O'REILLY®

Wydanie II

Kafka w praktyce

Przetwarzanie strumieniowe i potoki danych o dużej skali

Gwen Shapira, Todd Palino
Rajini Sivaram, Krit Petty

Helion

Tytuł oryginału: Kafka The Definitive Guide Real-Time Data and Stream Processing at Scale, 2nd Edition

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-9320-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/kafpra
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/kafpra.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Konsumenty Kafki — odczytywanie danych z Kafki

Aplikacje, które muszą odczytywać dane z Kafki, używają konsumenta `KafkaConsumer`, by subskrybować tematy Kafki i otrzymywać komunikaty z tych tematów. Odczytywanie danych z Kafki różni się nieco od odczytywania danych z innych systemów wymiany komunikatów i wiąże się z tym kilka unikatowych koncepcji i pomysłów. Zrozumienie sposobu korzystania z interfejsu API konsumenta (Consumer API) bez uprzedniego zrozumienia tych pojęć może być trudne. Zaczniemy od wyjaśnienia kilku ważnych koncepcji, a następnie przeanalizujemy przykłady, które pokażą różne sposoby wykorzystania interfejsów API konsumenta do implementowania aplikacji o odmiennych wymaganiach.

Koncepcje konsumenckie Kafki

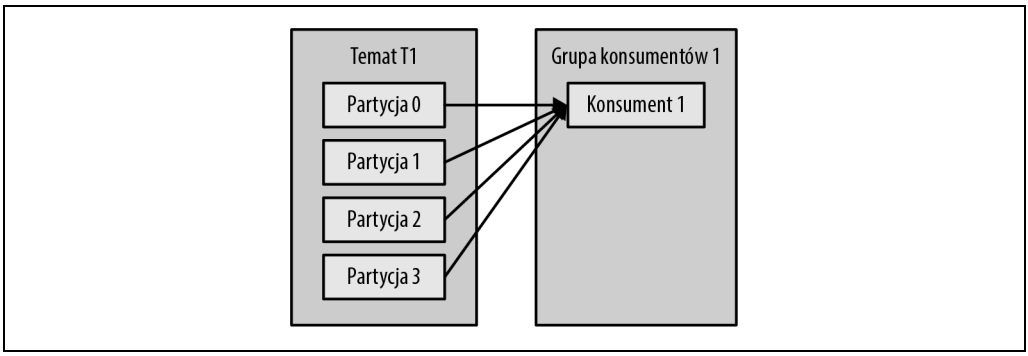
Aby zrozumieć, jak należy odczytywać dane z Kafki, najpierw trzeba poznać działanie jej konsumentów i grup konsumentów. Te koncepcje omówimy w kolejnych punktach.

Konsumenty i grupy konsumentów

Załóżmy, że masz aplikację, która musi odczytywać komunikaty z tematu Kafki, przeprowadzać ich walidacje i zapisywać wyniki w innym magazynie danych. W takim przypadku aplikacja tworzy obiekt konsumenta, subskrybuje odpowiedni temat i zaczyna odbierać komunikaty, sprawdzać ich poprawność i zapisywać wyniki. Może to działać przez jakiś czas, lecz co się stanie, jeśli szybkość zapisywania komunikatów w temacie przez producenty zacznie przekraczać szybkość, z jaką Twoja aplikacja będzie mogła je weryfikować? Jeżeli jesteś ograniczony do pojedynczego konsumenta odczytującego i przetwarzającego dane, aplikacja może zacząć pozostawać coraz bardziej w tyle, nie nadążając za szybkością przychodzących komunikatów. Oczywiście istnieje potrzeba skalowania konsumpcji z tematów. Podobnie jak wielu producentów może zapisywać w tym samym temacie, musisz umożliwić odczytywanie z tego samego tematu wielu konsumentom, dzieląc dane między nich.

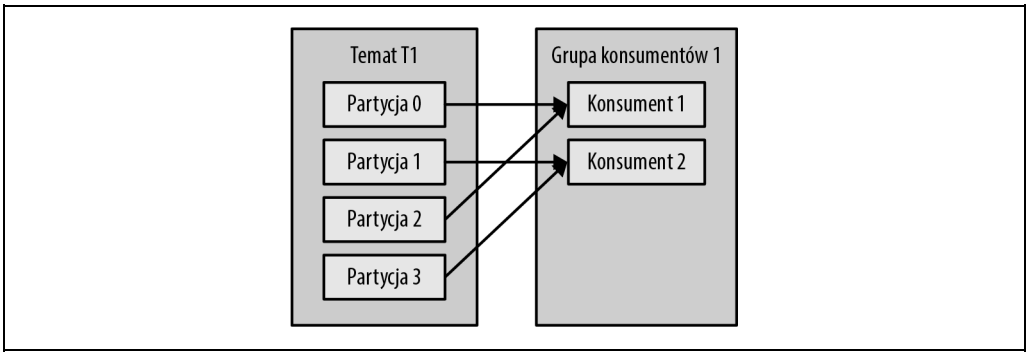
Konsumenty Kafki są zazwyczaj częścią **grupy konsumentów** (ang. *consumer group*). Gdy wiele konsumentów subskrybuje określony temat i należy do tej samej grupy konsumentów, każdy konsument z grupy będzie otrzymywał komunikaty z innego podzbioru partycji z tematu.

Przyjmijmy, że mamy temat T1 z czterema partycjami. Załóżmy również, że utworzyliśmy nowy konsument K1, będący jedynym konsumentem w grupie G1, i używamy go do subskrybowania tematu T1. Konsument K1 będzie otrzymywał wszystkie komunikaty ze wszystkich czterech partycji T1, jak pokazaliśmy na rysunku 4.1.



Rysunek 4.1. Jedna grupa konsumentów z czterema partycjami

Jeśli do grupy G1 dodamy kolejny konsument K2, każdy z nich będzie otrzymywał komunikaty tylko z dwóch partycji. Być może komunikaty z partycji 0 i 2 będą trafiać do K1, a komunikaty z partycji 1 i 3 do K2, jak pokazaliśmy na rysunku 4.2.

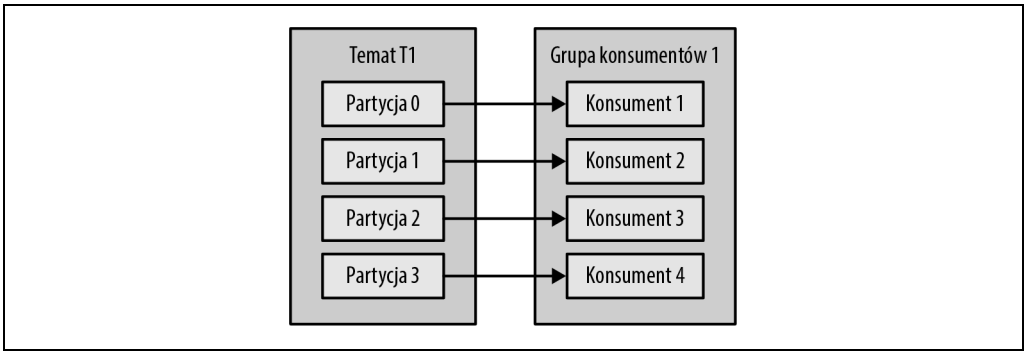


Rysunek 4.2. Cztery partycje podzielone na dwa konsumenty w grupie

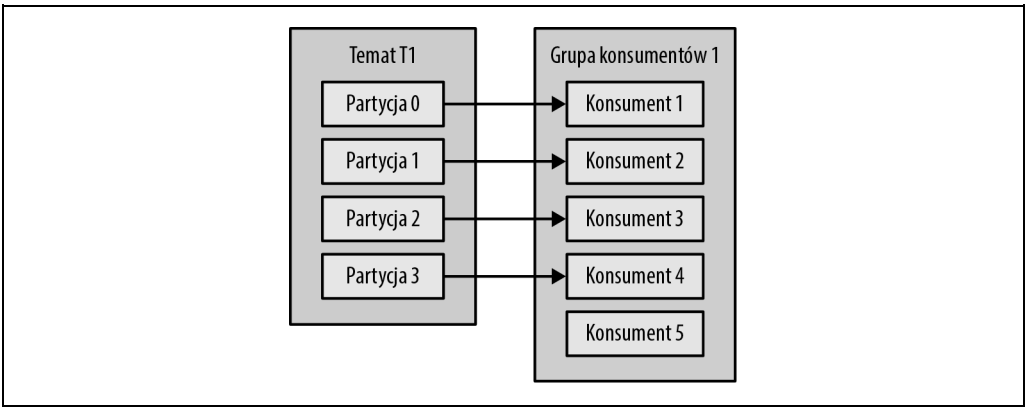
Gdyby grupa G1 miała cztery konsumenty, każdy z nich odczytywałby komunikaty z jednej partycji. Pokazaliśmy to na rysunku 4.3.

Jeżeli dodamy do pojedynczej grupy z jednym tematem więcej konsumentów, niż mamy partycji, niektóre konsumenty będą bezczynne i nie będą otrzymywać żadnych komunikatów. Pokazaliśmy to na rysunku 4.4.

Głównym sposobem skalowania konsumowania danych z tematu Kafki jest dodanie do grupy większej liczby konsumentów. Konsumenty Kafki często wykonują operacje charakteryzujące się dużym opóźnieniem, np. zapisy w bazie danych lub czasochłonne obliczenia na danych. W takich przypadkach pojedynczy konsument prawdopodobnie nie będzie w stanie nadążyć za prędkością przepływu



Rysunek 4.3. Cztery konsumenty w grupie; każdy ma po jednej partycji



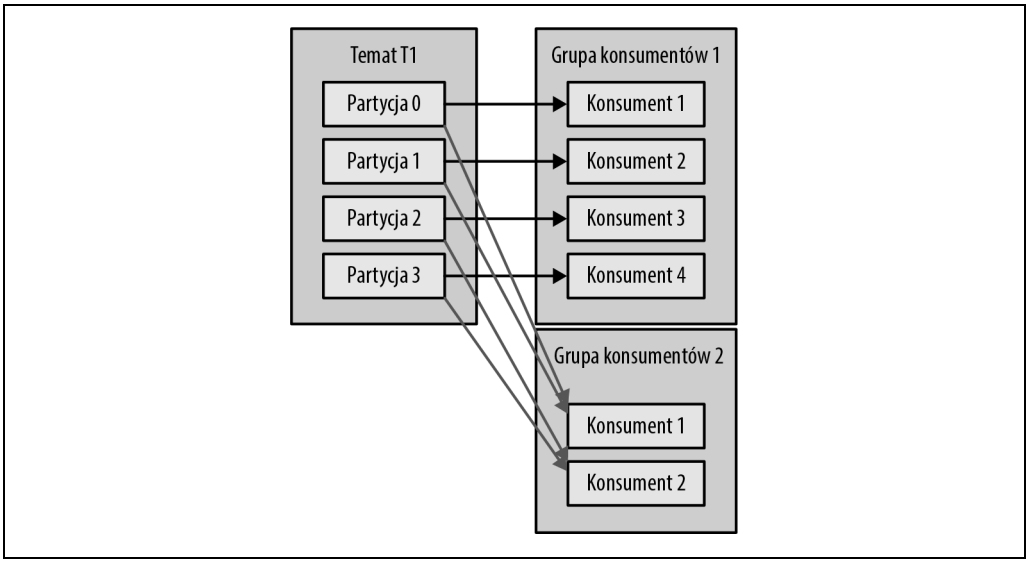
Rysunek 4.4. Gdy w grupie jest więcej konsumentów niż partycji, mamy bezczynne konsumenty

danych do tematu, więc główną metodą skalowania jest dodanie większej liczby konsumentów współdzielących obciążenie w taki sposób, że każdy konsument jest właścicielem tylko podzbioru partycji i komunikatów. To dobry powód, aby tworzyć tematy z dużą liczbą partycji, ponieważ gdy obciążenie wzrośnie, można dodawać konsumenty. Pamiętaj, że nie ma sensu dodawanie więcej konsumentów, niż jest partycji w temacie — niektóre konsumenty byłyby po prostu bezczynne. Kilka sugestii dotyczących ustalania liczby partycji w temacie znajdziesz w rozdziale 2.

Oprócz dodawania konsumentów w celu skalowania pojedynczej aplikacji bardzo często zdarza się, że wiele aplikacji musi odczytywać dane z tego samego tematu. Tak naprawdę jednym z głównych celów projektowych w Kafce było umożliwienie wykorzystania danych produkowanych do tematów Kafki w wielu przypadkach użycia w całej organizacji. W takich sytuacjach chcemy, aby każda aplikacja otrzymywała wszystkie komunikaty, a nie tylko ich podzbiór. By mieć pewność, że aplikacja będzie otrzymywać wszystkie komunikaty z tematu, trzeba jej zagwarantować własną grupę konsumentów. Inaczej niż wiele tradycyjnych systemów wymiany komunikatów Kafka skaluje się do dużej liczby konsumentów i grup konsumentów bez zmniejszania wydajności.

