Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Piotr Grabowski
Nr albumu: 394307

Piotr Wojtczak
Nr albumu: 394980

Michał Szostek
Nr albumu: 394846

Wojciech Bączkowski
Nr albumu: 394071

Replikacja bazy danych ScyllaDB z użyciem klienta Kafki

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
mgr Michała Możdżonka

Warszawa, czerwiec 2020

Streszczenie

Praca skupia się na przedstawieniu implementacji serwisu replikującego rekordy między instancjami nierelacyjnych systemów zarządzania bazami danych Scylla, a także problemów wydajnościowych i poprawnościowych związanych z tym zagadnieniem. Opisane rozwiązanie utylizuje brokera wiadomości Apache Kafka jako pośrednika w tym procesie, ustalając tym samym protokół komunikacji na protokół Kafka wzbogacony o schematy wspierane przez platformę Confluent. W tak opisanym modelu macierzysta instancja pełni rolę producenta, a instancja docelowa - konsumenta.

W celu uzyskania wysokiej wydajności, zaimplementowany został również klient producenta oparty o framework Seastar. Ze względu na całkowitą zależność Scylli od Seastara oraz jego istotę w jej kodzie, opisany klient jest natywny dla systemu zarządzania danymi Scylla. Zaproponowane rozwiązanie wykorzystuje ten fakt, a także bezpośrednie benefity wynikające ze wspomnianego frameworka, w celu zwiększenia szeroko rozumianej efektywności serwisu.

Praca zawiera opis rezultatów i zestawienie ich z istniejącymi na rynku rozwiązaniami, w szczególności zysk z zastąpienia pośredników na etapie produkcji natywnym klientem. Przedstawione zostają również możliwe dalsze opcje usprawnienia implementacji całego procesu.

Słowa kluczowe

Scylla, Kafka, replikacja, baza danych, producent, Seastar

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

10011007.10010940.10010971.10010972.10010975 Software and its engineering Publish-subscribe / event-based architectures 300

Tytuł pracy w języku angielskim

ScyllaDB database replication using custom Kafka client

Spis treści

Rozdział 1

Wprowadzenie

Na świecie jest coraz więcej urządzeń podłączonych do sieci i nie przewiduje się w najbliższych latach zaniku tej tendencji wzrostowej (zob. [Stat01]). Stały wzrost liczby urządzeń w naturalny sposób stwarza większe zapotrzebowanie na szybkie i skalowalne systemy.

Wśród nich można znaleźć m.in.: Apache Kafka – popularny broker wiadomości wśród rozwiązań opartych na wzorcu wydawca-subskrybent (ang. publish-subscribe), jak i produkt o nazwie Scylla – nierelacyjna baza danych (ang. NoSQL).

Nasz projekt zwiększa integrację między tymi systemami, otwierając drogę do nowych sposobów na zaspokajanie potrzeb rynku, a następnie sam z tego korzysta.

Implementowany serwis replikujący nie jest bowiem jedynie przykładem zastosowania. Został wykonany na zamówienie firmy zlecającej, co potwierdza istnienie popytu na takie projekty.

1.1. Zleceniodawca

Firmą zlecającą projekt jest ScyllaDB.

1.2. Apache Kafka

Otwartoźródłowy broker wiadomości. Obecnie rozwijany przez Apache Software Foundation. Projekt stawia sobie za cel dostarczenie rozwiązania pozwalającego na wydajne przesyłanie wiadomości w czasie rzeczywistym. Duże przepustowości, mały czas oczekiwania.

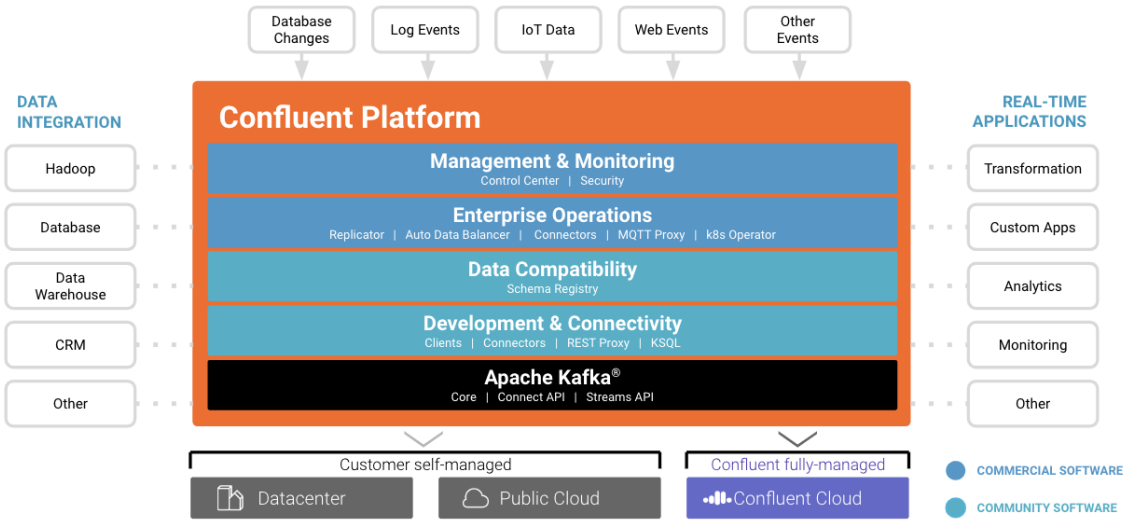
Kiedy jest mowa o Producer API, to właśnie chodzi o producenta w kontekście Kafki.

1.3. Confluent Platform

System zarządzania strumieniową transmisją zdarzeń (ang. Event Streaming Platform) dostarczany przez firmę Confluent. Bazuje na Apache Kafka i dostarcza szereg usprawnień pozwalających na łatwe zarządzanie i integrację z innymi narzędziami.

Nasza implementacja jest niezwiązana z tym narzędziem, ale używamy go do replikacji.

Rysunek 1.1: Architektura platformy Confluent



źródło: https://docs.confluent.io/current/platform.html

1.4. Scylla

Otwartoźródłowy system zarządzania bazą danych NoSQL napisany w C++. Stanowi alternatywę dla Apache Cassandra. Sami twórcy używają terminu "drop-in replacement", co oznacza, że można wymienić jedną bazę na drugą bez potrzeby rekonfiguracji lub z minimalnymi poprawkami.

1.5. Seastar

Framework o otwartym kodzie napisany w C++. Służy do pisania wydajnych aplikacji działających po stronie serwera w paradygmacie programowania asynchronicznego. Wykorzystywany jest w Scylli.

1.6. Produkt

Produktem, który zobowiązaliśmy się dostarczyć jest Kafka Producer API zaimplementowane w C++ jako osobna biblioteka zależna od frameworka Seastar. Dodatkowo dostarczamy Scyllowy serwis, który korzystając ze wspomnianego API replikuje dane z wybranych tabel na wskazany klaster Kafki. Poprawne działanie obu części wykazujemy poprzez skonfigurowanie środowiska, w którym wykorzystując wspomniane powyżej narzędzia, dokonujemy replikacji tabel z jednej instancji Scylli do drugiej, innej instancji, wykorzystując klaster Kafki jako pośrednika.

Rozdział 2

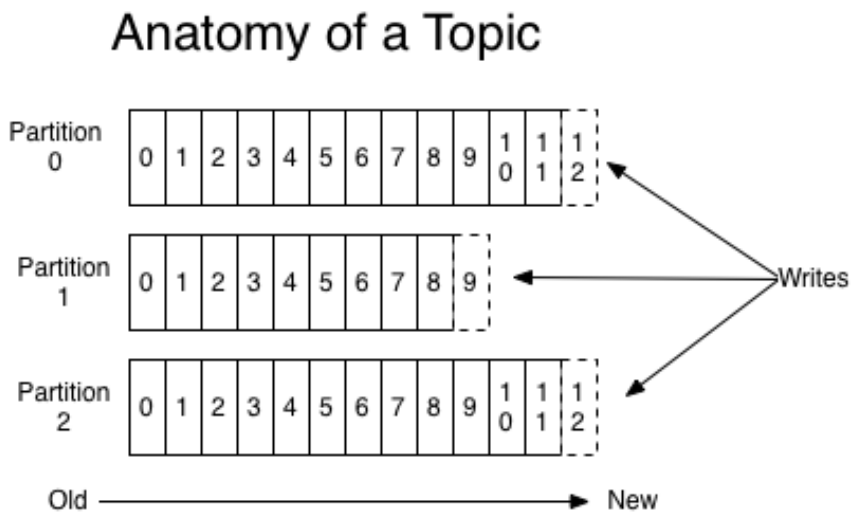
Producer API

Nasz produkt co prawda skupia się tylko na jednym API, ale warto najpierw nieco przybliżyć ogólny obraz działania Kafki.

2.1. Architektura Kafki

Wiadomości w Kafce są parami klucz-wartość. Każda wiadomość trzymana w klastrze jest przypisana do pewnego tematu (ang. topic). Tematy z kolei podzielone są na partycje – w zależności od przyjętej strategii partycjonowania, wiadomości trafiają na różne partycje na podstawie ich klucza. Przesunięcie (ang. offset) w obrębie pojedynczej partycji jednoznacznie identyfikuje konkretną wiadomość.

Rysunek 2.1: Struktura tematu (ang. topic) w Kafce



źródło: https://kafka.apache.org/intro

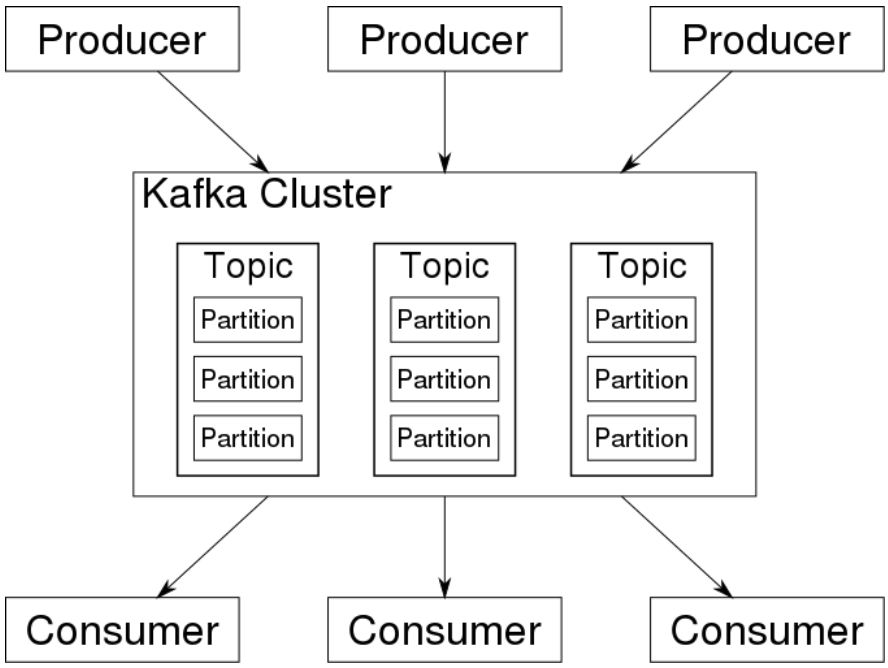
Klaster Kafki składa się z wielu serwerów zwanych brokerami. Partycje są przechowywane na brokerach i replikowane między nimi. Każda partycja może znajdować się na wielu bro-

kerach, a każdy broker trzyma partycje należące do różnych tematów. Partycje w Kafce są ważnym elementem, który pozwala na zrównoleglenie wykonywanych operacji.

Najważniesze interfejsy programowania aplikacji, które udostępnia Apache Kafka to:

- Producer API – pozwala aplikacjom publikować do tematów w klastrze rekordy danych. Aplikacje zajmujące się publikowaniem wiadomości nazywa się producentami.
- Consumer API – pozwala aplikacjom czytać wiadomości z tematów w klastrze. Takie aplikacje nazywamy konsumentami.
- Streams API – pozwala na manipulowanie strumieniami danych z tematów wejściowych do tematów wyjściowych.
- Connect API – pozwala na tworzenie aplikacji wypychających dane z zewnętrznego systemu do Kafki lub z Kafki do zewnętrznego systemu. Standaryzuje sposób integracji innych systemów przechowujących dane z Kafką.
- Admin API – pozwala na zarządzanie klastrem Kafki, w tym obserwowanie brokerów, partycji, etc.

