na poziomie coordinatora: nodetool proxyhistograms

```
sudo nodetool proxyhistograms
```

Zwracamy rozkład opóźnień zapytań mierzony na węźle pełniącym rolę coordinatora — czyli opóźnienie odczuwane faktycznie przez klienta, łącznie z czasem oczekiwania na odpowiedzi replik.

Przykład:

```
Percentile     Read Latency    Write Latency    Range Latency
                   (micros)         (micros)         (micros)
50%                  245.00            88.15           612.35
75%                  421.23           127.45          1015.88
95%                 1214.55           298.51          3874.20
99%                 3821.44           891.02          9812.44
Max                48213.00          9821.33        128744.00
```

□□ **nodetool** = narzędzie administracyjne węzła Cassandry

□□ TYLKO ODCZYT

Sprawdzenie histogramów opóźnień dla konkretnej tabeli: nodetool tablehistograms

```
sudo nodetool tablehistograms NazwaKeyspace NazwaTabeli
```

Zwracamy rozkład opóźnień odczytu i zapisu zmierzony lokalnie na tym węźle, wraz z rozkładem liczby SSTable odpytywanych na jeden odczyt i rozmiaru partycji — przydatne przy podejrzeniu, że konkretna tabela ma za szeroko rozrzucone dane po wielu SSTable albo zbyt duże partycje.

Przykład:

```
Percentile  SSTables     Write Latency      Read Latency    Partition Size        Cell
Count
                              (micros)          (micros)             (bytes)
50%             1.00             88.15            612.35
421               12
75%             2.00            127.45           1015.88
1240              34
95%             4.00            298.51           3874.20
8987              210
99%             6.00            891.02           9812.44
```

```
21504                512
Max              12.00           9821.33         128744.00
98213                1840
```

nodetool = narzędzie administracyjne węzła Cassandry

Z życia wzięte

Z ŻYCIA WZIĘTE

W poniedziałek rano cass02 znika z nodetool status jako DN, mimo że proces cassandra na nim wciąż działa i odpowiada lokalnie na cqlsh. Sprawdzamy nodetool info na tym węźle i widzimy Gossip active: false, czyli węzeł faktycznie przestał wymieniać się informacją z resztą klastra przez gossip. Po restarcie usługi węzeł wraca do stanu UN w ciągu kilku sekund, ale zanim to się stanie, uruchamiamy nodetool repair, żeby domknąć rozjazdy powstałe podczas jego nieobecności. Marek w Swarzędzu nie zauważa przerwy, bo pozostałe dwa węzły cały czas przyjmowały telemetrię skanerów.

Czym jest nodetool

Pojęcie: nodetool. Narzędzie wiersza poleceń łączące się z lokalnym (albo wskazanym przez `-h`) węzłem przez JMX i wywołujące na nim operacje administracyjne oraz odpytujące o metryki — status klastra, statystyki kompaktowania, uruchomienie naprawy. W przeciwieństwie do `cqlsh`, `nodetool` nie odpytuje danych zapisanych w tabelach — odpytuje i steruje samym procesem `cassandra`.

```
sudo nodetool -u NazwaUsera -p HasloUzytkownika status
```

Bez `-h` komenda łączy się z węzłem, na którym jest uruchomiona — typowy sposób pracy, bo administrację wykonujemy zwykle zalogowani bezpośrednio na danym węźle. W środowiskach z włączonym uwierzytelnianiem JMX flagi `-u`/`-p` są wymagane przy każdym wywołaniu; wiele zespołów chowa je za aliasem powłoki, żeby nie wpisywać poświadczeń przy każdej komendzie — konto używane do tego celu bywa tym samym kontem, które opisujemy przy okazji monitoringu JMX na stronie Role, uprawnienia i monitoring JMX.

Częste błędy

Interpretowanie `DN` wyłącznie jako awarii usługi. Proces `cassandra` na węźle widocznym jako `DN` bywa uruchomiony i odpowiada lokalnie — przyczyną jest zwykle brak łączności gossipowej z resztą klastra, nie zatrzymana usługa. Pierwsza komenda do sprawdzenia to `nodetool info` na samym podejrzanym węźle, kolumna `Gossip active`.