Wróćmy do naszego przykładu. Jeśli dodamy nową grupę konsumentów (G2) z pojedynczym konsumentem, będzie on otrzymywać wszystkie komunikaty z tematu T1 bez względu na to, co będzie

robić G1. G2 może mieć więcej niż jeden konsument, a wtedy każdy z nich otrzyma podzbiór partycji, tak jak pokazaliśmy dla G1, ale G2 jako całość nadal będzie otrzymywać wszystkie komunikaty niezależnie od innych grup konsumentów. Pokazaliśmy to na rysunku 4.5.



Rysunek 4.5. Gdy dodamy nową grupę konsumentów, obie grupy będą otrzymywać wszystkie komunikaty

Możemy podsumować, że dla każdej aplikacji, która potrzebuje wszystkich komunikatów z co najmniej jednego tematu, tworzymy nową grupę konsumentów. Aby skalować odczytywanie i przetwarzanie komunikatów z tematów, do istniejącej grupy dodajemy kolejne konsumenty, dzięki czemu każdy dodatkowy konsument w grupie będzie otrzymywał tylko podzbiór komunikatów.

Grupy konsumentów i równoważenie partycji

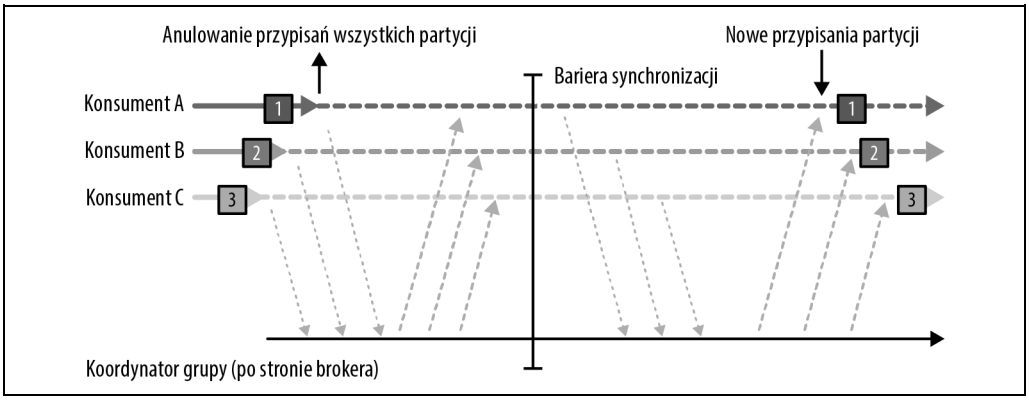
Jak dowiedziałeś się z poprzedniego punktu, konsumenty w grupie konsumentów mają udział we własności partycji w tematach, które subskrybują. Po dodaniu do grupy nowy konsument zaczyna konsumować komunikaty z partycji przetwarzanej wcześniej przez inny konsument. To samo dzieje się, gdy jakiś konsument się wyłącza lub ulega awarii, przez co opuszcza grupę, a przetwarzane przez niego partycje zaczynają być konsumowane przez jeden z pozostałych konsumentów. Ponowne przypisanie partycji do konsumentów ma miejsce również wtedy, gdy modyfikowane są tematy przetwarzane przez daną grupę konsumentów, np. kiedy administrator dodaje nowe partycje.

Przenoszenie własności partycji z jednego konsumenta na drugi nazywa się **równoważeniem obciążenia** lub **rebalansowaniem** (ang. *rebalance*). Równoważenie obciążenia jest istotne, ponieważ zapewnia grupie konsumentów wysoką dostępność i skalowalność (umożliwiając nam łatwe i bezpieczne dodawanie i usuwanie konsumentów), ale w normalnym toku zdarzeń może być niepożądane.

W zależności od strategii przypisywania partycji używanej przez daną grupę konsumentów dostępne są dwa rodzaje równoważenia obciążenia[1]:

Gorliwe równoważenie obciążenia

Podczas gorliwego równoważenia obciążenia (ang. *eager rebalance*) wszystkie konsumenty wstrzymują przetwarzanie, rezygnują z własności wszelkich partycji, ponownie dołączają do danej grupy konsumentów i otrzymują zupełnie nowe przypisania partycji. Generalnie jest to krótkie okno niedostępności całej grupy konsumentów. Czas trwania okna niedostępności zależy od wielkości grupy konsumentów oraz od kilku parametrów konfiguracyjnych. Na rysunku 4.6 pokazaliśmy, że gorliwe równoważenie obciążenia ma dwie odrębne fazy: po pierwsze, wszystkie konsumenty rezygnują ze swoich przypisań partycji, a po drugie, po zakończeniu tego procesu i ponownym dołączeniu do grupy otrzymują nowe przypisania partycji i mogą wznowić konsumowanie komunikatów.



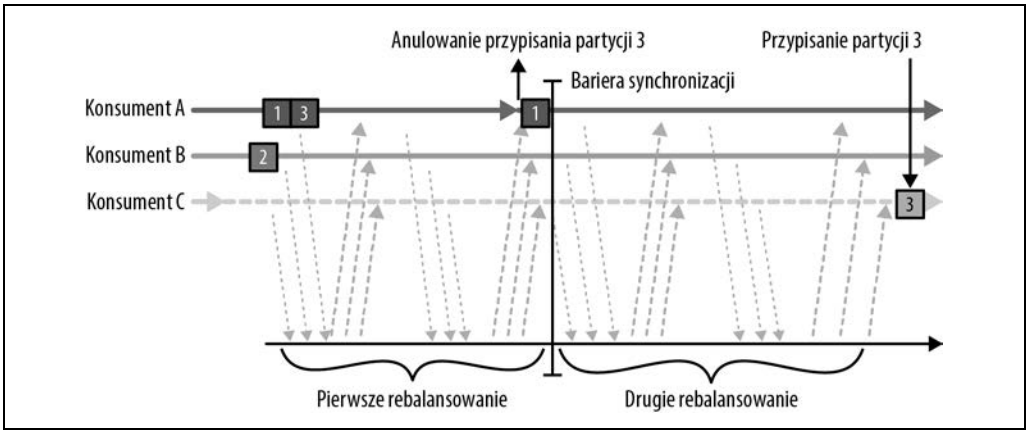
Rysunek 4.6. Gorliwe równoważenie obciążenia anuluje wszystkie przypisania partycji, wstrzymuje przetwarzanie i ponownie przypisuje partycje

Kooperacyjne równoważenie obciążenia

Kooperacyjne równoważenie obciążenia (nazywane również **równoważeniem przyrostowym**; ang. *incremental rebalance*) zazwyczaj polega na ponownym przypisywaniu tylko niewielkiego podzbioru partycji po kolei dla poszczególnych konsumentów i umożliwieniu konsumentom kontynuowania przetwarzania rekordów ze wszystkich partycji, które nie zostały ponownie przypisane. Osiąga się to przez rebalansowanie w co najmniej dwóch fazach. Na początku lider grupy konsumentów informuje wszystkie konsumenty, że stracą własność podzbioru swoich partycji, następnie konsumenty przestają przetwarzać komunikaty z tych partycji i rezygnują z ich własności. W drugiej fazie lider grupy konsumentów przypisuje te osierocone partycje nowym właścicielom. Takie przyrostowe podejście może wymagać wykonania kilku iteracji, dopóki nie zostanie osiągnięte stabilne przypisanie partycji, ale pozwala uniknąć całkowitej niedostępności („zatrzymania świata"), która występuje przy podejściu gorliwym. Jest to szczególnie ważne w dużych grupach konsumentów, w których równoważenie obciążenia może

[1] Diagramy Sophie Blee-Goldman z jej postu *From Eager to Smarter in Apache Kafka Consumer Rebalances* (*https://oreil.ly/fZzac*) na blogu Confluenta z maja 2020 r.

zająć sporo czasu. Na rysunku 4.7 pokazaliśmy, na czym polega przyrostowość kooperacyjnego równoważenia obciążenia i że w dowolnym momencie zaangażowany jest tylko podzbiór konsumentów i partycji.



Rysunek 4.7. Kooperacyjne równoważenie obciążenia wstrzymuje przetwarzanie tylko dla podzbioru partycji, które są ponownie przypisywane

Konsumenci zachowują członkostwo w grupie konsumentów i własność przypisanych im partycji, wysyłając do brokera Kafki wyznaczonego jako **koordynator grupy** (ten broker może być różny dla różnych grup konsumentów) **sygnały heartbeat**. Sygnały są wysyłane przez działający w tle wątek konsumenta i dopóki konsument wysyła heartbeaty w regularnych odstępach czasu, zakłada się, że jest aktywny.

Jeśli konsument nie będzie wysyłał sygnału heartbeat przez dłuższy czas, to jego sesja przekroczy limit czasu, a koordynator grupy uzna go za nieaktywny i uruchomi rebalansowanie. Gdy konsument ulegnie awarii i przestanie przetwarzać komunikaty, upłynie kilka sekund bez wysyłania sygnału heartbeat, zanim koordynator grupy stwierdzi, że konsument przestał działać, i wywoła rebalansowanie. W ciągu tych kilku sekund nie będą przetwarzane żadne komunikaty z partycji należących do nieaktywnego konsumenta. W przypadku czystego zakończenia działania konsument powiadamia o tym koordynatora grupy, który natychmiast uruchamia rebalansowanie, skracając przerwę w przetwarzaniu komunikatów. Dalej w tym rozdziale omówimy opcje konfiguracyjne, które kontrolują częstotliwość sygnału heartbeat, limity czasu sesji i inne parametry, których można użyć w celu dostosowania zachowania konsumenta.

Jak działa proces przypisywania partycji konsumentom?

Gdy konsument chce dołączyć do grupy, wysyła do koordynatora grupy żądanie `JoinGroup`. Pierwszy konsument, który dołączy do grupy, zostaje jej **liderem**. Lider otrzymuje od koordynatora grupy listę wszystkich należących do niej konsumentów (lista obejmuje wszystkie konsumenty, które ostatnio wysłały sygnał heartbeat i w związku z tym są uznawane za aktywne) i staje się odpowiedzialny za przypisywanie podzbioru partycji do każdego konsumenta. Do decydowania, które partycje powinny być obsługiwane przez poszczególne konsumenty, używa implementacji klasy `PartitionAssignor`.

Kafka ma kilka wbudowanych reguł przypisywania partycji, które omówimy szerzej dalej w tym rozdziale. Po podjęciu decyzji o przypisaniu partycji lider grupy konsumentów wysyła listę przypisań do koordynatora grupy (`GroupCoordinator`), który przekazuje te informacje do wszystkich konsumentów. Każdy konsument zna tylko własne przypisanie — lider jest jedynym procesem klienckim, który ma pełną listę konsumentów w grupie i ich przypisań. Proces ten powtarza się za każdym razem, gdy uruchamiane jest równoważenie obciążenia.

Statyczne członkostwo w grupie

Domyślnie tożsamość konsumenta jako członka grupy konsumentów jest przejściowa. Gdy konsumenty opuszczają grupę, ich przypisania partycji są anulowane, a po ponownym dołączeniu do grupy konsument otrzymuje nowy identyfikator członka i nowy zbiór partycji za pośrednictwem protokołu równoważenia obciążenia.

Nie ma to zastosowania, jeżeli skonfiguruje się konsument z unikatowym identyfikatorem `group.instance.id`, co czyni go **statycznym** członkiem grupy. Gdy konsument po raz pierwszy dołącza do grupy konsumentów jako jej statyczny członek, przypisywany jest do niego zbiór partycji zgodnie ze strategią przypisywania partycji, z której w normalnym trybie korzysta grupa. Jednak po zakończeniu działania konsumenta nie opuszcza on automatycznie grupy — pozostaje członkiem grupy do czasu wygaśnięcia jego sesji. Gdy konsument ponownie dołącza do grupy, jest rozpoznawany na podstawie swojej statycznej tożsamości i jest ponownie przypisywany do tych samych partycji, które miał wcześniej, bez wywoływania równoważenia obciążenia. Koordynator grupy buforujący przypisania dla wszystkich członków grupy nie musi uruchamiać równoważenia obciążenia, a jedynie wysłać przypisanie z pamięci podręcznej do ponownie dołączającego statycznego członka.

Jeśli do tej samej grupy dołączają dwa konsumenty o tym samym identyfikatorze `group.instance.id`, drugi w kolejności konsument otrzyma komunikat błędu informujący, że konsument o tym identyfikatorze już istnieje.