Rysunek 2.2: Kluczowe pojęcia Apache Kafka



źródło: https://upload.wikimedia.org/wikipedia/commons/thumb/6/64/Overview_of_Apache_Kafka.svg/677px-Overview_of_Apache_Kafka.svg.png

2.2. Istniejące implementacje

Implementacji w różnych językach jest wiele (zob. [Url01]). Po krótce przybliżymy wybrane z nich.

2.2.1. KafkaProducer (Java)

Producent dostarczany w oficjalnej implementacji Kafki przez Apache. Napisany jest w Javie. Jako że jest dostarczany przez Apache, wspiera najnowsze funkcje dostępne w Producent API Kafki. Jego dodatkowym plusem jest wieloplatformowość.
Strona projektu: https://github.com/apache/kafka

2.2.2. librdkafka (C/C++)

Implementacja tworzona z myślą o wydajności i niezawodności. Według autorów obecna wersja producenta potrafi wysyłać ponad milion wiadomości na sekundę. Wspiera wiele Kafka Improvement Proposals oraz wersji brokerów.
Strona projektu: https://github.com/edenhill/librdkafka

2.3. Nasza implementacja

Nasz klient pisany był przede wszystkim w celu użycia go w serwisie replikującym. Z tego powodu istnieją funkcjonalności, których nie udostępniamy na ten moment. Tworząc własnego producenta zależało nam na wydajności rozwiązania, ponieważ to ona ogranicza wydajność całego serwisu. Na przesłanki jego efektywności składały się następujące rzeczy:

- napisanie go w języku C++ - jak w każdej implementacji w tym języku unikamy problemu ładowania aplikacji do JVM oraz mamy większą kontrolę nad pamięcią
- użycia frameworku Seastar i jego innowacji - szczegóły w podrozdziale
- bycia rozwiązaniem natywnym dla Scylli, posiadając w związku z tym najlepszą możliwą integrację z bazą, jak też pozbywając się pośredników w komunikacji między instancją macierzystą a klastrem Kafki

2.3.1. Seastar

Seastar jest frameworkiem napisanym w języku C++. Służy do pisania wydajnych aplikacji serwerowych. Niestety, ogranicza on działanie aplikacji do systemów Linux oraz OSv.
W zamian oferuje:

Seastar futures and promises - Są to rozwiązania należące do podzbioru programowania reaktywnego. Różnią się od swoich nazwowych odpowiedników z biblioteki standardowej lub biblioteki Boost. Służą do zarządzania nieblokującymi mikro-zadaniami.(zob. [Sea01])
W tym celu:

- nie wymagają blokowania zasobów
- nie alokują pamięci
- wspierają kontynuacje

Shared-nothing design - Seastar przydziela każdemu wątkowi aplikacji jeden rdzeń. Wszelka wymiana informacji musi zostać explicite zaimplementowana, zamiast polegać na pamięci współdzielonej pomiędzy wątkami (zob. [Sea02]).

Własna obsługa sieci - Korzystając z DPDK, aplikacje seastar'owe są w stanie uzyskać bezpośredni dostęp do interfejsów sieciowych, omijając przy tym obsługę pakietów przez jądro

systemu. Zamiast tego obsługuje je natychmiast stos technologiczny TCP/IP w przestrzeni użytkownika (zob. [Sea03]).

Własna komunikacja między-procesowa - Dla każdej pary różnych rdzeni procesora tworzone są dwie kolejki - jedna na spełnione żądania i jedna na pozostałe. Żądania realizowane są za pomocą Promises - unikamy w ten sposób blokowania zasobów.

2.3.2. Zaimplementowane funkcjonalności

Nasza implementacja wspiera następujące funkcjonalności:

- batching – wysyłanie wiadomości w grupach
- retries – ponowne wysłanie wiadomości w przypadku błędu
- różne poziomy ACK (acknowledgment) – oczekiwanie na potwierdzenie odbioru wiadomości przez ustaloną liczbę brokerów (brak potwierdzeń, potwierdzenie od lidera partycji, potwierdzenie od wszystkich replik)
- wersjonowanie protokołu – klient wspiera wiele wersji brokerów
- partycjonowanie – wiele strategii partycjonowania wiadomości
- multi-broker – obsługa klastrów Kafki składających się z wielu brokerów

Rozdział 3

Klient Kafki

Nasz klient Kafki jest biblioteką napisaną w języku C++, korzystającą z frameworka Seastar. Implementuje on Producer API.

3.1. Cele projektowe

Na początku projektu określiliśmy cele i priorytety przyświecające naszej implementacji klienta Kafki.

3.1.1. Możliwość użycia w bazie Scylla

Głównym celem własnej implementacji klienta Kafki była możliwość używania go z frameworkiem Seastar. Wynika to z faktu, że klient jest używany w procesie replikacji w bazie Scylla.

Baza danych Scylla w pełni korzysta z frameworka Seastar. Cała funkcjonalność bazy jest oparta o mikro-zadania bez wywłaszczania, zaimplementowane we frameworku. Sprawia to, że wykonywane mikro-zadania nie mogą blokująco czekać na operacje dyskowe lub sieciowe, co spowodowałoby wstrzymanie pracy na całym rdzeniu sprzętowym. Operacje te muszą korzystać z nieblokującego I/O (operacji wejścia/wyjścia) oferowanego przez framework Seastar.

3.1.2. Wysoka wydajność

Nasz klient Kafki jest wykorzystywany w procesie replikacji w bazie danych Scylla, która jest w stanie obsłużyć setki tysięcy operacji na sekundę w jednym węźle klastra (zob. [Scy01]). Sprawia to, że implementacja musi być wysokowydajna, na przykład poprzez:

- optymalizację gorącej ścieżki (ang. hot path) – metoda wysyłająca wiadomość na klaster może być używana setki tysięcy razy na sekundę
- batching – wysyłanie wiadomości w grupach, aby ograniczyć liczbę zapytań do klastra Kafki, liczbę pakietów sieciowych, narzut wielkości elementów protokołów (protokołu Kafki, TCP/IP)
- wydajne użycie frameworka Seastar oraz możliwości języka C++

3.1.3. Poprawność rozwiązania

Chcieliśmy, aby nasze rozwiązanie było jak najbardziej odporne na awarie, nie powodowało utraty danych lub krytycznego zatrzymania programu. Aby nasz klient spełniał ten cel, wprowadziliśmy następujące rozwiązania:

Obsługa błędów Kafki – Nasz klient interpretuje kody błędów zwracane przez brokerów Kafki i w odpowiedni sposób reaguje na nie, np. ponawiając zapytanie lub pytając klaster o metadane działających brokerów.

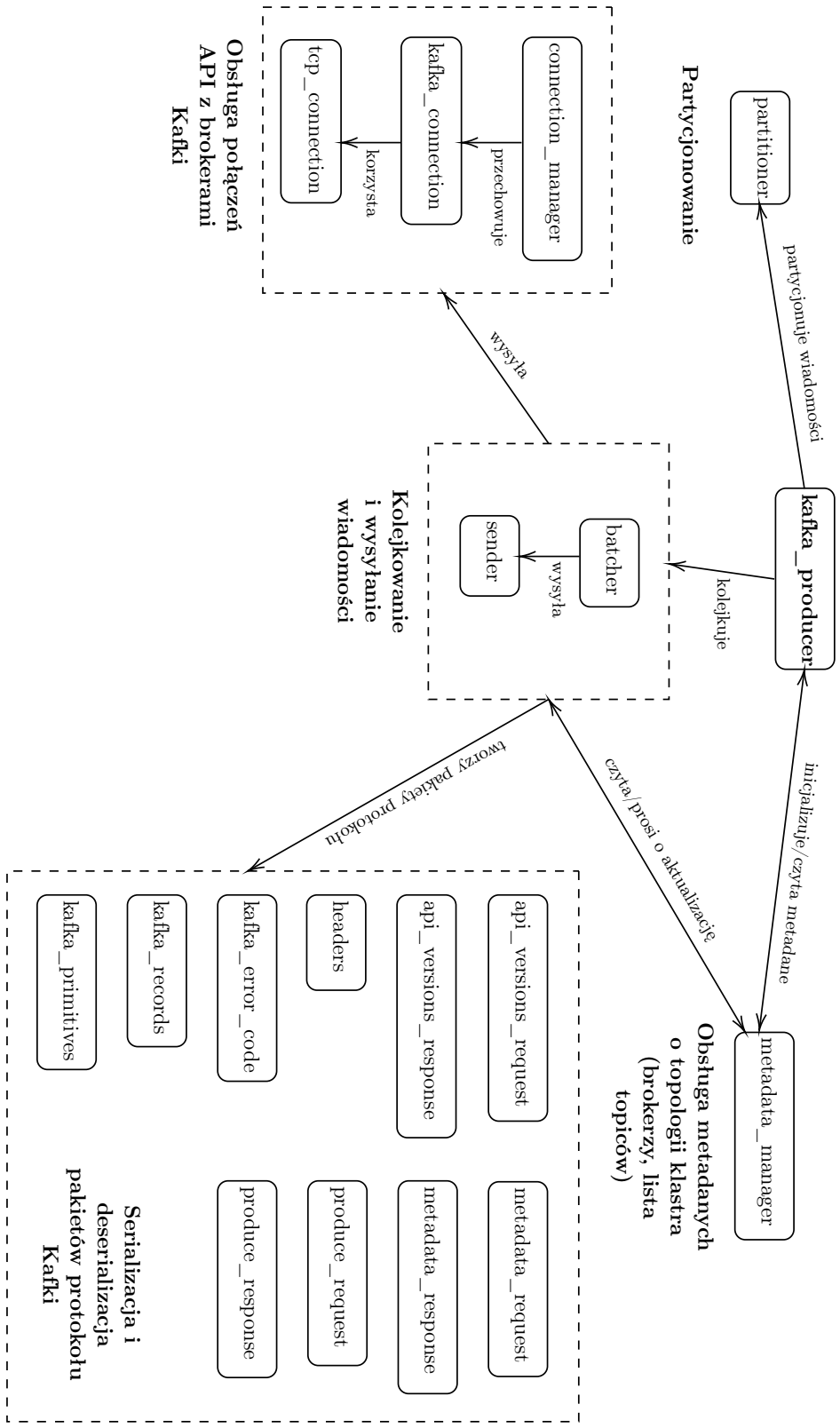
Obsługa błędów sieciowych – Nasz klient sprawdza poprawność wykonanych operacji sieciowych, wprowadza limity na czas nawiązania połączenia (ang. timeout) lub odbioru/wysłania danych i odpowiednio reaguje na zerwanie się połączenia.

Oparcie się na oficjalnym kliencie Java – Implementując naszego klienta, aby rozstrzygnąć poprawne zachowanie klienta na przykład w trakcie obsługi błędów, analizowaliśmy istniejącą implementację klienta Kafki w języku Java. Jest to oficjalny klient Kafki, rozwijany przy projekcie Apache Kafka.

Obsługa wielu wersji protokołu – API udostępniane przez brokerów Kafki jest wersjonowane. Przy nawiązywaniu połączenia z brokerem, nasz klient pyta się o wersje API brokera, a wszystkie zapytania wysyła w odpowiedniej wersji protokołu.

Wykonanie testów automatycznych – Aby sprawdzić poprawność klienta, zaimplementowaliśmy testy całościowe (ang. E2E – end-to-end). Korzystają one z oficjalnego konsumenta Kafki w języku Java i sprawdzają poprawność wiadomości wysłanych naszym producentem.

3.2. Architektura klienta



3.2.1. kafka_producer

Główna klasa naszego producenta. Klasa udostępnia następujące metody:

- `init()` – inicjalizacja producenta. Metoda nawiązuje początkowe połączenie z klastrem. Rozpoczyna się proces utrzymywania i odświeżania metadanych na temat klastra i listy tematów (ang. topics).
- `produce(sstring topic_name, sstring key, sstring value)` – wysłanie wiadomości do tematu Kafki. W zależności od ustawień producenta, wiadomość może zostać zakolejkowana do późniejszego grupowego wysłania (ang. batching).
- `flush()` – wymuszenie natychmiastowego wysłania zakolejkowanych wiadomości.
- `disconnect()` – rozłączenie się z klastrem Kafki i zakończenie pracy producenta.