Uruchomienie `nodetool compact` jako rutynowej czynności porządkowej. Przy `SizeTieredCompactionStrategy` sklejenie wszystkich SSTable w jedną dużą jest nieodwracalne bez kolejnego pełnego kompaktowania i tymczasowo wymaga dodatkowego miejsca na dysku — sięgamy po tę komendę świadomie, nie w ramach cyklicznych zadań konserwacyjnych.

Wykonanie `nodetool rebuild` bez zatrzymania usługi i wyczyszczenia katalogu danych. `rebuild` zakłada start od pustego węzła — uruchomiony na węźle z istniejącymi, częściowo uszkodzonymi danymi może pozostawić klaster w stanie trudniejszym do zdiagnozowania niż przed próbą naprawy.

Powiązane: Architektura Cassandry: pierścień, węzeł, replikacja · Utrzymanie klastra: repair, compaction, czyszczenie · Konfiguracja i diagnostyka · Backup i odtwarzanie

Źródła: dokumentacja Apache Cassandra, rozdz. „nodetool" · `nodetool help`

Utrzymanie klastra: repair, compaction, czyszczenie

Poziom: średni **Dotyczy: Apache Cassandra 3.x / 4.x** **Czas: ~30 min**

Cassandra nie utrzymuje się sama — repair, kompaktowanie i czyszczenie po zmianach topologii to trzy cykliczne zadania, bez których klaster wolno, ale pewnie traci spójność i wydajność.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
RAM	Random Access Memory: pamięć operacyjna
RF	Replication Factor: liczba kopii tego samego wiersza trzymanych naraz w klastrze Cassandry
SSD	Solid State Drive: dysk półprzewodnikowy
TTL	Time To Live: czas życia rekordu lub danych

Z życia wzięte

Z ŻYCIA WZIĘTE

Trzy miesiące po uruchomieniu telemetrii skanerów ktoś zauważa, że stare, dawno usunięte zdarzenia magazynowe wracają w wynikach zapytań, mimo że nikt ich świadomie nie przywracał. Okazuje się, że cykliczny nodetool repair nie był uruchamiany od dwóch miesięcy, dłużej niż gc_grace_seconds tabeli, więc jedna z replik, która była wtedy offline, nigdy nie dowiedziała się o usunięciu i wygląda teraz jak mająca nowszą wersję danych. Zakładamy zadanie cron uruchamiające nodetool repair -pr rotacyjnie po węzłach, w oknie krótszym niż gc_grace_seconds. Wskrzeszone dane znikają po najbliższym cyklu napraw i już nie wracają.

Po co to istnieje

Mechanizmy automatyczne opisane na stronie Architektura Cassandry: pierścień, węzeł, replikacja — hinted handoff i read repair — działają w tle bez naszego udziału, ale żaden z nich nie gwarantuje, że każdy wiersz w klastrze prędzej czy później zostanie sprawdzony i naprawiony. Wiersz, którego nikt nigdy nie odczytał, i węzeł, który był offline dłużej niż okno hintów, wymagają trzeciego mechanizmu — anti-entropy repair — uruchamianego świadomie, na harmonogramie, a nie z automatu.

Naprawa (repair) i dlaczego musi być cykliczna

Pojęcie anti-entropy repair i porównanie z pozostałymi dwoma mechanizmami uzgadniania stanu wprowadza strona Architektura Cassandry: pierścień, węzeł, replikacja — tutaj skupiamy się na tym, dlaczego repair jest zadaniem cyklicznym, nie jednorazową interwencją.

Związek repair z `gc_grace_seconds` opisanym na stronie Model danych: keyspace, tabela, klucz partycji, klucz klastrujący jest bezpośredni: tombstone zapisany przy `DELETE` może zostać trwale usunięty przez kompaktowanie dopiero po upływie `gc_grace_seconds` (domyślnie 10 dni). Jeśli w tym oknie repair nie dotarł do wszystkich replik, a jedna z nich była offline w chwili usunięcia, po powrocie do klastra ta replika nie dowie się nigdy o tombstone — bo do czasu jej powrotu tombstone mógł już zostać skompaktowany i zniknąć z pozostałych replik. Efektem jest zjawisko nazywane wskrzeszeniem danych (zombie data) — dawno usunięty wiersz wraca przy najbliższym read repair, bo replika, która go nigdy nie usunęła, wygląda z perspektywy pozostałych jak posiadająca „nowszą" wersję. Repair wykonywany cyklicznie, w oknie krótszym niż `gc_grace_seconds`, jest jedynym zabezpieczeniem przed tym scenariuszem.