Statyczne członkostwo w grupie jest przydatne, gdy aplikacja utrzymuje lokalny stan lub lokalną pamięć podręczną zapełnianą przez partycje przypisywane do poszczególnych konsumentów. Jeżeli odtworzenie tej pamięci podręcznej okaże się czasochłonne, nie jest pożądane, by ten proces następował przy każdym restarcie konsumenta. Z drugiej strony należy pamiętać, że przy restarcie jednego konsumenta partycje należące do poszczególnych konsumentów nie są ponownie przypisywane. Przez pewien czas żaden konsument nie będzie konsumował z komunikatów z tych partycji, a gdy dany konsument w końcu uruchomi kopię zapasową, będzie w tyle za najnowszymi komunikatami z tych partycji. Należy mieć pewność, że konsument będący właścicielem tych partycji będzie w stanie nadrobić opóźnienie powstałe po restarcie.

Trzeba zwrócić uwagę, że statyczni członkowie grup konsumentów przy zamykaniu nie opuszczają grupy proaktywnie, a wykrywanie, że „naprawdę ich nie ma", zależy od konfiguracji parametru `session.timeout.ms`. Powinieneś ustawić dla niego na tyle dużą wartość, by ustrzec się wyzwalania równoważenia obciążenia po zwykłym restarcie aplikacji, ale jednocześnie wartość ta powinna umożliwiać automatyczne ponowne przypisywanie danych partycji w przypadku dłuższego przestoju w celu uniknięcia dłuższych przerw w przetwarzaniu tych partycji.

Tworzenie konsumenta Kafki

Pierwszym krokiem do rozpoczęcia konsumowania rekordów jest utworzenie instancji klasy `Kafka ↪Consumer`. Tworzenie obiektu `KafkaConsumer` bardzo przypomina konstruowanie obiektu `Kafka ↪Producer` — tworzysz instancję klasy `Properties` Javy z właściwościami, które chcesz przekazać konsumentowi. Wszystkie właściwości omówimy szczegółowo dalej w tym rozdziale. Na początek wystarczy użyć trzech obowiązkowych właściwości: `bootstrap.servers`, `key.deserializer` i `value.deserializer`.

Właściwość `bootstrap.servers` to łańcuch znaków dla połączenia z klastrem Kafki. Używa się go dokładnie w taki sam sposób, jak w przypadku instancji `KafkaProducer` (szczegółowe informacje na temat jego definiowania znajdziesz w rozdziale 3.). Pozostałe dwie właściwości, `key.deserializer` i `value.deserializer`, są podobne do serializatorów definiowanych dla producenta, ale zamiast określania klas przekształcających obiekty Javy w tablice bajtów należy określić klasy przekształcające tablice bajtów w obiekty Javy.

Istnieje też czwarta właściwość, która nie jest bezwzględnie obowiązkowa, ale bardzo powszechnie stosowana. Jest to właściwość `group.id` określająca grupę konsumentów, do której należy dana instancja `KafkaConsumer`. Chociaż możliwe jest tworzenie konsumentów, które nie należą do żadnej grupy konsumentów, jest to stosunkowo rzadkie, dlatego dalej w tym rozdziale będziemy przyjmować założenie, że konsument jest częścią jakiejś grupy.

Poniższy fragment kodu pokazuje, jak utworzyć obiekt `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

Jeśli czytałeś rozdział 3. poświęcony tworzeniu producentów, większość tego kodu powinna wyglądać znajomo. Zakładamy, że konsumowane rekordy będą miały obiekty typu `String` zarówno jako klucz, jak i wartość rekordu. Jedyną nową właściwością jest tutaj `group.id`, czyli nazwa grupy konsumentów, do której należy dany konsument.

Subskrybowanie tematów

Następnym krokiem po utworzeniu konsumenta jest zasubskrybowanie co najmniej jednego tematu. Metoda `subscribe()` przyjmuje jako parametr listę tematów, więc jest całkiem prosta w użyciu:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Tworzymy listę z jednym elementem: nazwą tematu `customerCountries`.

Możliwe jest również wywoływanie `subscribe` z wyrażeniem regularnym. Wyrażenie może dopasowywać wiele nazw tematów, a jeśli zostanie utworzony nowy temat o pasującej nazwie, niemal

natychmiast przeprowadzone zostanie rebalansowanie, a konsumenty zaczną go przetwarzać. Jest to przydatne w aplikacjach, które muszą korzystać z wielu tematów i mogą obsługiwać różne typy danych, które będą zawierały tematy. Subskrybowanie wielu tematów za pomocą wyrażenia regularnego jest najczęściej używane w aplikacjach replikujących dane między Kafką a innym systemem lub w aplikacjach przetwarzania strumieniowego.

Aby zasubskrybować np. wszystkie tematy testowe, możemy wykonać takie wywołanie:

```
consumer.subscribe(Pattern.compile("test.*"));
```

Jeśli Twój klaster Kafki ma dużą liczbę partycji, np. 30 000 lub więcej, powinieneś mieć świadomość, że filtrowanie tematów pod kątem subskrypcji odbywa się po stronie klienta. Oznacza to, że gdy zasubskrybujesz podzbiór tematów za pomocą wyrażenia regularnego, a nie za pomocą bezpośredniej listy, konsument będzie w regularnych odstępach czasu żądał od brokera listy wszystkich tematów i ich partycji. Klient będzie następnie używał tej listy do wykrywania nowych tematów, które powinien uwzględnić i zasubskrybować. Gdy lista tematów jest długa i masz wiele konsumentów, wówczas rozmiar listy tematów i partycji jest duży, a subskrybowanie przy użyciu wyrażeń regularnych generuje znaczne obciążenie brokera, klienta i sieci. W niektórych przypadkach przepustowość wykorzystywana przez metadane tematu jest większa niż przepustowość wykorzystywana do wysyłania danych. Oznacza to również, że w celu subskrybowania za pomocą wyrażenia regularnego klient potrzebuje uprawnień do opisywania wszystkich tematów w klastrze — czyli pełnych uprawnień `describe` dla całego klastra.

Pętla odpytywania

Sercem interfejsu Consumer API jest prosta pętla do odpytywania serwera w celu pozyskiwania kolejnych danych. Główne ciało konsumenta będzie wyglądało następująco:

```
Duration timeout = Duration.ofMillis(100);

while (true) { ❶
    ConsumerRecords<String, String> records = consumer.poll(timeout); ❷

    for (ConsumerRecord<String, String> record : records) { ❸
        System.out.printf("temat = %s, partycja = %d, przesunięcie = %d, " +
                          "klient = %s, kraj = %s\n",
        record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);

        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()); ❹
    }
}
```

1. Jest to w rzeczywistości pętla nieskończona. Konsumenty to zwykle długo działające aplikacje, które nieustannie odpytują Kafkę, by pozyskiwać kolejne dane. Dalej w tym rozdziale pokażemy, jak czysto wyjść z pętli i zamknąć konsument.
2. To najważniejsza linia kodu w rozdziale. Podobnie jak rekiny muszą się poruszać albo umrą, konsumenty muszą stale odpytywać Kafkę, gdyż inaczej zostaną uznane za nieaktywne, a konsumowane przez nie partycje zostaną przekazane innemu konsumentowi z grupy, by kontynuował ich przetwarzanie. Parametrem przekazywanym do metody `poll()` jest interwał limitu czasu, który kontroluje, jak długo metoda będzie blokować, jeśli dane nie będą dostępne w buforze konsumenta. Gdy ustawiona jest wartość `0` lub istnieją już dostępne rekordy, metoda `poll()` natychmiast kończy wykonywanie; w przeciwnym razie będzie czekać przez określoną liczbę milisekund.
3. Metoda `poll()` zwraca listę rekordów. Każdy rekord zawiera temat i partycję swojego pochodzenia, przesunięcie rekordu w obrębie partycji oraz rzecz jasna klucz i wartość rekordu. Zazwyczaj iterujemy przez listę i przetwarzamy poszczególne rekordy indywidualnie.
4. Przetwarzanie zwykle kończy się zapisaniem wyniku w magazynie danych lub aktualizacją przechowywanego rekordu. W tym przypadku celem jest utrzymywanie bieżącej liczby klientów z poszczególnych krajów, dlatego aktualizujemy tablicę mieszającą i wypisujemy wynik w formacie JSON. Bardziej realistycznym przykładem byłoby zapisywanie wyników aktualizacji w magazynie danych.

Pętla `poll` nie zajmuje się jedynie pobieraniem danych. Przy pierwszym wywołaniu z nowym konsumentem metoda `poll()` jest odpowiedzialna za znalezienie instancji `GroupCoordinator`, dołączenie konsumenta do grupy i odebranie przypisania partycji. Jeśli uruchomione zostanie równoważenie obciążenia, również zostanie ono obsłużone w pętli odpytywania, łącznie z powiązanymi wywołaniami zwrotnymi. Oznacza to, że prawie wszystko, co może pójść nie tak z konsumentem lub w wywołaniach zwrotnych używanych przez jego nasłuchiwacze, prawdopodobnie wypłynie jako wyjątek rzucony przez `poll()`.

Należy pamiętać, że jeśli metoda `poll()` nie będzie wywoływana przez okres dłuższy niż `max.poll.interval.ms`, konsument zostanie uznany za niedziałający i usunięty z grupy konsumentów, unikaj więc robienia wszystkiego, co może blokować wykonywanie w pętli odpytywania przez nieprzewidywalne przedziały czasu.

Bezpieczeństwo wątków

W jednym wątku nie możesz mieć wielu konsumentów należących do tej samej grupy i nie możesz mieć wielu wątków bezpiecznie korzystających z tego samego konsumenta. Regułą jest jeden konsument na wątek. Aby uruchomić wielu konsumentów w tej samej grupie w jednej aplikacji, każdego musisz uruchomić w osobnym wątku. Przydatne jest opakowanie logiki konsumenta we własny obiekt, a następnie użycie usługi `ExecutorService` Javy do uruchomienia wielu wątków, z których każdy będzie miał własnego konsumenta. Na blogu Confluenta znajdziesz samouczek (*https://oreil.ly/8YOVe*), który pokazuje, jak to zrobić.

W starszych wersjach Kafki pełną sygnaturą metody było `poll(long)`; ta sygnatura jest już przestarzała, a nowy interfejs API to `poll(Duration)`. Oprócz zmiany typu argumentu nieznacznie zmieniła się semantyka blokowania wykonywana przez tę metodę. Oryginalna metoda `poll(long)` będzie blokować, dopóki nie uzyska z Kafki potrzebnych metadanych, nawet jeśli przekroczy limit czasu. Nowa metoda `poll(Duration)` będzie przestrzegać ograniczeń limitu czasu i nie będzie czekać na metadane. Jeżeli masz kod konsumenta wykorzystujący `poll(0)` jako metodę zmuszającą Kafkę do pobrania metadanych bez konsumowania jakichkolwiek rekordów (dość powszechna sztuczka), nie możesz zmienić go po prostu na `poll(Duration.ofMillis(0))` i oczekiwać tego samego zachowania. Musisz wymyślić nowy sposób na osiągnięcie swoich celów. Częstym rozwiązaniem jest umieszczenie logiki w metodzie `rebalanceListener.onPartitionAssignment()`, której wywołanie jest gwarantowane po uzyskaniu metadanych dla przypisanych partycji, ale przed rozpoczęciem napływania rekordów. Kolejne rozwiązanie zostało udokumentowane przez Jessego Andersona na blogu w poście *Kafka's Got a Brand-New Poll* (*https://oreil.ly/zN6ek*).

Innym możliwym podejściem może być zapełnianie kolejki zdarzeń przez jednego konsumenta i wykonywanie pracy z tej kolejki przez wiele wątków roboczych. Przykład tego wzorca znajdziesz na blogu w poście Igora Buzatovicia (*https://oreil.ly/uMzj1*).