Konfiguracja producenta jest przekazywana w argumencie konstruktora `kafka_producer` poprzez obiekt `producer_properties`. Opcje konfiguracji są wzorowane na oficjalnym producencie Kafki. `producer_properties` pozwala na zmianę następujących opcji producenta:

- `_servers` – początkowa lista brokerów Kafki; **obowiązkowy parametr**
- `_acks` – oczekiwanie na potwierdzenie odbioru wiadomości przez ustaloną liczbę brokerów:
  - `ack_policy::NONE` – brak oczekiwania na potwierdzenie
  - `ack_policy::LEADER` – oczekiwanie na potwierdzenie lidera partycji
  - `ack_policy::ALL` – oczekiwanie na potwierdzenie lidera partycji i wszystkich replik
- `_linger` – maksymalna liczba milisekund przez które wiadomość może czekać na wysłanie
- `_buffer_memory` – maksymalna sumaryczna długość w bajtach kluczy i wartości wiadomości oczekujących na wysłanie
- `_retries` – maksymalna liczba powtórzeń wysłania wiadomości (np. w przypadku błędu wysłania)
- `_request_timeout` – maksymalny czas oczekiwania na operację sieciową podany w milisekundach
- `_metadata_refresh` – liczba milisekund co którą będą odświeżane metadane na temat klastra Kafki i tematów
- `_client_id` – nazwa klienta; informacja ta jest przekazywana w zapytaniach do brokerów
- `_partitioner` – obiekt udostępniający logikę rozdzielania wiadomości do partycji
- `_retry_backoff_strategy` – funkcja określająca czas oczekiwania pomiędzy kolejnymi próbami wysłania wiadomości

Przykładowy kod podstawowego użycia producenta (wewnątrz Seastar thread):

Listing 3.1: Przykładowe użycie producenta

```
// Ustawienie opcji producenta.
kafka4seastar::producer_properties properties;
properties._client_id = "example";
properties._servers = {
    {"localhost", 9092}
};

// Stworzenie i zainicjowanie producenta.
kafka4seastar::kafka_producer producer(std::move(properties));
producer.init().wait();

// Nadanie "Hello World!" do "test-topic".
producer.produce("test-topic", "Wiadomosc", "Hello World!").wait();

// Koniec pracy z producentem.
producer.disconnect().wait();
```

Wszystkie metody producenta zwracają wynik jako `seastar::future`.

3.2.2. Kolejkowanie i wysyłanie wiadomości

Wiadomości wysłane metodą `kafka_producer::produce` są przekazywane do klasy `batcher`, gdzie oczekują na wysłanie.

`batcher` udostępnia metodę `flush()`, która wykonuje wysłanie oczekujących wiadomości. W tym celu korzysta z klasy `sender`, która zajmuje się podziałem wiadomości na brokerów, przygotowaniem oraz wysłaniem pakietów protokołu Kafki, jak również obsługą odpowiedzi o powodzeniu operacji.

`batcher` na podstawie informacji o powodzeniu operacji wykonanych w stworzonym `sender` wykonuje ponowne wysłanie tych wiadomości, które nie zostały poprawnie wysłane.

`batcher` może automatycznie wykonywać metodę `flush`. Dzieje się to:

- okresowo: co interwał czasu określony parametrem `_linger`
- w przypadku przepełnienia bufora – sumarycznej długości kluczy i wartości oczekujących wiadomości (parametr `_buffer_memory`)

3.2.3. Obsługa metadanych o topologii klastra

Nasz klient, poprzez klasę `metadata_manager`, utrzymuje aktualne metadane na temat klastra. Składają się one z listy brokerów (ID węzła, host, port) i listy tematów (nazwy i lista partycji).

Jako że klaster Kafki jest dynamiczny – mogą być dynamicznie dodawani lub usuwani brokerzy – ważne jest, aby utrzymywać stosunkowo aktualne metadane klastra. Gdyby metadane nie były odświeżane przez długi czas, stara lista mogłaby zawierać wyłącznie nieistniejących już brokerów.

Metadane listy tematów używane są przez klasę `sender` do ustalenia adresu sieciowego brokera, do którego będzie wysłana dana wiadomość. W przypadku otrzymania niektórych błędów po wysłaniu wiadomości, klasa `sender` może poprosić `metadata_manager` o odświeżenie metadanych. Przykładem takiego błędu jest `NOT_LEADER_FOR_PARTITION` mówiący o tym,

że wysłaliśmy wiadomość do brokera, który nie jest liderem partycji. Ponieważ wiadomości w naszym kodzie wysyłamy tylko do liderów partycji, oznacza to, że metadane w momencie wysłania nie były aktualne i konieczne jest ich odświeżenie.

3.2.4. Serializacja i deserializacja pakietów protokołu Kafki

Komunikacja między klientem a brokerami Kafki odbywa się używając binarnego protokołu.

W pliku `kafka_primitives` została zaimplementowana serializacja i deserializacja podstawowych typów danych, które są używane w protokole:

- `kafka_number_t` – liczba całkowita. Dostępne są różne warianty, m. in. `kafka_int8_t` (`kafka_number_t<int8_t>`), `kafka_int64_t` (`kafka_number_t<int64_t>`)
- `kafka_varint_t` – liczba całkowita zapisana w formacie zmiennej długości z kodowaniem zig-zag; wzorowana typem varint w Protocol Buffers (zob. [Url02])
- `kafka_buffer_t` i `kafka_nullable_buffer_t` – (nullowalny) ciąg bajtów; używany w dwóch wariantach: ciąg znaków lub ciąg bajtów
- `kafka_array_t` – nullowalna tablica elementów wyznaczonego typu

Powyższe podstawowe typy danych zostały zaimplementowane jako obiekty C++, udostępniające metody `serialize` i `deserialize`.

Zaimplementowaliśmy również obiekty reprezentujące następujące zapytania (i odpowiedzi):

- ApiVersions – zapytanie o wspierane typy zapytań i wspierane wersje protokołu
- Metadata – zapytanie o metadane klastra
- Produce – zapytanie o dodanie nowej wiadomości do partycji tematu Kafki

Każde z zapytań jest wersjonowane oddzielnie. Wersja zapytania jest liczbą naturalną. Nasz klient (na podstawie odpowiedzi `ApiVersions`) wysyła i parsuje odpowiedni wariant danego pakietu protokołu.

Kod deserializacji sprawdza poprawność pakietu, a w przypadku błędu rzuca odpowiedni wyjątek.

3.2.5. Partycjonowanie

Wyboru strategii partycjonowania dokonuje się przekazując `producer_properties` odpowiedni obiekt dziedziczący po klasie `partitioner`. Taki obiekt definiuje strategię partycjonowania, którą producent będzie stosować.

Domyślnie udostępniamy dwie strategie partycjonowania:

`basic_partitioner` - Ten partitioner definiuje bardzo prymitywną strategię. Pytany o partycję zawsze zwróci pewną pseudolosową. Nie dajemy żadnych gwarancji co do jakości generatora liczb losowych.

`rr_partitioner` - Round robin partitioner. W przypadku przekazania pustego klucza w wiadomości, będzie je rozmieszczać zgodnie z algorytmem karuzelowym. W ten sposób osiąga równomierne rozłożenie zapisów pomiędzy partycje. Dla niepustych kluczy wybiera partycję na podstawie ich hasza (`std::hash`).

Klasa `partitioner` została zrealizowana jako `ABC` (Abstract Base Class), aby zapewnić możliwość implementacji kolejnych strategii w przyszłości, np. nowe strategie w oficjalnym kliencie Java. Potrzeba przechowywania stanu w `partitioner` nie pozwala tutaj użyć `std::function`.

3.2.6. Obsługa połączeń z brokerami

Połączenia są obsługiwane przez klasy: `tcp_connection`, `kafka_connection`, obsługujące pojedyncze połączenie oraz `connection_manager` zarządzające grupą połączeń. Każda kolejna klasa jest wyżej w hierarchii abstrakcji od poprzedniej i razem tworzą warstwową strukturę.

`tcp_connection` – klasa zarządzająca połączeniami TCP. Jest lekkim opakowaniem dostępnych funkcji we frameworku Seastar, implementując przy tym dodatkową funkcjonalność, taką jak limit czasu na operacje sieciowe połączenia.

`kafka_connection` – wysokopoziomowa klasa opakowująca klasę `tcp_connection`. Pozwala na wysłanie wysokopozimowych obiektów protokołu Kafki (opisane w podrozdziale Serializacja i deserializacja pakietów protokołu Kafki). Przy wysłaniu automatycznie serializuje obiekt, wysyła go poprzez połączenie TCP, następnie oczekuje na odpowiedź i ją deserializuje do obiektu odpowiedzi protokołu Kafki.

`connection_manager` – klasa zarządzająca zbiorem połączeń typu `kafka_connection`. Udostępnia wysokopoziomową metodę `send(RequestType request, const sstring host, uint16_t port, uint32_t timeout, bool with_response)`, która automatycznie tworzy połączenie `kafka_connection` (jeśli jeszcze go nie nawiązaliśmy), wysyła obiekt pakietu Kafki i zwraca obiekt odpowiedzi protokołu Kafki. W przypadku błędów połączenia, zamyka wadliwe połączenie i próbuje go nawiązać ponownie przy wysłaniu kolejnego pakietu.

Rozdział 4

Bazy danych

> *[...], system bazodanowy to po prostu komputerowy system przechowywania rekordów: innymi słowy, jest to system komputerowy, którego ogólnym przeznaczeniem jest przechowywanie informacji i umożliwienie użytkownikom wydobywania oraz uaktualniania owych informacji na życzenie. Wspomnianą informacją może być wszystko o jakimkolwiek znaczeniu dla jednostek, lub też organizacji związanych z systemem - to jest, wszystko co jest pomocne w procesie prowadzenia ich biznesu.* [Book01]

Niezmiennie od lat, bazy danych są integralną częścią systemów informatycznych w każdej niemal skali. Szybki postęp technologiczny, rodzący potrzebę przetwarzania gigantycznych ilości informacji, wymusił również dynamiczną ewolucję baz - w systemy rozbudowane, implementujące istotnie różne między sobą zestawy rozwiązań. W tym rozdziale przybliżymy pokrótce jeden z ogólniejszych podziałów baz danych oraz sklasyfikujemy Scyllę i jej cechy charakterystyczne. Wskażemy także trudności wpływające na rozwiązywany przez nas problem replikacji, wynikające ze specyfikacji tych systemów.

4.1. Podział baz danych

Istnieje wiele płaszczyzn klasyfikacji baz danych, rozróżniających po zarówno większych jak i mniejszych cechach. Przedstawiamy jeden z bardziej ogólnych podziałów - ze względu na sposób trzymania danych - a resztę cech przybliżymy już tylko w zakresie ich implementacji przez Scyllę.

4.1.1. Model relacyjny

Jest to najbardziej standardowy model baz danych, opierający się na reprezentacji zbiorów informacji w formie relacji - tabel składających się z wierszy oraz kolumn, gdzie każdy wiersz (zwany inaczej rekordem) posiada swój identyfikujący klucz. Taki model sprawdza się najlepiej w opisywaniu dobrze ustrukturyzowanych zestawów danych, zapewniając łatwość w definiowaniu powiązań między tabelami.

4.1.2. Model NoSQL

Model nierelacyjny, zwany często modelem NoSQL (z powodu używania innych niż SQL języków zapytań), reprezentuje dane w sposób nietablicowy. Dokładna struktura, przy pomocy

której jest to osiągnięte, zależy od konkretnego produktu. Modele te, zyskujące znacznie na popularności w ostatniej dekadzie, oferują wysoką elastyczność oraz pozwalają w łatwy sposób współdzielić dane w klastrze maszynowym (grupa połączonych ze sobą maszyn, np. serwerów). Bazy NoSQL wyszły naprzeciw potrzebie rynkowej stworzonej przez dynamiczne aplikacje operujące na dużych zbiorach danych o niskim poziomie strukturyzacji (np. komunikatory internetowe).