Rodzaje repair: pełny, przyrostowy, primary range

> **Pojęcie: zakres naprawy.** Repair można ograniczać na dwóch niezależnych osiach — ile danych sprawdza (pełny zbiór czy tylko dane od ostatniej naprawy) i które zakresy tokenów w ogóle bierze pod uwagę (wszystkie repliki tego węzła czy tylko te, dla których jest podstawową repliką).

Pełny repair (`--full`) przelicza drzewa Merkle dla całego zbioru danych węzła i porównuje je z odpowiadającymi zakresami na innych replikach — najbardziej kosztowny, ale jedyny, który

gwarantuje sprawdzenie absolutnie wszystkiego, niezależnie od tego, kiedy dane zostały zapisane.

Repair przyrostowy (incremental, domyślny od Cassandry 4.0, komenda bez dodatkowych flag) oznacza dane sprawdzone w poprzedniej naprawie i przy kolejnym uruchomieniu pomija je, ograniczając się do danych zapisanych od tamtej pory — znacznie tańszy w typowej eksploatacji, kosztem tego, że błąd w śledzeniu, które dane są już naprawione, może w rzadkich przypadkach spowodować pominięcie czegoś, co pełny repair by wykrył.

Repair primary range (`-pr`) ogranicza sprawdzane zakresy do tych, w których dany węzeł jest pierwszą (podstawową) repliką — uruchomiony po kolei na każdym węźle klastra pokrywa cały zbiór danych klastra dokładnie raz, zamiast RF razy, jak przy repair bez `-pr` uruchomionym na każdym węźle z osobna. Dokładną składnię wszystkich trzech wariantów pokazuje strona nodetool: stan klastra i operacje administracyjne.

Wariant	Co sprawdza	Koszt	Kiedy używamy
Pełny (`--full`)	cały zbiór danych węzła	wysoki	rzadko, po większej awarii albo migracji
Przyrostowy (domyślny)	dane od poprzedniej naprawy	niski	rutynowo, cyklicznie
Primary range (`-pr`)	tylko zakresy, gdzie węzeł jest podstawową repliką	umiarkowany, ale pokrywa klaster raz	rutynowo, uruchamiany po kolei na każdym węźle

Harmonogram napraw

Repair uruchamiany ręcznie na każdym węźle z osobna, pamiętany w głowie administratora, nie skaluje się poza klaster o garstce węzłów. W praktyce harmonogram napraw zleca się albo systemowemu planiście zadań (`cron` wywołujący `nodetool repair -pr` rotacyjnie po węzłach, w oknie mieszczącym się w `gc_grace_seconds`), albo dedykowanemu narzędziu do orkiestracji napraw, które śledzi stan każdego zakresu tokenów w całym klastrze i sam dba o to, żeby żaden nie został pominięty ani naprawiony niepotrzebnie dwa razy — takim narzędziem jest Cassandra Reaper, przechowujący własny stan w keyspace zwyczajowo nazywanym `reaper_db` i udostępniający panel administracyjny do definiowania harmonogramów per klaster.

Kompaktowanie

> **Pojęcie: kompaktowanie (compaction).** Proces scalania wielu plików SSTable leżących na dysku w mniejszą liczbę większych plików, przy okazji trwale usuwający dane oznaczone tombstone'ami, których `gc_grace_seconds` już minęło, oraz nadpisane wcześniejsze wersje tego samego wiersza. Bez kompaktowania każdy zapis do istniejącego już wiersza tworzyłby

kolejny, osobny fragment danych do przejrzenia przy każdym odczycie tego wiersza — kompaktowanie jest mechanizmem, który utrzymuje koszt odczytu w ryzach mimo modelu zapisu, który nigdy nie modyfikuje danych w miejscu.

Pojęcie: SizeTieredCompactionStrategy (STCS). Domyślna strategia — scala SSTable o zbliżonym rozmiarze w większe, gdy zbierze się ich odpowiednio dużo. Sprawdza się dobrze przy obciążeniu zapisowym bez wyraźnego wzorca czasowego, ale bywa kosztowna miejscowo (do scalenia kilku dużych SSTable potrzeba chwilowo dodatkowego miejsca porównywalnego z ich sumarycznym rozmiarem) i może pozostawiać starą wersję danych nieskompaktowaną długo, jeśli SSTable, w którym leży, nie trafi akurat do tej samej grupy rozmiaru.