Konfiguracja konsumentów

Do tej pory koncentrowaliśmy się na nauce interfejsu Consumer API i przyjrzeliśmy się tylko kilku właściwościom konfiguracyjnym — obowiązkowym `bootstrap.servers`, `key.deserializer` i `value.deserializer` oraz powszechnie stosowanej `group.id`. Cała konfiguracja konsumenta została opisana w dokumentacji Apache Kafki (*https://oreil.ly/Y00Gl*). Większość parametrów ma rozsądne wartości domyślne i nie wymaga modyfikacji, ale niektóre mają wpływ na wydajność i dostępność konsumentów. Przyjrzyjmy się kilku ważniejszym właściwościom.

fetch.min.bytes

Ta właściwość umożliwia konsumentowi określenie minimalnej ilości danych, które chce otrzymywać od brokera podczas pobierania rekordów — domyślnie jeden bajt. Jeżeli broker otrzyma od konsumenta żądanie rekordów, ale nowe rekordy będą miały mniej bajtów niż `fetch.min.bytes`, to przed wysłaniem rekordów do konsumenta broker poczeka, aż dostępnych będzie więcej komunikatów. Zmniejsza to obciążenie zarówno konsumenta, jak i brokera, ponieważ muszą wymieniać między sobą mniejszą liczbę komunikatów w przypadkach, gdy w tematach nie będzie zbyt dużo nowej aktywności (lub w godzinach zmniejszonej aktywności w ciągu dnia). Jeżeli konsument zużywa zbyt dużo procesora, gdy nie ma zbyt wielu dostępnych danych, lub potrzebujesz zmniejszyć obciążenie brokerów, gdy masz dużą liczbę konsumentów, możesz ustawić dla tego parametru wartość wyższą niż domyślna — pamiętaj jednak, że zwiększenie tej wartości może zwiększyć opóźnienia w przypadkach niskiej przepustowości.

fetch.max.wait.ms

Ustawienie `fetch.min.bytes` instruuje Kafkę, by przed przekazaniem konsumentowi odpowiedzi poczekała, aż będzie miała wystarczającą ilość danych do wysłania. Właściwość `fetch.max.wait.ms` pozwala kontrolować czas oczekiwania. Domyślnie Kafka czeka do 500 ms. Powoduje to do 500 ms dodatkowego opóźnienia w przypadku, gdy do tematu Kafki nie przepływa wystarczająca ilość danych, by spełnić wymagania minimalnej ilości danych do zwrócenia. Jeśli chcesz ograniczyć potencjalne opóźnienie (zwykle ze względu na umowy SLA kontrolujące maksymalne opóźnienia aplikacji), możesz ustawić niższą wartość `fetch.max.wait.ms`. Jeżeli dla `fetch.max.wait.ms` ustawisz wartość 100 ms, a dla `fetch.min.bytes` wartość 1 MB, to po otrzymaniu od konsumenta żądania pobrania Kafka odpowie danymi, gdy będzie miała 1 MB danych do zwrócenia lub po upływie 100 ms, w zależności od tego, co nastąpi wcześniej.

fetch.max.bytes

Ta właściwość pozwala określić maksymalną liczbę bajtów, które Kafka będzie zwracać przy każdym odpytywaniu brokera przez konsument (domyślnie 50 MB). Służy do ograniczania rozmiaru pamięci używanej przez konsument do przechowywania danych zwracanych z serwera, niezależnie od liczb partycji lub komunikatów. Zwróć uwagę, że rekordy są wysyłane do klienta w partiach, a jeśli pierwsza partia rekordów, którą ma wysłać broker, przekroczy ten rozmiar, partia zostanie wysłana, a limit zignorowany. Gwarantuje to konsumentowi robienie ciągłych postępów. Warto zauważyć, że istnieje analogiczna konfiguracja brokera, która pozwala administratorowi Kafki ograniczyć także maksymalny rozmiar pobierania. Konfiguracja brokera może być przydatna, ponieważ żądania dużych ilości danych mogą się wiązać z dużymi odczytami z dysku i długimi wysyłkami przez sieć, co może powodować rywalizację i zwiększać obciążenie brokera.

max.poll.records

Ta właściwość kontroluje maksymalną liczbę rekordów, które zwróci pojedyncze wywołanie `poll()`. Używaj jej do kontrolowania ilości danych (ale nie rozmiaru danych), które aplikacja będzie musiała przetworzyć w jednej iteracji pętli odpytywania.

max.partition.fetch.bytes

Właściwość kontroluje maksymalną liczbę bajtów, które serwer będzie zwracał na partycję (domyślnie 1 MB). Gdy wywołanie `KafkaConsumer.poll()` zwraca obiekt `ConsumerRecords`, używa on co najwyżej `max.partition.fetch.bytes` na partycję przypisaną do konsumenta. Zwróć uwagę, że kontrolowanie wykorzystania pamięci za pomocą tej konfiguracji może być dość skomplikowane, ponieważ nie masz kontroli nad liczbą partycji, które zostaną uwzględnione w odpowiedzi brokera. Dlatego zdecydowanie zalecamy zamiast tego użycie właściwości `fetch.max.bytes`, chyba że masz specjalne powody, by próbować przetwarzać podobne ilości danych z każdej partycji.

session.timeout.ms i heartbeat.interval.ms

Czas, przez który konsument może nie mieć kontaktu z brokerami i nadal być uznawany za aktywny, wynosi domyślnie 10 sekund. Jeśli upłynie więcej czasu, niż określa `session.timeout.ms`, a konsument nie wyśle do koordynatora grupy sygnału heartbeat, zostanie uznany za nieaktywny i koordynator grupy uruchomi rebalansowanie grupy konsumentów w celu przydzielenia partycji nieaktywnego konsumenta innym konsumentom z grupy. Ta właściwość jest ściśle powiązana z `heartbeat.interval.ms`, która kontroluje częstotliwość wysyłania przez konsument Kafki sygnału heartbeat do koordynatora grupy, podczas gdy `session.timeout.ms` kontroluje, jak długo konsument może działać bez wysyłania tego sygnału. Z tego powodu te dwie właściwości są zwykle modyfikowane razem — wartość `heartbeat.interval.ms` musi być mniejsza niż wartość `session.time` ↪`out.ms` i z reguły ustawia się dla niej jedną trzecią limitu czasu. Stąd jeśli `session.timeout.ms` wynosi 3 sekundy, wartość `heartbeat.interval.ms` powinna wynosić 1 sekundę. Ustawienie `session.timeout.ms` poniżej wartości domyślnej umożliwi grupom konsumentów szybsze wykrywanie awarii i odzyskiwanie sprawności po awarii, ale może również spowodować niepożądane rebalansowanie. Ustawienie wyższej wartości `session.timeout.ms` zmniejsza ryzyko przypadkowego rebalansowania, ale oznacza również, że wykrywanie rzeczywistych awarii będzie zajmowało więcej czasu.

max.poll.interval.ms

Właściwość pozwala ustawić przedział czasu, przez który konsument może działać bez odpytywania i nie zostać uznany za nieaktywny. Jak wspomnieliśmy wcześniej, sygnał heartbeat i limit czasu sesji stanowią główny mechanizm, dzięki któremu Kafka wykrywa nieaktywne konsumenty i usuwa ich przydziały partycji. Wspomnieliśmy jednak również, że sygnały heartbeat są wysyłane przez wątek działający w tle. Istnieje ryzyko, że główny wątek konsumujący z Kafki zostanie zakleszczony, ale wątek w tle nadal będzie wysyłał sygnały heartbeat. Będzie to oznaczać, że rekordy z partycji należących do tego konsumenta nie będą przetwarzane. Najłatwiejszym sposobem zweryfikowania, czy konsument nadal przetwarza rekordy, jest sprawdzenie, czy prosi o kolejne rekordy. Interwały między żądaniami kolejnych rekordów są jednak trudne do przewidzenia i zależą od ilości dostępnych danych, rodzaju przetwarzania wykonywanego przez konsument, a czasem od opóźnienia dodatkowych usług. W aplikacjach wymagających czasochłonnego przetwarzania każdego zwróconego rekordu parametr `max.poll.records` pozwala ograniczać ilości zwracanych danych, a tym samym skracać czas, w którym aplikacja nie jest przejściowo dostępna dla wywołań metody `poll()`. Nawet po zdefiniowaniu `max.poll.records` interwał między wywołaniami `poll()` jest trudny do przewidzenia, a parametr `max.poll.interval.ms` jest stosowany jako zabezpieczenie przed awarią. Musi to być na tyle duży interwał, by zdrowy konsument rzadko w pełni go wykorzystywał, ale jednocześnie na tyle mały, aby uniknąć znaczącego wpływu wywieranego przez zawieszony konsument. Domyślną wartością jest 5. Po przekroczeniu tego limitu czasu wątek działający w tle wysyła żądanie „opuść grupę”, aby poinformować broker, że konsument jest nieaktywny i obciążenie dla grupy musi zostać zrównoważone, a następnie przestaje wysyłać sygnały heartbeat.

default.api.timeout.ms

Jest to limit czasu, który będzie miał zastosowanie do (niemal) wszystkich wywołań interfejsu API wykonanych przez konsument, gdy nie określisz dla nich bezpośredniego limitu czasu. Wartość domyślna to 1 minuta, a ponieważ jest ona większa niż domyślny limit czasu żądania, uwzględnia ponowienie próby, jeśli zachodzi taka potrzeba. Godnym uwagi wyjątkiem od interfejsów API używających tej wartości domyślnej jest metoda `poll()`, która zawsze wymaga bezpośredniego limitu czasu.

request.timeout.ms

Jest to maksymalny czas oczekiwania konsumenta na odpowiedź od brokera. Jeżeli broker nie odpowie w tym czasie, klient założy, że broker nie odpowie w ogóle, zamknie połączenie i spróbuje połączyć się ponownie. Wartość domyślna tej konfiguracji wynosi 30 sekund i nie zaleca się jej obniżania. Przed rozłączeniem się trzeba pozostawić brokerowi wystarczająco dużo czasu na przetworzenie żądania — ponawianie wysyłania żądań do już przeciążonego brokera niewiele daje, a czynność rozłączania i ponownego łączenia generuje dodatkowe obciążenie.

auto.offset.reset

Ta właściwość steruje zachowaniem konsumenta, gdy zaczyna on odczytywać partycję, dla której nie ma zatwierdzonego przesunięcia, lub jeśli zatwierdzone przesunięcie jest nieprawidłowe (zwykle jest to spowodowane nieaktywnością konsumenta przez tak długi czas, że rekord z danym przesunięciem został już zdezaktualizowany przez broker). Wartość domyślna `latest` (najnowsze) oznacza, że przy braku prawidłowego przesunięcia konsument zacznie odczytywanie od najnowszych rekordów (czyli tych, które zostały zapisane po tym, jak konsument zaczął działać). Dostępna jest również wartość `earliest` (najstarsze), która oznacza, że w przypadku braku prawidłowego przesunięcia konsument będzie odczytywał wszystkie dane z partycji, zaczynając od samego początku. Ustawienie dla `auto.offset.reset` wartości `none` spowoduje rzucenie wyjątku podczas próby konsumowania do nieprawidłowego przesunięcia.

enable.auto.commit

Ten parametr kontroluje, czy konsument będzie zatwierdzał przesunięcia automatycznie, a jego domyślna wartość to `true`. Ustaw wartość `false`, jeśli wolisz kontrolować moment zatwierdzania przesunięć, co jest konieczne, by zminimalizować liczbę duplikatów i uniknąć wybrakowania danych. Jeżeli ustawisz `enable.auto.commit` na `true`, możesz również zechcieć kontrolować częstotliwość zatwierdzania przesunięć za pomocą właściwości `auto.commit.interval.ms`. Różne opcje zatwierdzania przesunięć omówimy szerzej dalej w tym rozdziale.

partition.assignment.strategy

Dowiedziałeś się, że partycje są przypisane do konsumentów w grupie konsumentów. `Partition↪Assignor` to klasa, która na podstawie danych konsumentów i subskrybowanych przez nie tematów przypisuje określone partycje do poszczególnych konsumentów. Oto domyślne strategie przypisywania Kafki:

Przypisywanie zakresowe

Przypisuje poszczególnym konsumentom kolejne podzbiory partycji z każdego subskrybowanego tematu. Jeśli konsumenty K1 i K2 subskrybują np. dwa tematy, T1 i T2, a każdy z tematów ma trzy partycje, to K1 otrzyma partycje 0 i 1 z tematów T1 i T2, podczas gdy K2 otrzyma partycję 2 z tych tematów. Ponieważ każdy temat ma nieparzystą liczbę partycji, a przypisywanie odbywa się niezależnie dla każdego tematu, pierwszy konsument otrzymuje więcej partycji niż drugi. Dzieje się tak za każdym razem, gdy używane jest przypisywanie zakresowe (`Range ↪Assignor`), a liczba partycji w każdym temacie nie dzieli się równo przez liczbę konsumentów.

Przypisywanie karuzelowe

Przypisuje wszystkie partycje ze wszystkich subskrybowanych tematów kolejno poszczególnym konsumentom, jednemu po drugim. Gdyby opisane wcześniej konsumenty K1 i K2 używały przypisywania karuzelowego (`RoundRobinAssignor`), K1 otrzymałby partycje 0 i 2 z tematu T1 i partycję 1 z tematu T2. K2 dostałby partycję 1 z tematu T1 oraz partycje 0 i 2 z tematu T2. Generalnie jeśli wszystkie konsumenty subskrybują te same tematy (co jest bardzo częstym scenariuszem), przypisywanie karuzelowe powoduje, że wszystkie będą miały taką samą liczbę partycji (lub będą różnić się co najwyżej jedną partycją).