4.2. ScyllaDB

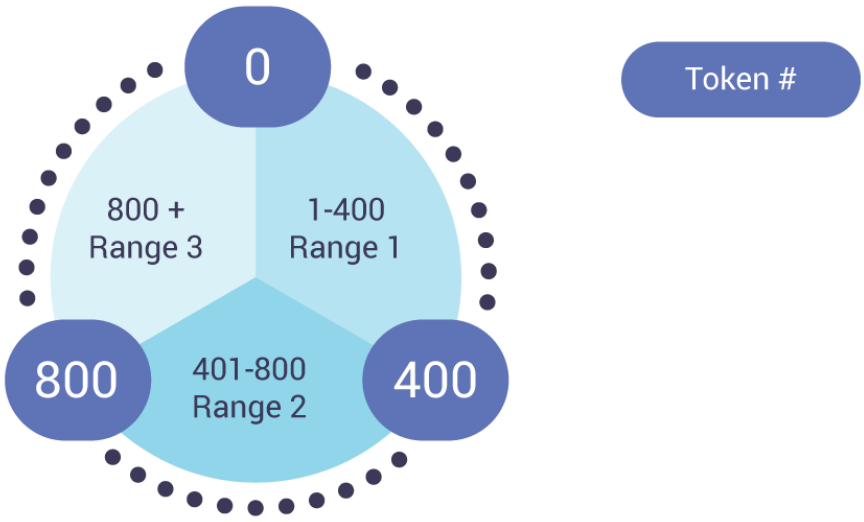
Na początek warto nadmienić, że Scylla jest bazą danych opartą w znacznym stopniu na bazie Apache Cassandra, w formie tzw. drop-in replacement. Oznacza to, że jedna z nich może zostać podmieniona na drugą przy tylko minimalnych, lub nawet zerowych, zmianach. W związku z tym, duża część rozwiązań jest wspólna dla obu produktów, a jeśli któraś funkcjonalność nie zostanie opisana, należy założyć, że działa tak samo jak w Cassandrze.

4.2.1. NoSQL

Scylla jest bazą nierelacyjną o charakterze szerokokolumnowym, implementującą architekturę opartą o węzły wirtualne (ang. VNode-oriented). Jej krótki opis wygląda następująco:

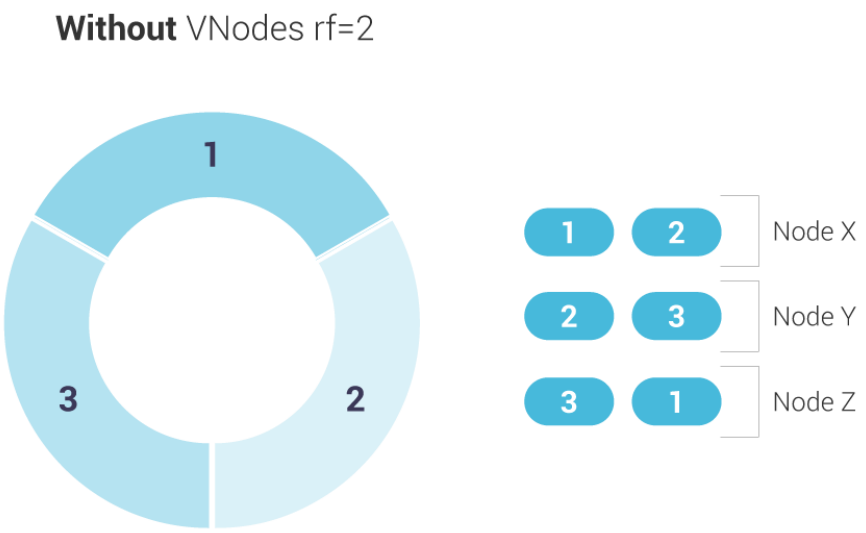
- Klastry składają się ze zbioru fizycznych węzłów
- Węzły fizyczne nie współdzielą żadnych danych, a jedynie komunikują się ze sobą jeśli trzeba (ang. shared-nothing approach)
- Każdy węzeł fizyczny zawiera szereg partycji, będących jednostkami danych w Scylli
- Klaster ma formę pierścienia podzielonego na ciągły zakres tokenów
- Partycja jest unikatowo identyfikowana przez swój klucz partycyjny, reprezentowany w formie tokena obliczanego na podstawie klucza głównego
- Klucze klastrujące mogą zostać wyszczególnione w celu utrzymania porządku wewnątrz danej partycji

Rysunek 4.1: Przestrzeń tokenów klastra podzielona między trzy węzły fizyczne [Scy03]



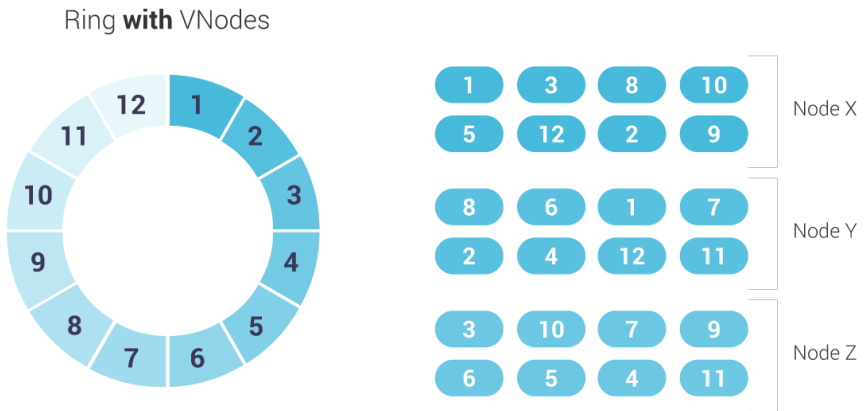
- Wewnątrz jednej instancji (klastra) dane mogą być replikowane n-krotnie, tzn. każda partycja będzie przetrzymywana łącznie na n węzłach

Rysunek 4.2: Rozłożenie tokenów na klastrze z trzema węzłami i dwukrotną replikacją [Scy03]



Dopiero na powyższe nakładana jest ostatnia warstwa abstrakcji, czyli węzły wirtualne. Węzły te reprezentują pewien ciągły przedział tokenów leżący w całości na danym węźle fizycznym. Następnie, każdemu węzłowi fizycznemu przypisywana jest odpowiednia liczba węzłów wirtualnych.

Rysunek 4.3: Klaster z dwukrotną replikacją oraz dwunastoma węzłami wirtualnymi - po cztery na każdy węzeł fizyczny [Scy03]



4.2.2. Przestrzenie nazw

Przestrzenie nazw reprezentują zbiory tabel o jednakowej dla całego zbioru krotności replikacji. Przestrzeń taka odwzorowuje bazę danych w rozumieniu języka SQL.

4.2.3. CQL

Podobnie jak Cassandra, Scylla korzysta z języka zapytań CQL.

4.2.4. Change Data Capture

CDC jest funkcjonalnością pozwalającą na odpytywanie bazy danych o historię zmian w danej tabeli. Po włączeniu tej opcji dla tabeli `ks.t` (gdzie `ks` jest nazwą przestrzeni nazw), tworzona jest równocześnie specjalna tabela `ks.t_scylla_cdc_log`. To do niej zapisywane są wszystkie zachodzące mutacje w postaci wierszy różnicowych, wzbogacone o dodatkowe metadane. W skład rekordu wchodzą:

- `cdc$stream_id` - klucz partycyjny wiersza, generowany na podstawie klucza oryginalnej mutacji oraz obecnej generacji CDC
- `cdc$time` - pierwsza część klucza klastrującego, składająca się ze znacznika czasu oryginalnej operacji sklejonego z ciągiem losowych bajtów pozwalających na rozróżnienie dwóch operacji wykonanych w tym samym czasie (o tym samym znaczniku czasu operacji)
- `cdc$batch_seq_no` - druga część klucza klastrującego, służy do grupowania i rozróżniania mutacji przeprowadzonych przy użyciu jednego zapisu (a więc z dokładnie tym samym polem cdc$time)
- `cdc$operation` - przechowuje typ mutacji, szczegóły poniżej
- `cdc$ttl` - reprezentuje czas życia danej mutacji zaaplikowanej na żywych kolumnach
- `cdc$deleted_{p}` - przechowywana dla każdej z kolumn nie należących do klucza głównego wartość logiczna informująca, czy kolumna `p` została w tej operacji usunięta
- `kolumny oryginalnej tabeli` - kolumny odpowiadające kolumnom ze śledzonej tabeli, przechowujące informacje na temat wartości ustawionych w danej kolumnie podczas tej mutacji

Dziennik CDC rozróżnia 10 typów mutacji, z czego 8 z nich jest wierszami różnicowymi:

0	zdjęcie stanu ‘sprzed’
1	aktualizacja wiersza
2	wstawienie wiersza
3	usunięcie wiersza
4	usunięcie partycji
5	lewostronnie domknięte przedziałowe usunięcie wierszy
6	lewostronnie otwarte przedziałowe usunięcie wierszy
7	prawostronnie domknięte przedziałowe usunięcie wierszy
8	lewostronnie otwarte przedziałowe usunięcie wierszy
9	zdjęcie stanu ‘po’

Ponadto, wiersze tabel CDC mają swój własny czas życia (parametr TTL), utrudniający całkowite zaśmiecenie bazy poprzez te dzienniki.

4.2.5. Seastar

Całość Scylli została napisana w oparciu o framework Seastar, naturalnie więc wszystkie jego cechy opisane w poprzednich rozdziałach aplikują się również do jej kodu. Seastar stanowi tak integralną i kluczową jej część, że nawet sam CTO ScyllaDB, Avi Kivity, porównywał Seastara do swoistego systemu operacyjnego dla tej bazy danych.

4.3. Wynikające problemy

Ze względu na bycie ogromnymi, dynamicznymi systemami, prace z bazami danych prezentują wiele wyzwań. Nawiązując do przytoczonego na początku tego rozdziału cytatu, informacjami trzymanymi w bazach może być "wszystko". W rzeczy samej, współczesne bazy danych obsługują wiele rozbudowanych formatów danych, a rozmiar przechowywanych danych dawno przestał być liczony w gigabajtach.

W parze z problemami rozmiaru baz idzie nierozłączny problem przepustowości. Implementując dodatkowy serwis wewnątrz systemu bazodanowego należy upewnić się, że jest on w stanie poprawnie funkcjonować i nadążać za przetwarzanymi danymi, nie spowalniając jednocześnie całości systemu. Wreszcie, skala niesie za sobą również zwiększone wykorzystanie sprzętu fizycznego, a ten - ryzyko zawodności i potrzebę obsługiwania takich sytuacji.

Rozdział 5

Replikacja baz danych

> *[...], Replikacja baz danych jest powszechnie używana ze względu na lepszą odporność na awarie, skalowalność i wydajność. Awaria pojedynczej repliki bazy nie zatrzymuje działania całego systemu bazodanowego, ponieważ inne repliki mogą przejąć zadania nieczynnej bazy. Skalowalność może być uzyskana poprzez rozproszenie zapytań między wszystkie repliki i dodawanie nowych, gdy zwiększa się obciążenie systemu. Replikacja baz danych może umożliwić szybki lokalny dostęp do danych, nawet gdy klienci są w różnych rejonach geograficznych, jeżeli kopie danych są zlokalizowane blisko klientów.* [Book02]

Replikacja baz danych to proces polegający na powieleniu danych między wieloma instancjami baz danych. Mimo atrakcyjnych zalet (odporność na awarie, skalowalność i wydajność), może być ona trudna w efektywnej implementacji i utrzymaniu. Przy wyborze zastosowanej replikacji należy się zastanowić nad tym jak będą obsługiwane konflikty zapisów oraz awarie liderów lub replik.

5.1. Typy replikacji

5.1.1. Replikacja z pojedynczym liderem

Replikacja z pojedynczym liderem to wariant replikacji baz danych, w którym wszystkie zapisy trafiają do jednego lidera i są replikowane od niego.

W tym typie replikacji nie występują potencjalne konflikty, ponieważ zapisy są obsługiwane przez jeden węzeł.

W przypadku awarii repliki, należy uruchomić nową bazę danych i ustawić ją jako nową replikę lub naprawić popsutą bazę oraz restartować proces replikacji do niej. Należy się upewnić, że proces replikacji uwzględni wszystkie zmiany, które wystąpiły przed oraz w trakcie awarii, a nie tylko zmiany po ponownym restarcie procesu.