Pojęcie: LeveledCompactionStrategy (LCS). Organizuje SSTable w ponumerowane poziomy o rosnącym rozmiarze, gwarantując, że w obrębie jednego poziomu zakresy kluczy się nie nakładają — dzięki temu odczyt pojedynczego wiersza sprawdza znacznie mniej plików niż przy STCS. Kosztem jest większy ruch dyskowy generowany przez samo kompaktowanie (więcej operacji zapisu i odczytu w tle), więc LCS sprawdza się na dyskach o wysokiej przepustowości (SSD) i przy obciążeniu z przewagą odczytów nad zapisami.

Pojęcie: TimeWindowCompactionStrategy (TWCS). Grupuje SSTable według okna czasowego, w którym dane zostały zapisane, zamiast według rozmiaru — dane z tego samego okna czasowego (na przykład jednej doby) trafiają razem, a starsze okna po zamknięciu w ogóle nie są już kompaktowane razem z nowszymi. Naturalny wybór dla danych szeregów czasowych z TTL, gdzie całe okno czasowe wygasa naraz — usunięcie całego wygasłego SSTable jest wtedy operacją natychmiastową, bez potrzeby przepisywania danych.

Strategia	Kiedy wybieramy
SizeTieredCompactionStrategy	ogólne zastosowanie, brak wyraźnego wzorca czasowego w danych
LeveledCompactionStrategy	przewaga odczytów nad zapisami, szybkie dyski, potrzeba przewidywalnej liczby SSTable na odczyt
TimeWindowCompactionStrategy	dane szeregów czasowych z TTL, zapis w kolejności rosnącego czasu

Strategię deklarujemy przy tabeli, analogicznie do parametrów cache'owania pokazanych na stronie CQL: zapytania i operacje na danych: `WITH compaction = {'class': 'TimeWindowCompactionStrategy', 'compaction_window_unit': 'DAYS', 'compaction_window_size': 1}`.

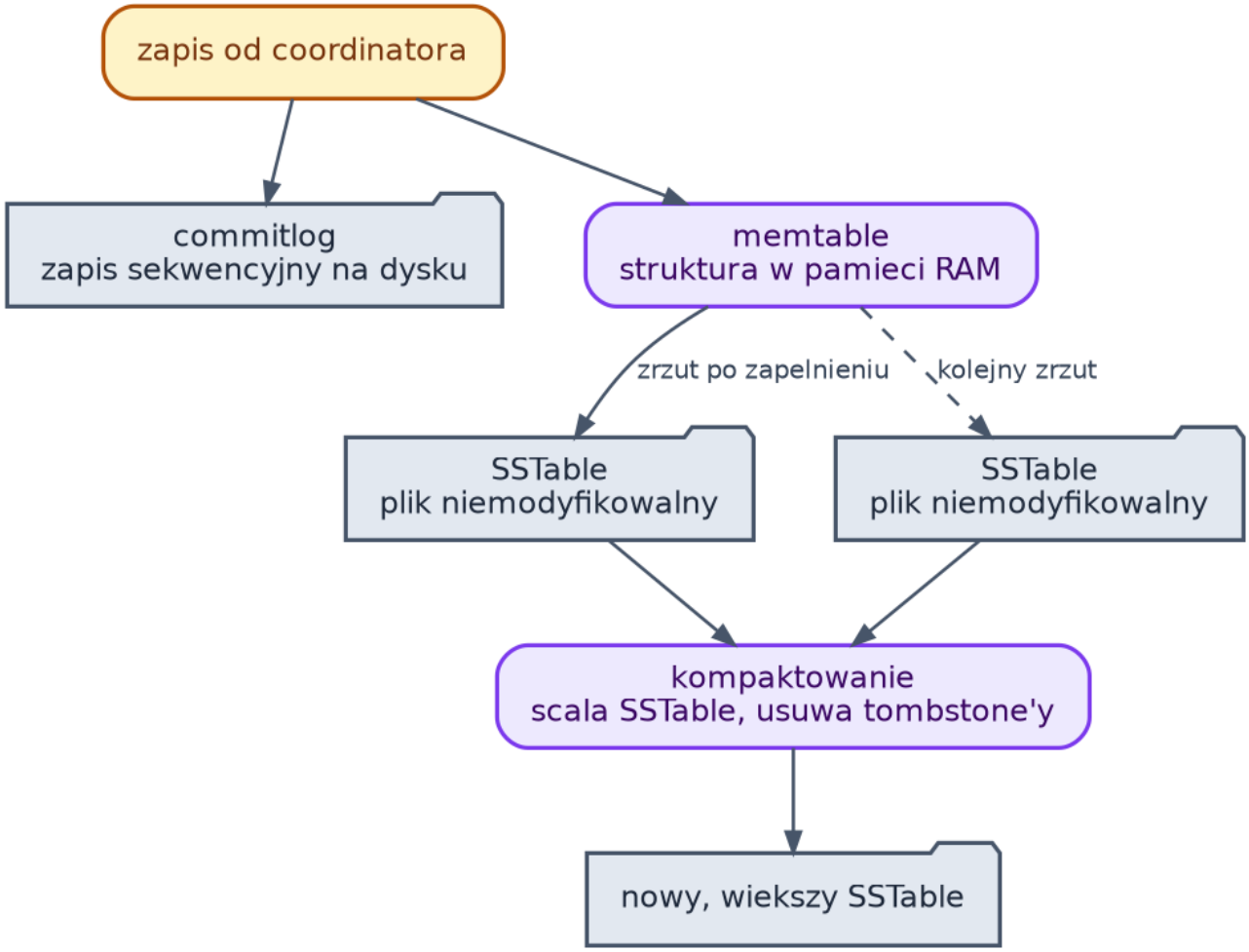
Droga zapisu: memtable, commitlog, SSTable