Przypisywanie lepkie

Przypisywanie lepkie (`StickyAssignor`) ma dwa cele: po pierwsze, utrzymywać jak najbardziej zrównoważony przydział, a po drugie, po rebalansowaniu zachowywać możliwie najwięcej niezmienionych przydziałów, minimalizując obciążenie związane z przenoszeniem przypisań partycji z jednego konsumenta na drugi. W typowym przypadku, gdy wszystkie konsumenty subskrybują ten sam temat, początkowe przypisania będą tak samo zrównoważone jak w przypisywaniu karuzelowym, podobnie jak kolejne, ale partycje będą rzadziej przenoszone. Gdy konsumenty z tej samej grupy subskrybują różne tematy, przypisania lepkie będą bardziej zrównoważone niż przypisania karuzelowe.

Przypisywanie kooperacyjne lepkie

Ta strategia przypisywania (`CooperativeStickyAssignor`) jest taka sama jak w przypisywaniu lepkim, ale obsługuje kooperacyjne równoważenie obciążenia, gdzie konsumenty mogą nadal konsumować partycje, które nie zostały ponownie przypisane. Więcej informacji na temat kooperacyjnego równoważenia obciążenia znajdziesz wcześniej w tym rozdziale w punkcie „Grupy konsumentów i równoważenie partycji”. Pamiętaj też, że jeśli przeprowadzasz uaktualnienie z wersji starszej niż 2.3, wówczas w celu udostępnienia strategii przypisywania kooperacyjnego lepkiego musisz podążać określoną ścieżką aktualizacji, zapoznaj się więc uważnie z przewodnikiem aktualizacji (*https://oreil.ly/klMI6*).

Strategię przypisywania partycji możesz wybrać przez ustawienie właściwości `partition.assign ↪ment.strategy`. Wartość domyślna to `org.apache.kafka.clients.consumer.RangeAssignor`, która implementuje opisaną wcześniej strategię przypisywania zakresowego. Możesz ją zastąpić ustawieniem `org.apache.kafka.clients.consumer.RoundRobinAssignor`, `org.apache.kafka.clients.consu ↪mer.StickyAssignor` lub `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`. Bardziej zaawansowaną opcją jest zaimplementowanie własnej strategii przypisywania, w której wartość `partition.assignment.strategy` powinna wskazywać na nazwę Twojej klasy.

client.id

Może to być dowolny łańcuch znaków i będzie używany przez brokery do identyfikowania żądań wysyłanych przez klienty, np. żądań pobierania. Jest używany do rejestrowania komunikatów dzienników i wskaźników oraz do kwot.

client.rack

Domyślnie konsumenty pobierają komunikaty z repliki głównej każdej partycji. Jednak gdy klaster obejmuje wiele centrów danych lub wiele stref dostępności w chmurze, pobieranie komunikatów z repliki znajdującej się w tej samej strefie co konsument ma swoje zalety zarówno pod względem wydajności, jak i kosztów. Aby umożliwić pobieranie z najbliższej repliki, musisz ustawić konfigurację `client.rack` i określić strefę, w której znajduje się dany klient. Następnie możesz skonfigurować brokery, aby zastąpiły domyślną klasę `replica.selector.class` klasą `org.apache.kafka.com` `↪mon.replica.RackAwareReplicaSelector`.

Możesz również zaimplementować własną klasę `replica.selector.class` z niestandardową logiką, aby najlepszą replikę do konsumowania wybierać na podstawie metadanych klienta i partycji.

group.instance.id

Może to być dowolny unikatowy łańcuch znaków i służy do zapewniania konsumentowi statycznego członkostwa w grupie.

receive.buffer.bytes i send.buffer.bytes

Są to rozmiary buforów wysyłania i odbierania TCP używanych przez gniazda podczas zapisywania i odczytywania danych. Jeśli mają ustawioną wartość –1, używane są domyślne ustawienia systemu operacyjnego. Dobrym pomysłem może być zwiększenie wartości tych parametrów, gdy producenty lub konsumenty komunikują się z brokerami w innym centrum danych, ponieważ te łącza sieciowe zwykle mają większe opóźnienia i niższą przepustowość.

offsets.retention.minutes

Jest to konfiguracja brokera, ale trzeba ją znać ze względu na jej wpływ na zachowanie konsumentów. Dopóki grupa konsumentów ma aktywnych członków (tj. członków, którzy aktywnie utrzymują członkostwo w grupie przez wysyłanie sygnałów heartbeat), Kafka będzie zachowywać ostatnie przesunięcia zatwierdzone przez grupę dla poszczególnych partycji, będzie więc można je odzyskać w przypadku zmiany przypisań lub restartu. Jednak gdy grupa stanie się pusta, Kafka zachowa swoje zatwierdzone przesunięcia tylko do czasu ustawionego w tej konfiguracji — domyślnie przez 7 dni. Gdy przesunięcia zostaną usunięte, a dana grupa stanie się ponownie aktywna, będzie się zachowywać jak zupełnie nowa grupa konsumentów bez informacji o tym, co konsumowała w przeszłości. Zwróć uwagę, że to zachowanie zmieniało się kilka razy, dlatego jeżeli używasz wersji starszych niż 2.1.0, sprawdź dokumentację swojej wersji pod kątem oczekiwanego zachowania.

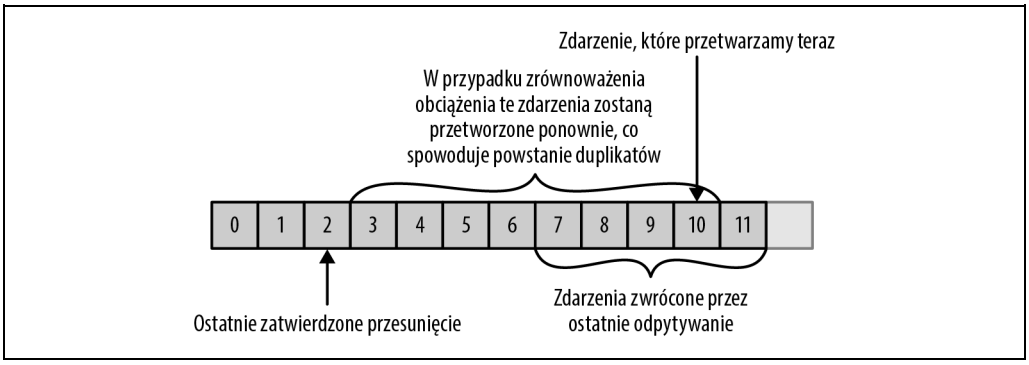
Zatwierdzenia i przesunięcia

Przy każdym wywołaniu metoda `poll()` zwraca zapisane w Kafce rekordy, których konsumenty z naszej grupy jeszcze nie odczytały. Oznacza to, że mamy sposób na śledzenie, które rekordy zostały odczytane przez konsumenta z danej grupy. Jak wspomnieliśmy wcześniej, jedną z unikatowych cech Kafki jest to, że nie śledzi potwierdzeń od konsumentów w taki sposób, jak robi to wiele kolejek JMS (ang. *Java Message Service*). Zamiast tego konsumenty mogą używać Kafki do śledzenia swoich pozycji (przesunięć) na poszczególnych partycjach.

Zaktualizowanie bieżącej pozycji na partycji nazywamy **zatwierdzeniem przesunięcia** (ang. *offset commit*). Inaczej niż tradycyjne kolejki komunikatów Kafka nie zatwierdza indywidualnych rekordów. Zamiast tego każdy konsument zatwierdza ostatni komunikat, który pomyślnie przetworzył z danej partycji, i domyślnie przyjmuje, że wszystkie wcześniejsze komunikaty również zostały przetworzone pomyślnie.

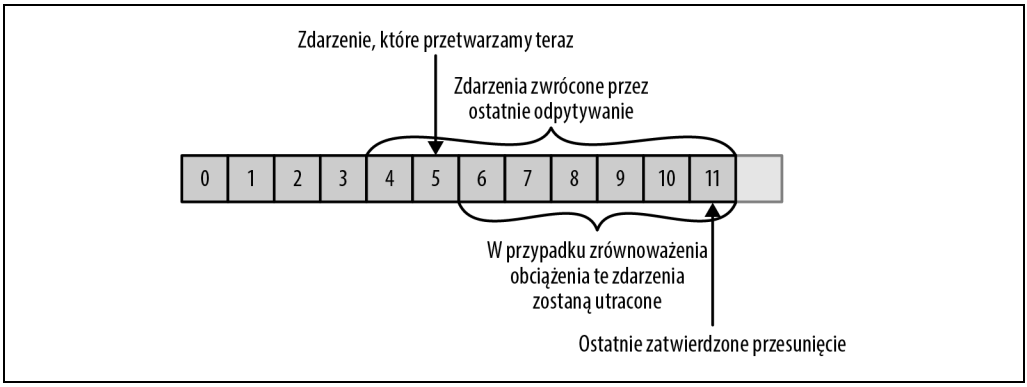
W jaki sposób konsument zatwierdza przesunięcie? Wysyła do Kafki komunikat, który aktualizuje specjalny temat `__consumer_offsets` zatwierdzonym przesunięciem dla poszczególnych partycji. Dopóki wszystkie konsumenty będą bezproblemowo działać, nie będzie to miało żadnych konsekwencji. Jeśli jednak jakiś konsument ulegnie awarii lub do grupy konsumentów dołączy nowy konsument, **uruchomione zostanie równoważenie obciążenia**. Po wykonaniu równoważenia obciążenia każdemu konsumentowi może zostać przypisany nowy zbiór partycji, inny od przetwarzanego wcześniej. Aby wiedzieć, od czego wznowić pracę, konsument odczytuje wtedy ostatnie zatwierdzone przesunięcie dla każdej partycji i od tego miejsca kontynuuje przetwarzanie komunikatów.

Jeśli zatwierdzone przesunięcie będzie mniejsze niż przesunięcie ostatniego komunikatu przetworzonego przez klient, komunikaty między ostatnim przetworzonym przesunięciem a zatwierdzonym przesunięciem zostaną przetworzone dwukrotnie. Pokazaliśmy to na rysunku 4.8.



Rysunek 4.8. Ponownie przetwarzane komunikaty

Jeżeli zatwierdzone przesunięcie będzie większe niż przesunięcie ostatniego komunikatu faktycznie przetworzonego przez klient, wówczas wszystkie komunikaty między ostatnim przetworzonym przesunięciem a zatwierdzonym przesunięciem zostaną przez daną grupę konsumentów pominięte. Pokazaliśmy to na rysunku 4.9.



Rysunek 4.9. Komunikaty pominięte między przesunięciami

Oczywiście zarządzanie przesunięciami ma duży wpływ na aplikację kliencką. Interfejs API `Kafka ↪Consumer` zapewnia wiele sposobów zatwierdzania przesunięć.

Które przesunięcie jest zatwierdzane?

Domyślnym zachowaniem przy zatwierdzaniu automatycznym lub bez określania zamierzonych przesunięć jest zatwierdzanie po ostatnim przesunięciu zwróconym przez `poll()`. Należy o tym pamiętać przy próbie ręcznego zatwierdzania określonych przesunięć lub przy dążeniu do ich zatwierdzenia. Żmudne jest również jednak wielokrotne czytanie komunikatu: „Zatwierdź przesunięcie, które jest o jeden większe niż ostatnie przesunięcie, które klient otrzymał z wywołania `poll()`" i w 99% przypadków nie ma to znaczenia. Dlatego gdy będziemy odnosić się do zachowania domyślnego, będziemy pisać: „Zatwierdź ostatnie przesunięcie". Pamiętaj o tej uwadze, jeżeli potrzebujesz ręcznie manipulować przesunięciami.

Zatwierdzanie automatyczne

Najłatwiejszym sposobem na zatwierdzenie przesunięć jest delegowanie tego zadania konsumentowi. Jeśli skonfigurujesz parametr `enable.auto.commit=true`, konsument co 5 sekund będzie zatwierdzał najnowsze przesunięcie, które klient otrzymał z wywołania metody `poll()`. Ustawieniem domyślnym jest interwał 5-sekundowy, który można kontrolować za pomocą właściwości `auto.commit.interval.ms`. Podobnie jak wszystkie inne kwestie dotyczące klienta, automatyczne zatwierdzenia są sterowane przez pętlę odpytywania. Przy każdym odpytywaniu konsument sprawdza, czy nadszedł czas na zatwierdzenie, a jeśli tak, zatwierdza przesunięcia zwrócone w ostatnim odpytywaniu.