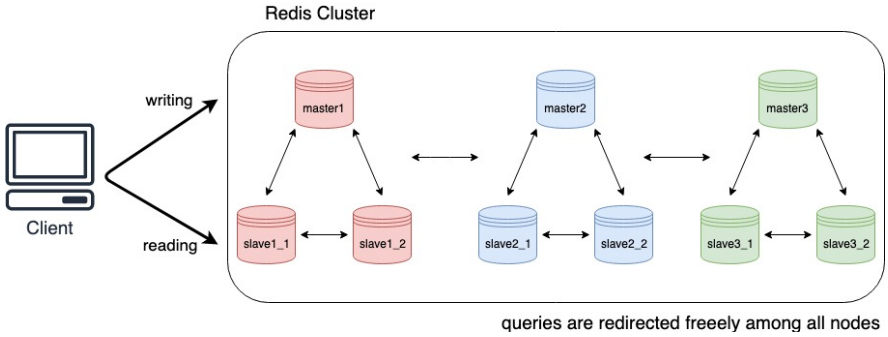
W przypadku awarii lidera możliwe jest szybkie przywrócenie poprawnego działania systemu bazodanowego, poprzez mianowanie pewnej repliki nowym liderem. W przypadku zapytań czytania z bazy nie jest nawet potrzebne mianowanie nowego lidera, gdyż ruch może zostać automatycznie przekierowany do jednej z replik.

W przypadku zastosowań charakteryzujących się dużą liczbą odczytów, możliwe jest bardzo łatwe skalowanie poprzez dodawanie nowych replik z których będą odczytywane dane.

Tego typu replikacja może być używana do skrócenia opóźnień odczytów w dalszych rejonach geograficznych, poprzez postawienie nowej repliki bliżej klientów.

Przykładem systemu bazodanowego korzystającego z replikacji z pojedynczym liderem jest Redis Cluster (zob. [Url03]). Dzieli on przestrzeń kluczy na partycje. Dla każdej partycji tworzy on "mini-klaster", składający się z lidera oraz replik. System aktywnie zarządza mini-klastrami, odpowiednio ustalając nowego lidera w przypadku awarii. Operacje są kierowane do lidera (zapisy) i replik (odczyt) odpowiednich dla danej partycji.

Rysunek 5.1: Diagram architektury Redis Cluster



źródło: https://engineering.grab.com/preventing-pipeline-calls-from-crashing-redis-clusters

5.1.2. Replikacja z wieloma liderami

Replikacja z wieloma liderami to wariant replikacji baz danych, w którym zapisy trafiają do wielu węzłów.

Pierwszy rodzaj tej replikacji zakłada, że będą istniały co najmniej dwa węzły nazywane liderami, do których klienci będą wysyłać operacje zapisu.

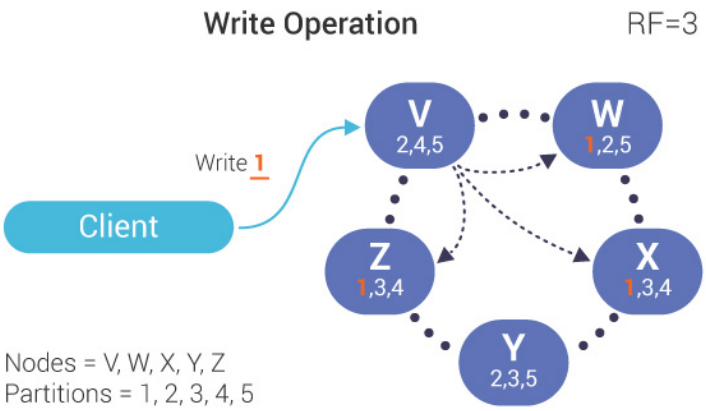
W drugim rodzaju tej replikacji zapisy są możliwe do wszystkich węzłów i nie jest używany koncept lidera.

W replikacji z wieloma liderami występuje problem potencjalnych konfliktów zapisów. Wymagane jest opracowanie strategii obsługi tego typu przypadków (na przykład ostatni zapis według czasu zegarowego jest wybierany).

Tego typu replikacja może być używana w systemach bazodanowych działających w wielu regionach geograficznych. W odróżnieniu od replikacji z pojedynczym liderem, możliwe jest uruchomienie lidera w każdym z rejonów geograficznych. Dzięki temu będą możliwe zarówno szybkie odczyty i szybkie zapisy w obu miejscach.

Przykładowym systemem bazodanowym implementującym replikację z wieloma liderami jest Scylla. Baza danych oferuje konfigurowalną liczbę kopii danych. Wszystkie węzły klastra są równe – nie ma lidera. Każdy węzeł przechowuje pewien zbiór partycji tabeli. Gdy zostaje wysłany zapis do węzła, komunikuje się on z innymi węzłami, które przechowują daną partycję i replikuje do nich zapis. Możliwe jest skonfigurowanie liczby potwierdzonych zapisów na które będzie czekał klient.

Rysunek 5.2: Operacja zapisu w bazie Scylla



źródło: https://docs.scylladb.com/architecture/architecture-fault-tolerance/

Rozdział 6

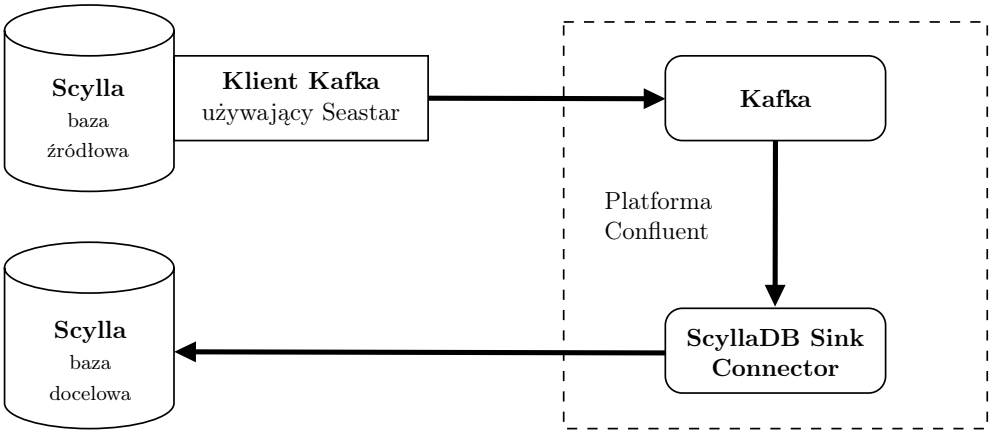
Serwis replikujący

Serwis replikujący jest drugą częścią zlecenia. Został on zaimplementowany jako dodatkowa funkcjonalność w Scylli. W tym rozdziale tłumaczymy sposób jego działania, narzędzia z których korzysta oraz mechanizmy których dotyka. Warto przypomnieć sobie architekturę Scylli (Rozdział 4.2).

6.1. Schemat działania

Działanie serwisu polega na okresowym odczytywaniu mutacji zawartych w historii operacji śledzonej tabeli i wysyłaniu ich na klaster Kafki używając naszego producenta. Jedna tabela w Scylli będzie odpowiadać dokładnie jednemu tematowi w klastrze. Po dotarciu, mutacje mogą być czytane przez dowolnego konsumenta, jednakże format w jakim są wysyłane został ustalony z myślą o ScyllaDB Sink Connectorze, który wyśle je dalej do innej instancji bazy danych ScyllaDB, gdzie zostaną zaaplikowane.

Rysunek 6.1: Architektura procesu replikacji



6.1.1. Potwierdzenie replikacji

Implementacja będzie uruchamiana na każdym węźle bazy danych. Każda instancja będzie odpowiadać za swój węzeł. Powoduje to, że kontrola poprawnej transmisji danych jest o wiele

prostsza. Nie ma potrzeby komunikacji z innymi węzłami i dochodzenia do konsensusu w systemie rozproszonym. Jest to duży plus pomimo redundancji, jaką wprowadza ta decyzja. Potwierdzenie przesłania do klastra Kafki zostaje wówczas wyłącznie w gestii producenta z którego korzystamy.

6.1.2. Śledzenie zmian i jego ograniczenia

Podczas działania utrzymywać będziemy wskaźnik na najstarszą, jeszcze niezreplikowaną mutację. Producent będzie próbował sekwencyjnie i ciągle wysyłać kolejne dane, w porządku chronologicznym. Wskaźnik przesuwamy dopiero po potwierdzonym przesłaniu zmiany. Scylla sama dba aby w historii operacji znalazły się jedynie te zmiany na tabelach, które już zaszły. Mutacje te muszą być trzymane aż do momentu potwierdzenia replikacji przez serwis. Możliwym sposobem zagwarantowania tego jest dopasowanie parametru `TTL` w tabelach CDC, aby jej wiersze nie zostały wymazane przed zreplikowaniem.

Ta strategia działania stwarza jednak pewne ryzyko. Istnieje możliwość, że nie będziemy nadążać z napływem danych, a będziemy wymuszać przetrzymywanie niezreplikowanych mutacji. Tak naprawdę niewystarczająca wydajność może zawsze wystąpić niezależnie od wybranej strategii. Z tego powodu serwis dopuszcza możliwość odrzucenia pewnych mutacji w zamian za pozbycie się potrzeby kosztownej synchronizacji producenta z dziennikiem mutacji, a także nadmiernie długiego trzymania wpisów w CDC. W najgorszym przypadku pewne tabele będą miały braki.

6.1.3. Gwarancja zgodności

Decyzja projektowa, w wyniku której uruchamiamy serwis na każdym węźle, w prosty sposób zwiększa prawdopodobieństwo zgodności replikowanych danych. Wiemy, że wszelkie mutacje znajdujące się na danym węźle prędzej czy później zostaną zreplikowane.

Istnieje zawsze ryzyko ulegnięcia awarii - w tym wypadku ubezpieczają nas wewnętrzne, Scyllowe repliki. Zawsze możemy liczyć na to, że mutacje które właśnie utraciliśmy zostaną przesłane przez któryś z węzłów zawierających replikę tych danych. Sytuacja w której awarii ulega wiele węzłów Scylli w taki sposób, że tracimy wszystkie repliki pewnych danych jest sytuacją skrajną.

Aby mówić o gwarancji poprawności poprzez repliki musimy jednak rozważyć gwarancję pojedynczego węzła. Dbamy o nią używając Seastarowego paradygmatu obietnic - aktualizujemy wskaźnik na ostatnio zreplikowany wiersz w tabeli CDC dopiero kiedy producent otrzyma potwierdzenie od klastra Kafki. Gwarancja niezaburzonej kolejności jest również łatwo wzmocniona poprzez modelowanie rozwiązania kontynuacjami.

6.1.4. Duplikaty

Gdy mamy do czynienia w Scylli z przestrzenią kluczy o współczynniku replikacji (na przykład) 3, moglibyśmy się obawiać, że replikacja spowoduje nadmierne powielenie mutacji, przez co w wyniku nie dostaniemy takiej tabeli, jakiej oczekiwaliśmy. Nie jest to jednak przeszkodą ze względu na idempotentność Scylli. Złożenie parę razy dokładnie tych samych operacji nie spowoduje dodatkowych zmian. W tej sytuacji, większy współczynnik replikacji zwiększa jedynie dostępność i odporność na awarię.

6.1.5. Zachowanie kolejności

Nawet jeśli niektóre operacje zostaną wykonane wielokrotnie, to cały ich zestaw z jednego węzła zostanie wykonany w dokładnie tej samej kolejności, co na każdym innym. W rezultacie stan końcowy nie różni się od tego, w którym aplikowalibyśmy konkretne operacje tylko raz.

6.1.6. Wydajność

W przedstawionym rozwiązaniu nie ma żadnej potrzeby tworzenia nowych struktur, ani pośredniego magazynowania danych. Nie dokonujemy żadnych dodatkowych zapisów, a wykorzystywane zasoby sieciowe i obliczeniowe są minimalne. Wąskim gardłem serwisu jest właśnie moduł wysyłający i komunikacja sieciowa, innymi słowy - producent. Zaimplementowanie go od zera wykorzystując Seastara pozwala nam znacznie zwiększyć jego wydajność i przepustowość, czerpiąc znaczące korzyści z jego funkcjonalności TCP/IP opartych o DPDK.

6.2. Narzędzia

Do replikacji tabel do osobnej instancji ScyllaDB korzystamy z szeregu narzędzi oraz struktur udostępnianych przez same systemy zawierające replikowane dane. Apache Kafka została już wystarczająco przybliżona w poprzednich rozdziałach. Jest istotnie ważną częścią projektu, ale nie jedyną.