> **Pojęcie: commitlog.** Plik na dysku, do którego Cassandra zapisuje każdą mutację natychmiast po jej otrzymaniu, sekwencyjnie, zanim jeszcze cokolwiek trafi do struktury w pamięci — zabezpieczenie na wypadek awarii węzła przed zrzuceniem danych z pamięci na dysk. Jego rolę i miejsce w gwarancji trwałości zapisu opisuje ogólnie, dla wszystkich silników, strona Log transakcyjny: WAL, redo, transaction log.

> **Pojęcie: memtable.** Struktura w pamięci RAM węzła, do której trafia zapis zaraz po commitlogu — jedna memtable na tabelę, posortowana według klucza, żeby zrzut na dysk mógł zapisać dane w gotowej, posortowanej kolejności. Odczyt świeżo zapisanego wiersza, zanim memtable zostanie zrzucona, czyta właśnie z niej, nie z dysku.

> **Pojęcie: SSTable (Sorted String Table).** Niemodyfikowalny plik na dysku, powstały ze zrzutu memtable, gdy ta osiągnie swój limit rozmiaru (parametr `memtable_heap_space` opisany na stronie Konfiguracja i diagnostyka) — po zapisaniu SSTable nigdy nie jest modyfikowany, tylko odczytywany albo w całości usuwany przez kompaktowanie.

Droga pojedynczego zapisu wygląda więc tak: klient wysyła mutację do coordinatora, ten przekazuje ją do replik; każda replika zapisuje ją najpierw do commitlogu (sekwencyjnie, na dysku), potem do memtable (w pamięci); gdy memtable się zapełni, zostaje zrzucona jako nowy, niemodyfikowalny plik SSTable; z czasem kompaktowanie scala rosnącą liczbę SSTable w mniejszą liczbę większych, usuwając po drodze przedawnione tombstone'y i nadpisane wersje.



Droga zapisu od commitlogu przez memtable do SSTable i kompaktowania (kliknięcie otwiera pełny rozmiar)

Cleanup po zmianie topologii

Dołączenie nowego węzła do klastra przelicza granice zakresów tokenów, za które odpowiadają istniejące węzły — część danych, którą stary węzeł wcześniej przechowywał jako swoją, formalnie przechodzi teraz pod odpowiedzialność nowego węzła. Stare dane fizycznie zostają jednak na dysku dotychczasowego węzła, dopóki nie wykonamy `nodetool cleanup`, opisanego na stronie nodetool: stan klastra i operacje administracyjne — do tego momentu węzeł po prostu zajmuje więcej miejsca na dysku, niż wynikałoby z jego aktualnego udziału w pierścieniu. Cleanup wykonujemy po każdym dołączeniu nowego węzła, na każdym z pozostałych węzłów z osobna, poza godzinami szczytu — jest to operacja czasochłonna, choć bezpieczna, bo dotyczy wyłącznie danych już zreplikowanych gdzie indziej zgodnie z nową topologią.