Zanim jednak skorzystasz z tej wygodnej opcji, powinieneś zrozumieć jej konsekwencje.

Przypomnijmy, że domyślnie automatyczne zatwierdzenia odbywają się co 5 sekund. Załóżmy, że 3 sekundy po ostatnim zatwierdzeniu konsument uległ awarii. Po zrównoważeniu obciążenia pozostałe aktywne konsumenty zaczynają przetwarzać należące wcześniej do tego brokera partycje. Rozpoczynają jednak od ostatniego zatwierdzonego przesunięcia. W tym przypadku jest to przesunięcie sprzed 3 sekund, stąd wszystkie zdarzenia, które pojawiły się w ciągu tych 3 sekund, zostaną

przetworzone dwukrotnie. Możliwe jest skonfigurowanie krótszego interwału zatwierdzania i zmniejszenie okna, w którym rekordy będą duplikowane, ale nie można tego całkowicie wyeliminować.

Gdy przy włączonym automatycznym zatwierdzaniu nadchodzi czas na zatwierdzenie przesunięć, następne odpytywanie zatwierdza ostatnie przesunięcie zwrócone przez poprzednie odpytywanie. Nie wiadomo, które zdarzenia zostały rzeczywiście przetworzone, dlatego ważne jest, by przed ponownym wywołaniem `poll()` zawsze przetwarzać wszystkie zdarzenia zwrócone przez ostatnie wywołanie tej metody. (Metoda `close()` również automatycznie zatwierdza przesunięcia, podobnie jak `poll()`). Zwykle nie stanowi to problemu, ale musisz być ostrożny, gdy obsługujesz wyjątki lub przedwcześnie wychodzisz z pętli odpytywania.

Automatyczne zatwierdzenia są wygodne, lecz nie dają programistom wystarczającej kontroli, aby uniknąć duplikowania komunikatów.

Zatwierdzanie bieżących przesunięć

Programiści zazwyczaj starają się bardziej kontrolować momenty zatwierdzania przesunięć — w celu zarówno wyeliminowania ryzyka pominięcia komunikatów, jak i zmniejszenia liczby komunikatów duplikowanych podczas równoważenia obciążenia. Consumer API ma opcję zatwierdzania bieżącego przesunięcia w punkcie, który jest sensowny dla programisty aplikacji, a nie na podstawie timera.

Ustawienie `enable.auto.commit=false` powoduje, że przesunięcia są zatwierdzane tylko na wyraźne żądanie aplikacji. Najprostszym i najbardziej niezawodnym interfejsem API zatwierdzania jest `com`
`↪mitSync()`. Ten interfejs API zatwierdza ostatnie przesunięcie zwrócone przez `poll()` i kończy działanie po zatwierdzeniu przesunięcia, zgłaszając wyjątek, jeżeli zatwierdzenie z jakiegoś powodu się nie powiedzie.

Należy pamiętać, że `commitSync()` zatwierdza ostatnie przesunięcie zwrócone przez `poll()`, dlatego jeśli wywołasz `commitSync()` przed zakończeniem przetwarzania wszystkich rekordów ze zbioru, w przypadku awarii aplikacji ryzykujesz pominięcie komunikatów, które zostały zatwierdzone, ale nie przetworzone. Jeśli aplikacja ulegnie awarii podczas przetwarzania rekordów z danego zbioru, wszystkie komunikaty od początku ostatniej partii do momentu uruchomienia równoważenia obciążenia zostaną przetworzone dwukrotnie — być może lepsze to niż brakujące komunikaty.

W poniższym listingu pokazaliśmy sposób użycia `commitSync` do zatwierdzenia przesunięć po zakończeniu przetwarzania ostatniej partii komunikatów:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %d, przesunięcie =
            %d, klient = %s, kraj = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value()); ❶
    }
    try {
        consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
```

```
            log.error("zatwierdzenie się nie powiodło", e) ❸
        }
    }
```

❶ Załóżmy, że wypisując zawartość rekordu, zakończyliśmy jego przetwarzanie. Twoja aplikacja prawdopodobnie będzie robić z rekordami znacznie więcej — modyfikować je, wzbogacać, agregować, wyświetlać na dashboardzie lub powiadamiać użytkowników o ważnych zdarzeniach. Moment „kończenia" z rekordem powinieneś określać zgodnie ze swoim przypadkiem użycia.

❷ Gdy skończymy „przetwarzanie" wszystkich rekordów z bieżącej partii, przed odpytywaniem o kolejne komunikaty wywołujemy metodę `commitSync` w celu zatwierdzenia ostatniego przesunięcia w partii.

❸ Metoda `commitSync` ponawia próby zatwierdzenia, o ile nie ma błędu, którego nie można naprawić. Jeżeli taki błąd wystąpi, będziemy mogli jedynie go zarejestrować.

Zatwierdzanie asynchroniczne

Jedną z wad ręcznego zatwierdzania jest to, że aplikacja jest blokowana, dopóki broker nie odpowie na żądanie zatwierdzenia. Ogranicza to przepływność aplikacji. Przepływność można poprawić przez rzadsze zatwierdzanie, ale wtedy zwiększy się liczba potencjalnych duplikatów, które może wygenerować równoważenie obciążenia.

Kolejną opcją jest API zatwierdzania asynchronicznego. Zamiast czekać na odpowiedź brokera na zatwierdzenie, wysyłamy żądanie i kontynuujemy pracę:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s,
            przesunięcie = %d, klient = %s, kraj = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ Zatwierdzamy ostatnie przesunięcie i kontynuujemy pracę.

Minusem jest to, że o ile metoda `commitSync()` ponawia próbę zatwierdzenia, dopóki nie zakończy wykonywania pomyślnie lub nie napotka nieodwracalnego błędu, o tyle metoda `commitAsync()` nie podejmuje ponownej próby. Nieponawianie próby jest spowodowane tym, że zanim `commitAsync()` otrzyma odpowiedź z serwera, może nastąpić późniejsze zatwierdzenie, które już się powiodło. Wyobraź sobie, że wysłaliśmy żądanie zatwierdzenia przesunięcia o wartości 2000. Wystąpił tymczasowy problem z komunikacją, więc broker nigdy nie otrzymał żądania i dlatego nigdy nie odpowie. Tymczasem przetworzyliśmy kolejną partię i pomyślnie zatwierdziliśmy przesunięcie 3000. Jeżeli `commitAsync()` ponowi teraz poprzednie nieudane zatwierdzenie, może powieść się zatwierdzenie przesunięcia 2000 *po tym*, jak przesunięcie 3000 zostało już przetworzone i zatwierdzone. W przypadku uruchomienia równoważenia obciążenia spowoduje to wygenerowanie większej liczby duplikatów.

Wspominamy o tej komplikacji i znaczeniu prawidłowej kolejności zatwierdzeń, ponieważ metoda `commitAsync()` ma również opcję przekazania wywołania zwrotnego, które zostanie wywołane, gdy broker odpowie. Tego wywołania zwrotnego często używa się do rejestrowania błędów zatwierdzeń lub zliczania ich jako wskaźnika, ale jeśli chcesz wykorzystać wywołanie zwrotne do ponawiania prób, musisz mieć świadomość problemu z kolejnością zatwierdzeń:

```
Duration timeout = Duration.ofMillis(100);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s,
        przesunięcie = %d, klient = %s, kraj = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Nie powiodło się zatwierdzenie dla przesunięć {}", offsets, e);
        }
   }); (1)
}
```

(1) Wysyłamy zatwierdzenie i kontynuujemy pracę, ale jeśli zatwierdzenie się nie powiedzie, niepowodzenie i przesunięcia zostaną zarejestrowane.

Ponawianie zatwierdzania asynchronicznego

Prostym wzorcem umożliwiającym uzyskanie prawidłowej kolejności zatwierdzeń w przypadku asynchronicznego ponawiania prób jest użycie monotonicznie rosnącego numeru sekwencyjnego. Przy każdym zatwierdzeniu zwiększamy numer sekwencyjny i w momencie zatwierdzania dodajemy do wywołania zwrotnego `commitAsync`. Kiedy przygotowujemy się do wysłania ponownej próby, sprawdzamy, czy numer sekwencyjny zatwierdzenia otrzymany przez wywołanie zwrotne jest równy zmiennej instancyjnej. Jeśli tak, nie było nowszego zatwierdzenia i można bezpiecznie spróbować ponownie. Jeżeli numer sekwencyjny instancji jest wyższy, nie próbujemy ponownie, ponieważ wysłane zostało już nowsze zatwierdzenie.

Łączenie zatwierdzeń synchronicznych i asynchronicznych

Zwykle sporadyczne niepowodzenia zatwierdzania bez ponawiania prób nie stanowią dużego problemu, bo gdy problem jest tymczasowy, kolejne zatwierdzenie zakończy się sukcesem. Jeśli jednak wiemy, że jest to ostatnie zatwierdzenie przed zakończeniem działania konsumenta lub przed równoważeniem obciążenia, powinniśmy się upewnić, że zatwierdzenie się powiedzie.

Dlatego powszechnym wzorcem jest połączenie `commitAsync()` z `commitSync()` tuż przed zakończeniem działania. W poniższym listingu pokazaliśmy, jak to działa (zatwierdzanie tuż przed równoważeniem obciążenia omówimy, gdy przejdziemy do tematu nasłuchiwaczy rebalansowania):

```
Duration timeout = Duration.ofMillis(100);

try {
    while (!closing) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                klient = %s, kraj = %s\n",
                record.topic(), record.partition(),
                record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
    consumer.commitSync(); ❷
} catch (Exception e) {
    log.error("Nieoczekiwany błąd", e);
} finally {
        consumer.close();
}
```

❶ Gdy wszystko jest w porządku, używamy metody `commitAsync`. Jest szybsza, a jeżeli jedno zatwierdzenie się nie powiedzie, następne posłuży jako ponowienie próby.

❷ Jeśli jednak kończymy działanie, nie będzie „następnego zatwierdzenia". Wywołujemy metodę `commitSync()`, ponieważ będzie ona ponawiać próby, aż zatwierdzenie się powiedzie lub wystąpi nieodwracalny błąd.

Zatwierdzanie określonego przesunięcia

Zatwierdzanie ostatniego przesunięcia pozwala zatwierdzać jedynie po zakończeniu przetwarzania każdej partii. Co jednak zrobić, jeżeli wymagane jest częstsze zatwierdzanie? A jeśli wywołanie `poll()` zwróci dużą partię i pożądane będzie zatwierdzanie przesunięcia w środku partii, by uniknąć konieczności ponownego przetwarzania wszystkich tych rekordów w przypadku wystąpienia równoważenia obciążenia? Nie można wywołać po prostu `commitSync()` lub `commitAsync()` — spowoduje to zatwierdzenie ostatniego zwróconego przesunięcia, którego nie udało się jeszcze przetworzyć.

Na szczęście Consumer API pozwala wywołać `commitSync()` i `commitAsync()` oraz przekazać mapę partycji i przesunięć, które mają zostać zatwierdzone. Jeżeli jesteś w trakcie przetwarzania partii rekordów, a ostatni komunikat otrzymany z partycji 3 w temacie `customers` ma offset 5000, możesz wywołać `commitSync()`, aby zatwierdzić offset 5001 dla partycji 3 we wspomnianym temacie. Ponieważ konsument może przetwarzać więcej niż jedną partycję, będziesz musiał śledzić przesunięcia na wszystkich z nich, co zwiększa złożoność kodu.

Tak wygląda zatwierdzanie określonych przesunięć:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

....
Duration timeout = Duration.ofMillis(100);
```

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
            klient = %s, kraj = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value()); ❷
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset()+1, "brak metadanych")); ❸
        if (count % 1000 == 0) ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ To jest mapa, której użyjemy do ręcznego śledzenia przesunięć.

❷ Pamiętaj, że w miejsce `printf` możesz wstawić każde przetwarzanie, które będziesz wykonywał dla konsumowanych rekordów.

❸ Po odczytaniu każdego rekordu aktualizujemy mapę przesunięć o przesunięcie następnego komunikatu, który spodziewamy się przetworzyć. Zatwierdzane przesunięcie powinno być zawsze przesunięciem następnego komunikatu, który aplikacja będzie odczytywać. Od tego miejsca zaczniemy odczytywać następnym razem.

❹ Tutaj decydujemy się zatwierdzać bieżące przesunięcia co 1000 rekordów. W swojej aplikacji możesz zatwierdzać np. na podstawie czasu lub zawartości rekordów.

❺ Zdecydowaliśmy się wywołać `commitAsync()` (bez wywołania zwrotnego, dlatego drugi parametr ma wartość `null`), ale wywołanie `commitSync()` również będzie tu całkowicie poprawne. Oczywiście kiedy zatwierdzasz określone przesunięcia, nadal musisz wykonać całą obsługę błędów, którą pokazaliśmy w poprzednich przykładach.