6.2.1. Apache Avro

Apache Avro to otwartoźródłowy framework do serializacji danych. Wykorzystujemy go do pakowania wierszy w zwięzłą, binarną formę przed wysłaniem. Musi ona zgadzać się ze wcześniej ustalonym schematem danych. Klient próbujący odczytać binarne dane powinien wcześniej zostać poinformowany jak ich schemat wygląda.

Dlaczego Avro

Działanie z myślą o użyciu konektora od początku mocno ograniczało wybór. W praktyce jedynymi opcjami były JSON i Avro.
Do wyboru Avro przekonało nas:

- Binarne kodowanie – wiadomości w Avro można wysyłać zakodowane w binarnej, bezszkieletowej formie. Pozwala to pozbyć się z samej wiadomości niepotrzebnego opisu o strukturze i typie danych. Zamiast tego wysyłamy prawie same dane. Jest to korzyść której nie osiągamy używając JSONa.
- Bogate schematy danych – schematy, których używa się by opisać strukturę wiadomości mają bogate możliwości. Tworzy się je w czystym JSONie. Pozwalają na rozszerzanie schematów danych z zachowaniem kompatybilności z konsumentami, którzy mogą korzystać ze starych schematów. Jest to idealny wybór dla bazy danych w której schemat tabel również może ewoluować. Schematy ułatwiają integrację i chronią przed zniekształconymi pakietami.
- Kompresja – binarne Avro jest już samo w sobie minimalnie skompresowane. Typy całkowitoliczbowe są poddawane kodowaniu zig-zag. Całość w zależności od życzenia użytkownika może zostać skompresowana algorytmem `DEFLATE`. Niektóre biblioteki Avro udostępniają także `snappy`.

- Wydajność – serializacja i deserializcja wiadomości Avro jest bardzo wydajna (zob. [Url04]).

6.2.2. ScyllaDB Sink Connector

ScyllaDB Sink Connector pozwala na replikowanie wiadomości z Kafki do scyllowych tabel. Wspomnieliśmy wcześniej, że wiadomości w Avro muszą spełniać odpowiedni schemat. Nasz serwis uzależnia ten schemat nie tylko od schematu monitorowanej tabeli, ale także od formatu oczekiwanego przez ScyllaDB Sink Connector. Jest to istotne dla dalszej replikacji. Po przesłaniu wiadomości na Kafkę, możemy wówczas korzystając z niego przesłać dane do nowej instancji Scylli.

Używanie dostarczonego konektora niesie ze sobą niestety pewne ograniczenia w funkcjonalności. ScyllaDB Sink Connector obsługuje tylko dwa typy operacji: `INSERT` oraz `DELETE`. Z tego względu nasz serwis na chwilę obecną replikuje tylko mutacje tych dwóch typów oraz `UPDATE` - nie jest ona jednak odwzorowywana w 100%, ponieważ nasz serwis zmuszony jest tłumaczyć tę operację na operację `INSERT`.

Rozdział 7

Weryfikacja

7.1. Klient Kafki

7.1.1. Środowisko testowe

Aby wygodnie i efektywnie testować naszego klienta Kafki, na przykład w trakcie jego tworzenia, napisaliśmy skrypty umożliwiające łatwe stworzenie środowiska testowego.

Głównym komponentem środowiska jest lokalnie uruchomiony klaster Kafki. Brokery klastra Kafki są uruchamiane jako kontenery Docker.

Konfiguracja środowiska testowego jest opisana w Hashicorp Configuration Language - języku narzędzia Terraform. Narzędzie to jest używane przez nas do automatycznego tworzenia i usuwania lokalnej instancji naszego środowiska. Stworzona konfiguracja pozwala na łatwe ustalenie oczekiwanego adresu IP klastra oraz liczby brokerów Kafki.

Terraform jest używany przez nas do opisu tworzonych kontenerów Docker, ich konfiguracji (zmienne środowiskowe używane przez Kafkę), tworzonej sieci Docker (typu bridge) oraz instancji Zookeeper (zarządzającej klastrem).

7.1.2. Testy automatyczne

Zaimplementowaliśmy testy całościowe (ang. E2E – end-to-end). Zaimplementowane są one jako skrypty w języku Bash. Korzystając z narzędzia Terraform, tworzą one tymczasowe środowisko testowe. Następnie uruchamiają naszego producenta Kafki oraz oficjalnego (konsolowego) konsumenta Kafki i sprawdzają, czy konsument otrzymał wszystkie wiadomości od producenta.

Stworzone testy sprawdzają poprawność w kilku scenariuszach:

- `test_producer1.sh` – test tworzący klaster Kafki złożonony z jednego brokera. Producent wysyła trzy testowe wiadomości.
- `test_producer2.sh` – test tworzący klaster Kafki złożonony z trzech brokerów. Tworzone są dwa tematy z trzema partycjami i trzema replikami. Producent wysyła trzy testowe wiadomości do pierwszego tematu oraz jedną testową wiadomość do drugiego tematu.

- `test_producer3.sh` – test tworzący klaster Kafki złożonony z trzech brokerów. Tworzone są dwa tematy z trzema partycjami i trzema replikami. Producent wysyła po sto wiadomości do każdego z tematów.
- `test_producer4.sh` – test tworzący klaster Kafki złożonony z trzech brokerów. Tworzone są dwa tematy z trzema partycjami i trzema replikami. Producent wysyła po dziesięć wiadomości do każdego z tematów, a następnie jeden z brokerów jest wyłączany (inny niż ten z którym się początkowo połączyliśmy). Następnie producent wysyła po dziesięć wiadomości do każdego z tematów.
- `test_producer5.sh` – test tworzący klaster Kafki złożonony z trzech brokerów. Tworzone są dwa tematy z trzema partycjami i trzema replikami. Producent wysyła po dziesięć wiadomości do każdego z tematów, a następnie jeden z brokerów jest wyłączany (ten z którym się początkowo połączyliśmy). Następnie producent wysyła po dziesięć wiadomości do każdego z tematów.

Po skończonym teście sprawdzana jest poprawność otrzymanych danych przez konsumenta i wypisywana jest informacja na temat poprawności działania testu.

7.1.3. Testy jednostkowe

Część z funkcjonalności klienta Kafki jest testowana w formie testów jednostkowych. Zostały one napisane korzystając z biblioteki Boost.Test oraz pomocniczych funkcji frameworka Seastar.

Główną testowaną jednostkowo funkcjonalnością jest serializacja i deserializacja pakietów protokołu Kafki. Testowany jest każdy zaimplementowany pakiet protokołu Kafki. Testy sprawdzają działanie serializacji i deserializacji poprzez porównanie ich wyników z ręcznie stworzonymi pakietami.

Oprócz tego testowane są mniejsze części implementacji, takie jak nawiązywanie połączenia TCP/IP lub logika klas wykonujących powtarzanie akcji w przypadku otrzymania błędu.

7.1.4. Optymalizacja

W celu uzyskania jak najlepszej wydajności producenta, czyli kluczowego komponentu naszego rozwiązania w tym względzie, przeprowadziliśmy szereg iteracji profilowania naszego rozwiązania. Początkowo nasz klient był znacznie wolniejszy od producenta KafkaProducer w Java, jednak ostatecznie udało nam się uzyskać lepszą wydajność od obu przetestowanych klientów (zob. rozdział Rezultaty).

Przy pomocy narzędzia do analizy wydajności perf oraz programu FlameGraph zlokalizowaliśmy miejsca w kodzie zajmujące największą część czasu działania oraz potencjalne powody takiej sytuacji. Wykonaliśmy następujące optymalizacje:

- redukcja kopiowania danych wysyłanych wiadomości w klasie `sender` poprzez `std::move` oraz niekopiujące operacje dodawania elementów `std::vector::emplace_back`
- dodatkowe przetworzenie metadanych w momencie ich otrzymania, przyspieszając bardzo częstą operację przeszukiwania w nich
- zastąpienie `std::map` poprzez `std::unordered_map`

- statyczna deklaracja tymczasowych buforów (słowo kluczowe `thread_local`), która zredukowała liczbę alokacji pamięci
- ustawienie opcji `NO_DELAY` na połączeniu TCP, które wyłączyło algorytm Nagle'a
- zamiana implementacji liczenia CRC32 z tej dostępnej w bibliotece Boost na wydajniejszą implementację korzystającą z instrukcji `crc32` dostępnej w zestawie instrukcji SSE4.2
- redukcja kopiowania danych w `partitioner` poprzez przekazywanie wyniku przez referencję

Rysunek 7.1: Rezultaty programu profilującego podczas początkowej iteracji

W rezultacie udało nam się usprawnić zarządzanie pamięcią i zmniejszyć liczbę alokacji wykonywanych przez program, a także miejscami zoptymalizować wykorzystanie pamięci cache oraz struktury użyte do przechowywania informacji w procesie działania klienta (jak np. me-

tadane).

7.2. Wyniki testowej replikacji

7.2.1. Środowisko testowe

W celu przeprowadzenia testów odwzorowaliśmy lokalnie środowisko, w którym serwis replikujący będzie działał w praktyce (na serwerach użytkowników Scylli). Ograniczyliśmy się do dwóch instancji Scylli, ponieważ wszystkie jej serwisy i tak działają niezależnie od siebie na każdym węźle. Instancja docelowa uruchomiana została na obrazie Docker systemu Fedora, w najnowszej wersji release. Jako instancję macierzystą uruchomiliśmy zbudowaną lokalnie wersję z naszym kodem, również na systemie operacyjnym Fedora (w tym przypadku konfiguracji jako obraz Docker oraz na maszynie wirtualnej VirtualBox). Ponadto, potrzebne były dwa komponenty pośredniczące - ScyllaDB Sink Connector (opisany w sekcji 6.2.2) oraz instancja serwisu Confluent, składająca się z centrum zarządzania, rejestru schematów, klastra brokerów oraz pluginu podpinającego konektory.

Do manualnej kontroli nad obiema instancjami używaliśmy programu CQLsh, pozwalającego nawiązać połączenie z bazą danych i wykonywać komendy w języku CQL.

7.2.2. Testy poprawnościowe

Napisany przez nas serwis replikujący jest pierwszym serwisem w Scylli obsługującym replikację między instancjami. Z tego względu oraz tego, że Scylla jest środowiskiem wysoce złożonym, przeprowadzenie testów wydajnościowych i właściwe zinterpretowanie ich byłoby ciężkie. W kodzie tym mało jest jednak miejsca na poprawki, ponieważ większość operacji, takich jak zapytania do tabeli CDC, musi się odbyć. Wąskim gardłem replikacji jest klient Kafki - jego dogłębna analiza wydajnościowa, wraz z porównaniami do innych rozwiązań dostępnych na rynku, znajduje się w następnym rozdziale.

Niemniej jednak, w celu przetestowania funkcjonalności rozwiązania przeprowadziliśmy szereg testów poprawnościowych. Testy te były manualne, sprawdzające różne warianty obsługiwanej replikacji. Mutacje wprowadzane były poprzez CQLsh na instancji macierzystej, a następnie stan docelowej bazy danych monitorowany tym samym programem na drugiej instancji. Przetestowane warianty to między innymi:

- seria operacji `INSERT`
- seria operacji `INSERT`, następnie operacja `UPDATE`
- seria operacji `INSERT`, następnie seria operacji `DELETE`
- seria operacji, wyłączenie instancji, ponowne włączenie i następna seria
- poprawność działania dla jednego węzła na instancji macierzystej
- poprawność działania dla wielu węzłów na instancji macierzystej

Rozdział 8

Rezultaty

8.1. Testy wydajności klienta Kafki (środowisko lokalne)

8.1.1. Metodologia

W teście wydajności mierzymy czas potrzebny klientom Kafki na wysłanie danego zbioru wiadomości. Zmierzony czas uwzględnia proces inicjalizacji klienta (między innymi początkowego połączenia do klastra Kafki), ale nie zawiera czasu przygotowania danych do wysłania.

Konfiguracja parametrów wszystkich klientów została ustawiona w identyczny sposób. Klienci:

- pracują w trybie producenta
- nie kolejkują wiadomości
- wysyłają wiadomości synchronicznie (przed wysłaniem kolejnej czekają na potwierdzenie dostarczenia poprzedniej wiadomości)
- wymagają potwierdzenia odbioru tylko od jednego brokera Kafki

Dla każdego testowanego klienta został napisany prosty program testowy, wysyłający zadaną liczbę wiadomości. Każda z wysyłanych wiadomości jest kopią jednego statycznie predefiniowanego ciągu znaków.