Monitorowanie opóźnień i pending tasks

Trzy sygnały sprawdzane rutynowo, wszystkie dostępne komendami opisanymi na stronie nodetool: stan klastra i operacje administracyjne: liczba `pending tasks` w `nodetool compactionstats` (rosnąca trwale, nie chwilowo po dużym imporcie, oznacza, że węzeł nie nadąża kompaktować), kolumna `Pending` w `nodetool tpstats` dla `ReadStage` i `MutationStage` (rosnąca oznacza zator w przetwarzaniu zapytań) oraz percentyle w `nodetool proxyhistograms` i `nodetool tablehistograms` (percentyl 99 rosnący z dnia na dzień, nawet przy stabilnym percentylu 50, jest wczesnym sygnałem problemu, zanim odbije się on na medianie odczuwanej przez większość klientów). Interpretację tych wartości w kontekście konkretnych objawów awarii rozwija tabela na stronie Konfiguracja i diagnostyka.

Częste błędy

Brak cyklicznego repair, bo „klaster działa dobrze". Skutek zaniedbanego repair nie jest widoczny natychmiast — ujawnia się dopiero jako wskrzeszone dane albo rozjazd między replikami, zwykle długo po tym, jak przyczyna (pominięty cykl naprawy) przestała być oczywista.

Zmiana strategii kompaktowania na LCS bez sprawdzenia obciążenia dysków. LCS generuje więcej operacji I/O w tle niż STCS — na wolniejszych dyskach ta zmiana potrafi pogorszyć sytuację, którą miała poprawić.

Cleanup pominięty po dodaniu węzła. Dane nienależące już do węzła nie znikają same — bez `cleanup` węzeł zajmuje więcej miejsca, niż powinien, co przy kolejnym dodaniu węzła bywa mylnie odczytywane jako nierówne rozłożenie danych w klastrze.

□□ Ćwiczenia

> Pytania sprawdzają rozumienie mechanizmów utrzymania, nie wymagają uruchomionego klastra.

Pytania kontrolne

1. `gc_grace_seconds` tabeli wynosi 10 dni, a repair na całym klastrze wykonuje się raz na 21 dni. Jakie ryzyko to stwarza?

Odpowiedź

Ryzyko wskrzeszenia usuniętych danych — jeśli replika była offline dłużej niż 10 dni (albo po prostu nie została naprawiona w tym oknie), tombstone mógł już zostać trwale usunięty przez kompaktowanie na pozostałych replikach, zanim ta replika zdążyła się o nim dowiedzieć. Po jej powrocie do klastra wygląda ona, jakby miała nowszą wersję danych, i przy najbliższym read repair dawno usunięty wiersz wraca.

2. Tabela przechowuje dane telemetryczne z TTL 30 dni, zapisywane w kolejności rosnącego czasu. Którą strategię kompaktowania wybieramy i dlaczego nie SizeTieredCompactionStrategy?

Odpowiedź

TimeWindowCompactionStrategy — grupuje dane według okna czasowego, więc całe okno wygasłe wraz z TTL można usunąć jednym usunięciem pliku SSTable, bez przepisywania danych. SizeTieredCompactionStrategy grupowałaby SSTable według rozmiaru, mieszając dane z różnych okresów w tych samych plikach, co utrudniałoby tanie usunięcie wygasłych danych.

Powiązane: Architektura Cassandry: pierścień, węzeł, replikacja · Model danych: keyspace, tabela, klucz partycji, klucz klastrujący · nodetool: stan klastra i operacje administracyjne · Konfiguracja i diagnostyka · Log transakcyjny: WAL, redo, transaction log

Źródła: dokumentacja Apache Cassandra, rozdz. „Compaction" i „Repair" · dokumentacja Cassandra Reaper