Nasłuchiwacze równoważenia obciążenia

Jak wspomnieliśmy w poprzednim podrozdziale poświęconym zatwierdzaniu przesunięć, przed zakończeniem działania oraz przed równoważeniem obciążenia partycji konsument będzie chciał wykonywać pewne prace porządkowe.

Jeśli wiesz, że konsument wkrótce straci własność partycji, będziesz chciał zatwierdzić przesunięcia ostatniego przetworzonego zdarzenia. Prawdopodobnie będziesz musiał zamknąć także uchwyty plików, połączenia z bazą danych itp.

Consumer API umożliwia uruchamianie własnego kodu, gdy partycje są dodawane do konsumenta lub usuwane z niego. Wymaga to przekazania obiektu nasłuchiwacza `ConsumerRebalanceListener` podczas wywoływania metody `subscribe()`, którą omówiliśmy wcześniej. Klasa `ConsumerRebalan`
`↪ceListener` ma trzy metody, które możesz zaimplementować:

`public void onPartitionsAssigned(Collection<TopicPartition> partitions)`

Wywoływana po ponownym przypisaniu partycji do konsumenta, ale zanim konsument rozpocznie konsumowanie komunikatów. W tym miejscu przygotowujesz lub ładujesz dowolny

stan, którego chcesz użyć z partycją, poszukujesz poprawnych przesunięć, jeśli zachodzi taka potrzeba, itp. Wszelkie wykonywane tu przygotowania powinny zakończyć się przed upływem limitu czasu `max.poll.timeout.ms`, by konsument mógł pomyślnie dołączyć do grupy.

`public void onPartitionsRevoked(Collection<TopicPartition> partitions)`

Wywoływana, gdy konsument musi zrezygnować z posiadanych partycji w wyniku równoważenia obciążenia lub kończenia działania konsumenta. W typowym przypadku używania algorytmu gorliwego równoważenia obciążenia metoda ta jest wywoływana przed rozpoczęciem równoważenia i po zakończeniu konsumowania komunikatów przez konsument. Jeśli używany jest algorytm kooperacyjnego równoważenia obciążenia, metoda ta jest wywoływana na końcu równoważenia obciążenia i jedynie z podzbiorem partycji, z których konsument musi zrezygnować. W tym miejscu powinieneś zatwierdzać przesunięcia, by następny właściciel partycji wiedział, od czego zacząć.

`public void onPartitionsLost(Collection<TopicPartition> partitions)`

Wywoływana tylko przy zastosowaniu algorytmu kooperacyjnego równoważenia obciążenia i jedynie w wyjątkowych przypadkach, gdy partycje zostały przypisane innym konsumentom bez uprzedniego anulowania przypisań przez algorytm równoważenia (w normalnych sytuacjach wywoływana jest metoda `onPartitionsRevoked()`). W tym miejscu czyścisz stan lub zasoby używane z tymi partycjami. Zwróć uwagę, że należy to robić ostrożnie — nowy właściciel partycji mógł już zapisać własny stan i lepiej unikać konfliktów. Jeśli nie zaimplementujesz tej metody, zamiast niej wywoływana będzie metoda `onPartitionsRevoked()`.

Jeżeli używasz algorytmu kooperacyjnego równoważenia obciążenia, powinieneś pamiętać o kilku kwestiach:

- Metoda `onPartitionsAssigned()` będzie wywoływana przy każdym równoważeniu obciążenia jako sposób powiadamiania o nim konsumenta. Jeśli jednak danemu konsumentowi nie będą przypisywane żadne nowe partycje, metoda będzie wywoływana z pustą kolekcją.
- Metoda `onPartitionsRevoked()` będzie wywołana w normalnych warunkach równoważenia obciążenia, ale tylko wtedy, gdy dany konsument zrezygnuje z własności partycji. Nie będzie wywoływana z pustą kolekcją.
- Metoda `onPartitionsLost()` będzie wywoływana w wyjątkowych warunkach równoważenia obciążenia, a partycje z kolekcji będą już wtedy miały nowych właścicieli.

Jeśli zaimplementujesz wszystkie trzy metody, będziesz miał gwarancję, że podczas normalnego równoważenia metoda `onPartitionsAssigned()` będzie wywoływana przez nowego właściciela ponownie przypisywanych partycji dopiero po tym, jak poprzedni właściciel wywoła `onPartitionsRevoked()` i zrezygnuje z własności.

W poniższym przykładzie pokazaliśmy, jak używać `onPartitionsRevoked()` do zatwierdzania przesunięć przed utratą własności partycji:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
Duration timeout = Duration.ofMillis(100);

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
          partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
```

```
            System.out.println("Podczas rebalansowania utracono partycje. " +
                "Zatwierdzanie bieżących przesunięć:" + currentOffsets);
              consumer.commitSync(currentOffsets); ❸
        }
    }

    try {
        consumer.subscribe(topics, new HandleRebalance()); ❹

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(timeout);
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                      klient = %s, kraj = %s\n",
                      record.topic(), record.partition(), record.offset(),
                      record.key(), record.value());
                currentOffsets.put(
                    new TopicPartition(record.topic(), record.partition()),
                    new OffsetAndMetadata(record.offset()+1, null));
            }
            consumer.commitAsync(currentOffsets, null);
        }
    } catch (WakeupException e) {
        // Ignorujemy, ponieważ kończymy działanie
    } catch (Exception e) {
        log.error("Nieoczekiwany błąd", e);
    } finally {
        try {
            consumer.commitSync(currentOffsets);
        } finally {
            consumer.close();
            System.out.println("Zamknęliśmy konsument i zakończyliśmy");
        }
    }
```

❶ Zaczynamy od zaimplementowania nasłuchiwacza `ConsumerRebalanceListener`.

❷ W tym przykładzie nie musimy nic robić, gdy otrzymamy nową partycję; zaczynamy po prostu konsumować komunikaty.

❸ Jednak gdy grozi nam utrata partycji z powodu równoważenia obciążenia, musimy zatwierdzić przesunięcia. Robimy to dla wszystkich partycji, a nie tylko dla tych, które zaraz utracimy — przesunięcia dotyczą zdarzeń, które zostały już przetworzone, więc nie ma w tym nic złego. Ponadto używamy `commitSync()`, aby przed kontynuowaniem równoważenia obciążenia upewnić się, że przesunięcia zostały zatwierdzone.

❹ Najważniejsza część: przekazujemy `ConsumerRebalanceListener` do metody `subscribe()`, aby została wywołana przez konsument.

Konsumowanie rekordów z określonymi przesunięciami

Nauczyłeś się już, jak używać metody `poll()` do rozpoczęcia konsumowania komunikatów od ostatniego zatwierdzonego przesunięcia dla każdej partycji i kontynuowania przetwarzania po kolei wszystkich komunikatów. Czasami jednak potrzebujesz zacząć odczytywanie z innym przesunięciem.

Kafka oferuje różne metody, które powodują, że następne wywołanie `poll()` rozpoczyna konsumowanie z innym przesunięciem.

Jeśli chcesz zacząć odczytywanie wszystkich komunikatów od początku partycji albo przeskoczyć do końca partycji i zacząć przetwarzać tylko nowe komunikaty, możesz skorzystać ze specjalnie do tego celu przeznaczonych interfejsów API: `seekToBeginning(Collection<TopicPartition> tp)` i `seekToEnd(Collection<TopicPartition> tp)`.

API Kafki pozwala również wyszukać określone przesunięcie. Z tej funkcjonalności można korzystać na wiele sposobów, np. pozostająca w tyle aplikacja, w której istotny jest czas, może przeskoczyć do przodu o kilka rekordów, albo konsument zapisujący dane w pliku może zostać zresetowany do określonego punktu w czasie w celu odzyskania danych w przypadku utraty pliku.

Oto przykład, jak ustawić bieżące przesunięcie na wszystkich partycjach na rekordy, które zostały utworzone w określonym momencie:

```
Long oneHourEarlier = Instant.now().atZone(ZoneId.systemDefault())
          .minusHours(1).toEpochSecond();
Map<TopicPartition, Long> partitionTimestampMap = consumer.assignment()
        .stream()
        .collect(Collectors.toMap(tp -> tp, tp -> oneHourEarlier)); ❶
Map<TopicPartition, OffsetAndTimestamp> offsetMap
        = consumer.offsetsForTimes(partitionTimestampMap); ❷

for(Map.Entry<TopicPartition,OffsetAndTimestamp> entry: offsetMap.entrySet()) {
    consumer.seek(entry.getKey(), entry.getValue().offset()); ❸
}
```

❶ Tworzymy mapowanie wszystkich partycji przypisanych do danego konsumenta (za pośrednictwem `consumer.assignment()`) na znacznik czasu, do którego chcemy przywrócić konsumenty.

❷ Następnie pobieramy przesunięcia, które były aktualne w określonych momentach. Ta metoda wysyła do brokera żądanie, w którym do zwrócenia odpowiednich przesunięć używany jest indeks znacznika czasu.

❸ Na koniec resetujemy przesunięcie na każdej partycji do tego, które zostało zwrócone w poprzednim kroku.

Wychodzenie z pętli

Wcześniej w tym rozdziale, gdy pisaliśmy o pętli odpytywania, stwierdziliśmy, że nie należy się przejmować tym, iż konsumenty odpytują w pętli nieskończonej, a omówienie czystego wychodzenia z pętli odłożyliśmy na później. Porozmawiajmy więc teraz o tym, jak czysto wyjść z pętli odpytywania.

Gdy zdecydujesz się zamknąć konsument i będziesz chciał natychmiast wyjść z pętli, nawet jeśli konsument będzie czekał na zakończenie długiego wywołania `poll()`, będziesz potrzebować kolejnego wątku do wywołania metody `consumer.wakeup()`. Jeżeli uruchamiasz pętlę konsumenta w głównym

wątku, możesz to zrobić za pomocą zaczepu zamknięcia (ShutdownHook). Zwróć uwagę, że consu
↪mer.wakeup() jest jedyną metodą konsumenta, którą można bezpiecznie wywoływać z innego wątku. Wywołanie wakeup spowoduje zakończenie działania metody poll() z wyjątkiem Wakeup
↪Exception, a jeśli metoda consumer.wakeup() zostanie wywołana, gdy wątek nie oczekuje na rezultaty odpytywania, wyjątek zostanie zgłoszony przy wywołaniu poll() w następnej iteracji. Wakeup
↪Exception nie musi być obsługiwany, ale przed wyjściem z wątku należy wywołać metodę consu
↪mer.close(). Zamknięcie konsumenta spowoduje zatwierdzenie przesunięć, jeżeli zachodzi taka potrzeba, oraz wysłanie koordynatorowi grupy komunikatu, że konsument opuszcza grupę. Koordynator konsumentów natychmiast uruchomi równoważenie obciążenia i nie trzeba będzie czekać na upłynięcie limitu czasu sesji, by partycje zamykanego konsumenta zostały przypisane do innego konsumenta w grupie.

W poniższym listingu pokazaliśmy, jak będzie wyglądał kod wyjścia, jeśli konsument działa w głównym wątku aplikacji. Ten przykład jest nieco skrócony, ale możesz zobaczyć go w całości na GitHubie (*http://bit.ly/2u47e9A*):

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Rozpoczęcie procesu wyjścia...");
        consumer.wakeup(); ❶
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

...
Duration timeout = Duration.ofMillis(10000); ❷

try {
    // Zapętlamy do wciśnięcia Ctrl+C, a przy wyjściu z pętli zaczep zamknięcia wykona czyszczenie
    while (true) {
        ConsumerRecords<String, String> records =
            movingAvg.consumer.poll(timeout);
        System.out.println(System.currentTimeMillis() +
            "--  oczekiwanie na dane...");
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("przesunięcie = %d, klucz = %s, wartość = %s\n",
                record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Zatwierdzanie przesunięcia na pozycji:" +
                consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
    // Przy zamykaniu ignorujemy ❸
} finally {
    consumer.close(); ❹
    System.out.println("Konsument zamknięty - gotowe");
}
```

1. `ShutdownHook` jest uruchamiany w osobnym wątku, stąd jedyną bezpieczną czynnością, jaką możesz wykonać, jest wywołanie `wakeup`, by wyjść z pętli `poll`.
2. Wyjątkowo długi limit czasu odpytywania. Jeśli pętla odpytywania jest odpowiednio krótka i możesz trochę poczekać przed wyjściem z niej, nie musisz wywoływać `wakeup` — wystarczy sprawdzanie niepodzielnej wartości logicznej w każdej iteracji. Długie limity czasu odpytywania są przydatne podczas konsumowania tematów o niskiej przepływności; dzięki temu klient zużywa mniej procesora do ciągłego zapętlania, gdy broker nie ma żadnych nowych danych do zwrócenia.
3. Kolejny wątek wywołujący `wakeup` spowoduje, że `poll` zgłosi wyjątek `WakeupException`. Powinieneś wychwytywać ten wyjątek, by mieć pewność, że aplikacja nie zostanie nieoczekiwanie zamknięta, nie musisz jednak niczego z nim robić.
4. Na koniec upewnij się, że konsument został czysto zamknięty.