Każdy test został uruchomiony trzy razy, a czas trwania przedstawiony w wynikach jest średnią arytmetyczną tych prób.

8.1.2. Sprzęt testowy

Testy były uruchomione pod maszyną wirtualną VirtualBox.

- System hosta: Windows 10
- System gościa: Fedora 30
- Procesor: Intel Core i7 7700HQ (wszystkie rdzenie dostępne w maszynie wirtualnej)
- Pamięć RAM: 16GB (12GB dostępnych w maszynie wirtualnej)
- Dysk HDD: 1TB

8.1.3. Testowani klienci

Nasz klient

Został napisany prosty program testowy, korzystający z biblioteki Seastar oraz naszego klienta Kafki. Biblioteka Seastar, klient oraz program testowy zostały skompilowane za pomocą kompilatora G++ w wersji 9.0.1, używając poziomu optymalizacji `O2`.

KafkaProducer (Java)

Został napisany prosty program testowy w języku Java, korzystający z oficjalnej biblioteki Apache Kafka dla Java. Program korzysta z klasy `KafkaProducer`.

8.1.4. Wyniki testu wydajności

Liczba wiadomości	Wielkość wiadomości (bajty)	Czas KafkaProducer (s)	Czas naszego klienta (s)	Przyspieszenie
10.000	10	7,51	**3,48**	53%
10.000	100	7,77	**3,60**	53%
10.000	1000	8,16	**3,66**	55%
100.000	10	47,58	**30,46**	35%
100.000	100	48,43	**32,04**	33%
100.000	1000	53,83	**38,21**	29%
1.000.000	10	484,63	**372,48**	23%
1.000.000	100	501,90	**318,00**	36%
1.000.000	1000	514,89	**344,50**	33%
			Średnia geometryczna:	**32%**

(Wyniki dla liczby wiadomości równej 10.000 nie zostały doliczone do średniej)

8.2. Testy wydajności klienta Kafki (chmura Amazon)

8.2.1. Metodologia

W testach wydajności zastosowaliśmy podobną metodologię jak w przypadku środowiska lokalnego. W teście wydajności mierzymy czas potrzebny klientom Kafki na wysłanie danego zbioru wiadomości. W odróżnieniu od testu lokalnego, wiadomości wysyłamy grupami, symulując bardziej realistyczny przypadek użycia producenta. Zmierzony czas uwzględnia proces inicjalizacji klienta (między innymi początkowego połączenia do klastra Kafki), ale nie zawiera czasu przygotowania danych do wysłania.

Konfiguracja parametrów wszystkich klientów została ustawiona w jak najbardziej identyczny sposób (m.in. wymóg potwierdzenia odbioru wiadomości tylko od jednego brokera Kafki, wielkość buforów). Szczegóły zostały opisane w rozdziale Testowani klienci.

Dla każdego testowanego klienta został napisany prosty program testowy, wysyłający zadaną liczbę wiadomości. Każda z wysyłanych wiadomości jest kopią jednego statycznie predefiniowanego ciągu znaków. W przypadku testów z trzema brokerami, ostatnie dwa bajty klucza wiadomości są ustawiane na numer wiadomości modulo 100, aby umożliwić rozdysponowanie wiadomości między wieloma partycjami.

Listing 8.1: Pseudokod testowego programu

```
// Inicjalizacja
String wiadomosc = init_wiadomosc(dlugosc);

Time czas_start = aktualny_czas();
Producent producent = init_producent();

// Nadanie wiadomosci grupami
for (int i = 0; i < liczba_grup; i++) {
    for (int j = 0; j < wielkosc_grupy; j++) {
        wiadomosc = zmien_ostatnie_2_bajty(wiadomosc, j % 100);
        producent.produce(topic, wiadomosc, wiadomosc);
    }
    producent.flush();
}
producent.disconnect();

Time czas_koniec = aktualny_czas();
print(czas_koniec - czas_start);
```

Testy zostały przeprowadzone na klastrze złożonym z jednego brokera Kafki oraz na klastrze z trzema brokerami. Stworzony temat, na który wysyłaliśmy wiadomości miał liczbę partycji równą liczbie brokerów oraz współczynnik replikacji (ang. replication factor) równy jeden. Dzięki temu w testach z trzema brokerami, ruch był rozdzielony równo między brokerami.

8.2.2. Sprzęt testowy

Testy były uruchomione na chmurze Amazon AWS. Zostały uruchomione następujące maszyny:

- Maszyna testowa, z której uruchamiane były testy wydajności
- Instancja Apache ZooKeeper w wersji 3.5.7
- Jeden lub trzech brokerów Apache Kafka w wersji 2.5.0 (na osobnych instancjach)

Wszystkie uruchomione instancje były typu EC2 m5n.2xlarge, o specyfikacji:

- System: Ubuntu Server 20.04 LTS (wirtualizacja HVM)
- Procesor: 8 wątków procesora drugiej generacji Intel Xeon Scalable Processors (Cascade Lake) działający z utrzymanym taktowaniem turbo 3.1 GHz
- Pamięć RAM: 32 GB
- Dysk SSD: 25 GB, Amazon EBS typ gp2 (100 IOPS, 250 MB/s)
- Sieć: do 25 Gbps

Wybraliśmy ten typ instancji, ponieważ oferuje on dobry balans procesora CPU i wielkości RAM oraz oferuje on lepszą wydajność sieciową (litera “n” w nazwie). Instancje zostały uruchomione w tym samym regionie chmury i tej samej podsieci. Przy tworzeniu instancji wybraliśmy opcję “Placement group” w trybie “Cluster”, która stara się uruchomić instancje blisko siebie, dzięki temu ograniczając opóźnienia sieci.

8.2.3. Testowani klienci

Nasz klient

Został napisany prosty program testowy, korzystający z biblioteki Seastar oraz naszego klienta Kafki. Biblioteka Seastar, klient oraz program testowy zostały skompilowane za pomocą kompilatora G++ w wersji 9.0.1, używając poziomu optymalizacji `O3`. W kliencie zostało wyłączone automatyczne okresowe wysyłanie wiadomości, ponieważ jest wykonywane manualnie w programie testowym.

KafkaProducer (Java)

Został napisany prosty program testowy w języku Java, korzystający z oficjalnej biblioteki Apache Kafka dla języka Java. Program korzysta z klasy `KafkaProducer`. Parametr `batch.size` oraz `linger.ms` zostały ustawione na wysokie wartości, tak aby wszystkie wiadomości w wysyłanej grupie mieściły się w buforach oraz producent nie wysyłał automatycznie wiadomości przed zakończeniem grupy i manualnym wykonaniu `flush()`.

librdkafka (C++)

Został napisany prosty program testowy w języku C++, korzystający z biblioteki librdkafka dla C++. Program korzysta z klasy `RdKafka::Producer`. Zostały ustawione parametry `batch.num.messages` i `queue.buffering.max.messages` na wielkość wysyłanej grupy.

Problematycznym okazało się poprawne ustawienie parametru `linger.ms`, ponieważ metoda `flush()` nie wymuszała wysyłania wiadomości przed zakończeniem okresu oczekiwania. Z tego powodu testy wykonywaliśmy trzykrotnie, testując wartości `0`, `0.3`, `5` i wybierając najlepszy czas wykonania.

Librdkafka tworzy wątek roboczy dla każdego brokera, rozdzielając pracę na kilka rdzeni. Aby móc dobrze porównać czas wykonania z innymi klientami (które są jednowątkowe), ograniczyliśmy wielowątkowość klienta. Jako, że biblioteka nie umożliwia wyłączenia tej funkcjonalności, wykorzystaliśmy komendę `taskset`, aby uruchomić proces na jednym rdzeniu procesora.

8.2.4. Wyniki testu wydajności (klaster jeden broker)

Test 1

W tym teście wysyłaliśmy 80 milionów wiadomości, każda z nich po 10 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 1000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**53,40 s**	**1.498.127 wiad/s**	**14,20 MB/s**
KafkaProducer	70,83 s	1.129.464 wiad/s	10,77 MB/s
librdkafka	79,72 s	1.003.512 wiad/s	9,57 MB/s

Test 2

W tym teście wysyłaliśmy 80 milionów wiadomości, każda z nich po 100 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 1000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**81,68 s**	**979.431 wiad/s**	**93,40 MB/s**
KafkaProducer	95,46 s	838.047 wiad/s	79,92 MB/s
librdkafka	92,84 s	861.697 wiad/s	82,17 MB/s

Test 3

W tym teście wysyłaliśmy 8 milionów wiadomości, każda z nich po 1000 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 100.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**38,51 s**	**207.738 wiad/s**	**198,11 MB/s**
KafkaProducer	43,52 s	183.823 wiad/s	175,30 MB/s
librdkafka	41,24 s	193.986 wiad/s	184,99 MB/s

Test 4

W tym teście wysyłaliśmy 80 milionów wiadomości, każda z nich po 50 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 10.000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**60,52 s**	**1.321.877 wiad/s**	**63,03 MB/s**
KafkaProducer	77,01 s	1.038.826 wiad/s	49,53 MB/s
librdkafka	61,07 s	1.309.972 wiad/s	62,46 MB/s

8.2.5. Wyniki testu wydajności (klaster trzech brokerów)

Test 1

W tym teście wysyłaliśmy 80 milionów wiadomości, każda z nich po 10 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 1000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**46,53 s**	**1.719.320 wiad/s**	**16,39 MB/s**
KafkaProducer	72,00 s	1.111.111 wiad/s	10,59 MB/s
librdkafka	119,77 s	667.946 wiad/s	6,37 MB/s

Test 2

W tym teście wysyłaliśmy 78 milionów wiadomości, każda z nich po 10 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 3000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**38,96 s**	**2.002.053 wiad/s**	**19,09 MB/s**
KafkaProducer	57,68 s	1.352.288 wiad/s	12,89 MB/s
librdkafka	81,09 s	961.894 wiad/s	9,17 MB/s

Test 3

W tym teście wysyłaliśmy 100 milionów wiadomości, każda z nich po 100 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 5000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**69,59 s**	**1.436.988 wiad/s**	**137,04 MB/s**
KafkaProducer	90,49 s	1.105.094 wiad/s	105,39 MB/s
librdkafka	98,67 s	1.013.479 wiad/s	96,65 MB/s

Test 4

W tym teście wysyłaliśmy 10 milionów wiadomości, każda z nich po 1000 bajtów (sumaryczna długość klucza i wartości). Wiadomości były grupowane po 2000.

Nazwa klienta	Czas wykonania (s)	Liczba wiadomości na sekundę	Przepustowość (MB/s)
Nasz klient	**29,43 s**	**339.789 wiad/s**	**324,04 MB/s**
KafkaProducer	30,86 s	324.044 wiad/s	309,03 MB/s
librdkafka	31,17 s	320.821 wiad/s	305,95 MB/s

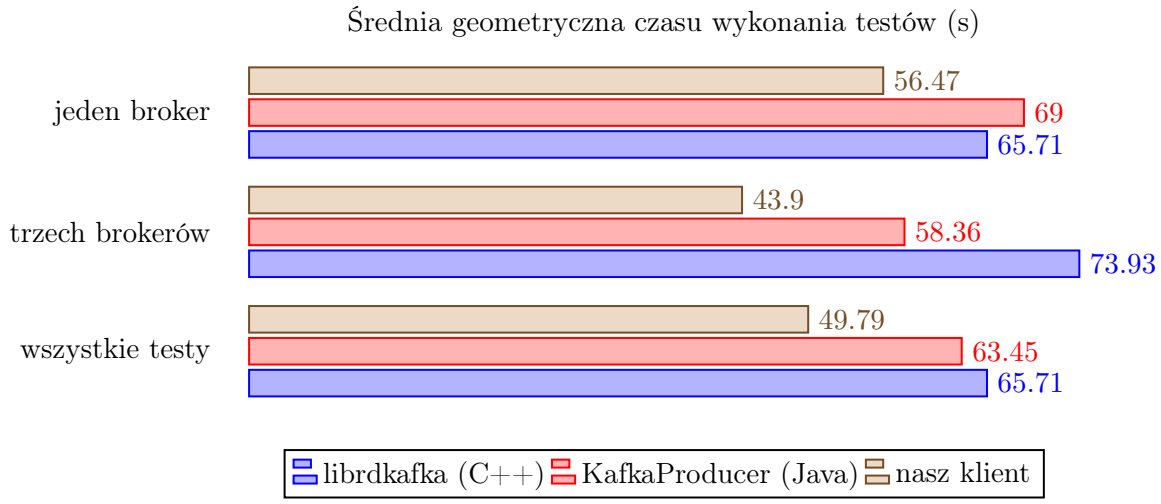
8.2.6. Podsumowanie wyników

Test	KafkaProducer (s)	Nasz klient (s)	Przyspieszenie
Testy wydajności (klaster jeden broker)			
Test 1	70,83 s	53,40 s	24%
Test 2	95,46 s	81,68 s	14%
Test 3	43,52 s	38,51 s	11%
Test 4	77,01 s	60,52 s	21%
Testy wydajności (klaster trzech brokerów)			
Test 1	72,00 s	46,53 s	35%
Test 2	57,68 s	38,96 s	32%
Test 3	90,49 s	69,59 s	23%
Test 4	30,86 s	29,43 s	4%
		Średnia geometryczna:	**22%**

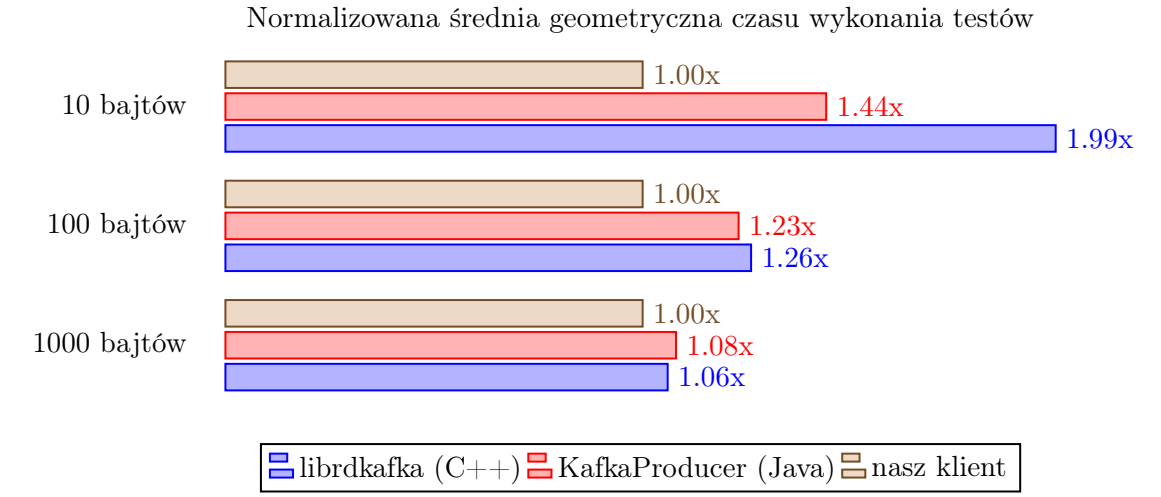
Test	librdkafka (s)	Nasz klient (s)	Przyspieszenie
Testy wydajności (klaster jeden broker)			
Test 1	79,72 s	53,40 s	33%
Test 2	92,84 s	81,68 s	12%
Test 3	41,24 s	38,51 s	6%
Test 4	61,07 s	60,52 s	0,9%
Testy wydajności (klaster trzech brokerów)			
Test 1	119,77 s	46,53 s	61%
Test 2	81,09 s	38,96 s	51%
Test 3	98,67 s	69,59 s	29%
Test 4	31,17 s	29,43 s	5%
		Średnia geometryczna:	**29%**

Jak pokazuje tabela, nasz klient działa szybciej niż KafkaProducer w Javie i librdkafka w C++. Udaje nam się uzyskać o 22% i 29% lepszą wydajność od tych klientów. Widać jednak

duże zróżnicowanie przyspieszenia, co wskazuje, że przy niektórych typach obciążenia nie będzie widać poprawy wyników.



Librdkafka jest szybszy od KafkaProducer w testach z jednym brokerem, natomiast KafkaProducer jest bardziej efektywny od librdkafka przy obsłudze wielu brokerów. Nasz klient w obu przypadkach jest szybszy od librdkafka i KafkaProducer, ale największa poprawa jest w testach z trzema brokerami.



Jak pokazuje powyższy wykres, nasz klient ma dużą przewagę w przypadku małych wiadomości. W przypadku wiadomości o wielkości 1000 bajtów, różnica wydajności między klientami jest bardzo mała (poniżej 10%).

Rozdział 9

Dalszy rozwój

Zaprezentowane przez nas rozwiązanie jest podstawową wersją serwisu replikującego i daje solidną bazę do dalszego rozwoju tego modułu Scylli. Napotkane ograniczenia tworzą nowe, możliwe ścieżki wzbogacania, a rozwiązane problemy podstawę do jeszcze lepszych optymalizacji. Nasz serwis jest pierwszą implementacją tej funkcjonalności w Scylli i jako taka stanowi dla przyszłych iteracji faktyczny punkt do porównań.

Poniżej przedstawiamy te z najbardziej naturalnych i kluczowych w naszym odczuciu ulepszeń aktualnego rozwiązania.

9.1. Funkcjonalność

Jedną z miar funkcjonalności serwisu replikacyjnego jest to ile typów mutacji jest w stanie obsłużyć. W obecnym stanie są to dwie operacje obsługiwane w pełni (tj. `INSERT` oraz `DELETE`) oraz operacja `UPDATE`, obsługiwana w sposób częściowy (szczegółowo opisane w sekcji 6.2.2). Ograniczenia te w dużej mierze nałożone zostały przez komponent z drugiego końca procesu replikacji - ScyllaDB Sink Connector. Jego obecna implementacja interpretuje przychodzące dane jako `INSERT` lub `DELETE`. Następnym krokiem w poszerzaniu funkcjonalności powinno więc być wzbogacenie konsumenckiej strony replikacji - albo usprawnienie obecnego konektora, albo wymiana na inny komponent.

9.2. Wydajność

W kwestii wydajności można na pewno powalczyć o dalsze, lokalne optymalizacje serwisu replikującego działającego po stronie instancji macierzystej bazy (tj. tej, którą replikujemy). Aktualnym wąskim gardłem procesu nie jest jednak w żadnym wypadku moduł produkujący, a strona konsumująca wysyłane informacje. W tym momencie droga każdego pakietu od brokera do instancji docelowej przebiega przez Sink Connector, którego zadaniem jest translacja wysłanych danych na polecenie CQL, a następnie wykonanie tego polecenia w podpiętej bazie. Usunięcie rzeczonego pośrednika i zastąpienie go natywnym rozwiązaniem (w formie, chociażby, serwisu konsumującego) powinno przynieść znaczne wzrosty wydajnościowe. Korzyści te wynikają z faktu zredukowania narzutu sieciowego stwarzanego przez dodatkowy węzeł w procesie (konektor), a także z potencjalnie dużo efektywniejszego konsumowania i mutacji bazy docelowej poprzez wykorzystanie zintegrowanego ze Scyllą konsumenta we frameworku Seastar.

Rozdział 10

Podział i systematyka pracy

10.1. Organizacja

Całość prac przebiegała w koordynacji z przedstawicielem ze strony zleceniodawcy. Po krótkim okresie wdrażania i zaznajomienia z frameworkiem oraz jego głównymi założeniami, rozwijanie rozwiązania postępowało przyrostowo, poczynając od klienta Kafki. Synchronizacje ze zleceniodawcą odbywały się w regularnych odstępach co tydzień i skupiały się na przedstawieniu najnowszych rezultatów oraz przedyskutowaniu następnych decyzji projektowych z uwzględnieniem nowo napotkanych przeszkód.

Nasz produkt końcowy nie jest autonomiczną aplikacją, a nowymi elementami rozszerzającymi bibliotekę Seastar oraz kod Scylli. Z tego powodu, oprócz wzajemnych rewizji przez członków zespołu, kod produktu musiał również przejść rewizje pracowników firmy ScyllaDB, poza standardową akceptacją funkcjonalności.

Śledzenie zmian oraz przydziału zadań w projekcie odbywało się przy użyciu platformy GitHub z funkcją Git Projects, jak również list mailingowych.

10.2. Podział prac

Warstwowy charakter klienta uniemożliwiał podział prac na kompletnie niezależne fragmenty, naturalnie narzucając rozwijanie iteracyjne. Z tego względu podział na zadania odbywał się na zasadzie puli mniejszych, zwartych problemów do zaimplementowania, lub zaprojektowania. Taka granularna struktura spowodowała, że większość elementów kodu została dotknięta przez każdego z zespołu. Poniżej przedstawiamy te zagadnienia, w których dana osoba miała największy udział.

Wojciech Bączkowski

- implementacja strategii partycjonowania
- serializacja i deserializacja danych w zapytaniach protokołu Kafka
- generowanie schematów tabel kompatybilnych z rejestrem schematów platformy Confluent
- opis schematu serializacji danych w formacie Avro

- konfiguracja środowiska deweloperskiego do replikacji
- organizacja płynnego przepływu informacji między promotorem i firmą zamawiająca oraz zespołem

Piotr Grabowski

- interfejs wysyłania zapytań (grupowanie, polityka ponownych prób)
- serializacja i deserializacja danych w zapytaniach protokołu Kafka
- testy E2E (end-to-end, całościowe) oraz automatyzacja infrastruktury testowej
- testy jednostkowe
- aplikacja demonstracyjna wykorzystująca napisanego klienta
- pomiary wydajności
- optymalizacja klienta Kafki

Michał Szostek

- konwersja i enkodowanie danych w formacie Avro
- komunikacja serwisu replikującego z bazą danych, obsługa zapytań
- menadżer metadanych klienta Kafki
- obsługa błędów opisanych przez protokół Kafki
- integracja biblioteki do obsługi formatu Avro jako submodułu Scylli

Piotr Wojtczak

- warstwa sieciowa i menadżer połączeń klienta
- zewnętrzny interfejs producenta
- aplikacja demonstracyjna wykorzystująca napisanego klienta
- konfiguracja klienta jako niezależnej biblioteki i jej integracja z systemem budowania Scylli
- wykrywanie tabel do potencjalnej replikacji
- integracja serwisu replikacji z napisanym producentem
- optymalizacja klienta Kafki

Bibliografia

[Book01] Christopher J. Date, *An Introduction To Database Systems, 8e, tłum. własne*

[Book02] Bettina Kemme, Ricardo Jiménez Peris, Marta Patiño-Martínez, *Database Replication (Synthesis Lectures on Data Management), tłum. własne*

[Book03] Laine Campbell, Charity Majors, *Database Reliability Engineering: Designing and Operating Resilient Database Systems*

[Stat01] Liczba urządzeń podłączonych do sieci, https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[Git01] Repozytorium Scylli, https://github.com/scylladb/scylla

[Git02] Repozytorium Seastara, https://github.com/scylladb/seastar

[Git03] kafka-connect-scylladb na Githubie, https://github.com/scylladb/kafka-connect-scylladb

[Url01] Lista implementacji klientów Kafki, https://cwiki.apache.org/confluence/display/KAFKA/Clients

[Url02] Encoding, https://developers.google.com/protocol-buffers/docs/encoding

[Url03] Redis Cluster Specification, https://redis.io/topics/cluster-spec

[Url04] Big Data Storage - Comparing Speed and Features for Avro, JSON, ORC, and Parquet, https://www.slideshare.net/Hadoop_Summit/big-data-storage-comparing-speed-and-features-for-avro-json-orc-and-parquet

[Sea01] Seastar Futures and Promises, http://seastar.io/futures-promises/

[Sea02] Seastar shared-nothing design, http://seastar.io/shared-nothing/

[Sea03] Seastar networking, http://seastar.io/networking/

[Scy01] Scylla vs. Cassandra at Samsung: YCSB Benchmark, https://www.scylladb.com/product/benchmarks/samsung-benchmark/

[Scy02] Scylla CDC Log Table, https://docs.scylladb.com/using-scylla/cdc/cdc-log-table/

[Scy03] Scylla Ring Architecture, https://docs.scylladb.com/architecture/ringarchitecture/

[Wiki01] Log trigger design pattern, https://en.wikipedia.org/wiki/Log_trigger

[Wiki02] Optimistic Concurrency Control https://en.wikipedia.org/wiki/Optimistic_concurrency_control