Deserializatory

Jak pisaliśmy w poprzednim rozdziale, producenty Kafki wymagają **serializatorów** (ang. *serializer*) do konwersji obiektów na tablice bajtów, które są następnie wysyłane do Kafki. Analogicznie konsumenty Kafki wymagają **deserializatorów** (ang. *deserializer*) do konwersji otrzymanych z Kafki tablic bajtów na obiekty Javy. W poprzednich przykładach założyliśmy, że zarówno klucz, jak i wartość każdego komunikatu są łańcuchami znaków, i w konfiguracji konsumenta korzystaliśmy z domyślnego deserializatora `StringDeserializer`.

W rozdziale 3. poświęconym producentowi Kafki pokazaliśmy, jak serializować typy niestandardowe. Dowiedziałeś się ponadto, jak używać platformy Avro i serializatorów `AvroSerializer` do generowania obiektów Avry na podstawie definicji schematów i serializowania tych obiektów podczas produkowania komunikatów do Kafki. Przyjrzymy się teraz tworzeniu niestandardowych deserializatorów dla własnych obiektów oraz korzystaniu z Avry i jego deserializatorów.

Powinno być oczywiste, że serializator stosowany do produkowania zdarzeń do Kafki musi odpowiadać deserializatorowi, który będzie używany podczas konsumowania tych zdarzeń. Serializacja za pomocą `IntSerializer`, a następnie deserializacja za pomocą `StringDeserializer` nie skończy się dobrze. Oznacza to, że jako programista musisz śledzić, które serializatory zostały wykorzystane do zapisywania w poszczególnych tematach, i upewnić się, że każdy temat zawiera tylko dane, które mogą być interpretowane przez używane przez Ciebie deserializatory. Jedna z korzyści serializacji i deserializacji za pomocą Avry i rejestru schematów polega na tym, że `AvroSerializer` może zagwarantować zgodność ze schematem tematu wszystkich zapisywanych w nim danych, co oznacza, że będzie można deserializować te dane odpowiednim deserializatorem i schematem. Wszelkie błędy w zgodności — po stronie producenta lub konsumenta — zostaną łatwo wychwycone z wygenerowaniem odpowiedniego komunikatu o błędzie i nie trzeba będzie debugować tablic bajtów pod kątem błędów serializacji.

Zaczniemy od pokazania, jak napisać niestandardowy deserializator, mimo że jest to mniej powszechna metoda, a potem przejdziemy do przykładu użycia Avry do deserializacji kluczy i wartości komunikatów.

Niestandardowe deserializatory

Wykorzystajmy ten sam niestandardowy obiekt, który serializowaliśmy w rozdziale 3., i napiszmy dla niego deserializator. Oto ten obiekt:

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

Niestandardowy deserializator będzie wyglądał następująco:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> { ❶

    @Override
    public void configure(Map configs, boolean isKey) {
        // Nie trzeba niczego konfigurować
    }

    @Override
    public Customer deserialize(String topic, byte[] data) {
        int id;
        int nameSize;
        String name;

        try {
            if (data == null)
                return null;
            if (data.length < 8)
                throw new SerializationException("Rozmiar danych otrzymanych " +
                    "przez deserializator jest mniejszy niż oczekiwany");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            nameSize = buffer.getInt();

            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            name = new String(nameBytes, "UTF-8");
```

```
            return new Customer(id, name); ❷

        } catch (Exception e) {
            throw new SerializationException("Błąd podczas deserializacji " +
                "byte[] do klienta " + e);
        }
    }

    @Override
    public void close() {
        // Nie trzeba niczego zamykać
    }
}
```

❶ Konsument również potrzebuje implementacji klasy `Customer`, a zarówno klasa, jak i serializator muszą być zgodne w aplikacjach produkujących i konsumujących. W dużej organizacji, w której wiele konsumentów i producentów współdzieli dostęp do danych, może to być problematyczne.

❷ Odwracamy logikę serializatora — pobieramy z tablicy bajtów identyfikator klienta i nazwę, a potem używamy ich do skonstruowania potrzebnego obiektu.

Kod konsumenta korzystającego z tego deserializatora będzie wyglądał mniej więcej tak:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    CustomerDeserializer.class.getName());

KafkaConsumer<String, Customer> consumer =
    new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout);
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("Id bieżącego klienta: " +
            record.value().getID() + " oraz
            nazwa bieżącego klienta: " + record.value().getName());
    }
    consumer.commitSync();
}
```

Jeszcze raz podkreślamy, że implementowanie niestandardowego serializatora i deserializatora nie jest zalecane. Powoduje to ścisłe powiązanie producentów i konsumentów, a ponadto jest kruche i podatne na błędy. Lepszym rozwiązaniem jest użycie standardowego formatu komunikatów, takiego jak JSON, Thrift, Protobuf lub Avro.

Zobaczymy teraz, jak stosować deserializatory Avry z konsumentem Kafki. Więcej informacji na temat platformy Apache Avro, jej schematów oraz zapewniania zgodności ze schematami znajdziesz w rozdziale 3.

Stosowanie deserializacji Avry z konsumentem Kafki

Załóżmy, że korzystamy w Avrze z implementacji klasy `Customer`, która została pokazana w rozdziale 3. Aby konsumować te obiekty z Kafki, należy zaimplementować aplikację konsumującą:

```
Duration timeout = Duration.ofMillis(100);
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("specific.avro.reader","true");
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts";
KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Czytanie tematu:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(timeout); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Nazwa bieżącego klienta to: " +
            record.value().getName());
    }
    consumer.commitSync();
}
```

❶ Do deserializacji komunikatów Avry wykorzystujemy `KafkaAvroDeserializer`.

❷ `schema.registry.url` to nowy parametr. Wskazuje lokalizacje przechowywania schematów. W ten sposób konsument może użyć schematu zarejestrowanego przez producenta do deserializacji wiadomości.

❸ Jako typ wartości rekordu podajemy wygenerowaną klasę `Customer`.

❹ `record.value()` jest instancją `Customer` i możemy używać jej zgodnie z definicją tej klasy.

Samodzielny konsument — dlaczego i jak korzystać z konsumenta bez grupy?

Do tej pory pisaliśmy o grupach konsumentów, w których partycje są automatycznie przypisywane do konsumentów i automatycznie rebalansowane przy dodawaniu konsumentów do grupy lub usuwaniu ich z niej. Zazwyczaj właśnie takie zachowanie jest wymagane, ale w niektórych przypadkach potrzebujemy czegoś znacznie prostszego. Czasami wiemy, że mamy jednego konsumenta, który zawsze musi odczytywać dane ze wszystkich partycji z tematu lub z określonej partycji z tematu. W takiej sytuacji nie ma powodu do korzystania z grup lub przeprowadzania równoważenia obciążenia — wystarczy przypisać temat i (lub) partycje charakterystyczne dla danego konsumenta, konsumować komunikaty i od czasu do czasu zatwierdzać przesunięcia (chociaż do

zatwierdzania przesunięć i tak trzeba skonfigurować `group.id`, bez wywoływania `subscribe` konsument nie dołączy do żadnej grupy).

Kiedy dokładnie wiesz, które partycje powinien odczytywać konsument, nie *subskrybujesz* tematu — zamiast tego *przypisujesz* sobie kilka partycji. Konsument może subskrybować tematy (i należeć do grupy konsumentów) lub przypisywać sobie partycje, ale nie może robić obu rzeczy naraz.

Poniżej pokazaliśmy, w jaki sposób konsument może przypisać sobie wszystkie partycje określonego tematu i konsumować z nich:

```
Duration timeout = Duration.ofMillis(100);
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(timeout);

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("temat = %s, partycja = %s, przesunięcie = %d,
                klient = %s, kraj = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

❶ Zaczynamy od zapytania klastra o partycje dostępne w danym temacie. Jeżeli planujesz przetwarzać tylko określoną partycję, możesz pominąć tę część.

❷ Gdy już wiemy, których partycji potrzebujemy, wywołujemy metodę `assign()` z ich listą.

Poza brakiem równoważenia obciążenia i koniecznością ręcznego wyszukania partycji, wszystko inne działa jak zwykle. Pamiętaj, że jeśli ktoś doda do tematu nowe partycje, konsument nie zostanie o tym powiadomiony. Będziesz musiał to obsłużyć przez okresowe sprawdzanie `consumer.parti`
`↪tionsFor()` lub restartowanie aplikacji po każdorazowym dodaniu partycji.

Podsumowanie

Ten rozdział rozpoczęliśmy od szczegółowego omówienia grup konsumentów Kafki i sposobu, w jaki umożliwiają one wielu konsumentom dzielenie się pracą polegającą na odczytywaniu zdarzeń z tematów. Teoretyczne rozważania poparliśmy praktycznym przykładem konsumenta subskrybującego określony temat i nieustannie odczytującego zdarzenia. Następnie przyjrzeliśmy się najważniejszym parametrom konfiguracji konsumenta i ich wpływowi na jego zachowanie. Dużą część rozdziału poświęciliśmy na omówienie przesunięć i metod ich śledzenia przez konsumenty. Zrozumienie, jak konsumenty zatwierdzają przesunięcia, ma kluczowe znaczenie przy pisaniu

niezawodnych konsumentów, dlatego dogłębnie wyjaśniliśmy to zagadnienie. Potem opisaliśmy dodatkowe elementy interfejsów API konsumentów, obsługę równoważenia obciążenia i zamykanie konsumenta.

Na koniec omówiliśmy deserializatory używane przez konsumenty do przekształcania bajtów przechowywanych w Kafce w obiekty Javy, które aplikacje mogą przetwarzać. Nieco szerzej przyjrzeliśmy się deserializatorom Avry i chociaż to tylko jeden z możliwych do zastosowania typów deserializatorów, są one najczęściej używane z Kafką.

Kafka: gwarancja niezawodności korporacyjnych aplikacji!

Każda aplikacja korporacyjna tworzy dane. Ich przenoszenie i przetwarzanie jest równie ważne jak same dane. Platforma Apache Kafka została zbudowana właśnie w tym celu: aby umożliwić płynną obsługę strumieni zdarzeń czasu rzeczywistego. Umożliwia to architektom nie tylko łączenie aplikacji z systemami danych, ale również budowanie niestandardowych aplikacji, które same w sobie wyzwalają strumienie danych. Okazuje się, że architektura skupiona wokół strumieni zdarzeń otwiera zupełnie nowy rozdział projektowania dużych systemów.

Dzięki temu praktycznemu przewodnikowi komfortowo rozpoczniesz pracę z platformą streamingową Apache Kafka. Poznasz najlepsze praktyki w zakresie jej wdrażania i konfiguracji, aby zapewnić sobie możliwość strumieniowego przetwarzania dużych ilości danych. Zaznajomisz się z AdminClient API Kafki, mechanizmem transakcji i z nowymi funkcjonalnościami zabezpieczeń. W książce znajdziesz szczegółowe instrukcje, w jaki sposób wdrażać klastry produkcyjne Kafki, pisać niezawodne mikrousługi oparte na zdarzeniach i budować skalowalne aplikacje przetwarzania strumieniowego. Opisano w niej też gwarancje niezawodności, kluczowe interfejsy API i szczegóły architektury, w tym protokół replikacji, kontroler i warstwę pamięci masowej.

Najciekawsze zagadnienia:

- wdrażanie i konfigurowanie Kafki w praktyce
- niezawodne dostarczanie danych
- budowanie potoków danych i aplikacji
- monitorowanie, dostrajanie i utrzymywanie działania Kafki w środowisku produkcyjnym
- wskaźniki pomiarów operacyjnych Kafki
- Kafka w systemach przetwarzania strumieniowego

Gwen Shapira pracuje w firmie Confluent, zarządza zespołem Kafki dla natywnej chmury. Odpowiada za wydajność, elastyczność i wielodzierżawność Kafki.
Todd Palino jest głównym inżynierem SRE w firmie LinkedIn. Odpowiada za planowanie pojemności zasobów i wydajności.
Rajini Sivaram jest głównym inżynierem w firmie Confluent. Rozwija replikację międzyklastrową i funkcjonalności bezpieczeństwa dla Kafki.
Krit Petty jest menedżerem SRE do spraw Kafki w firmie LinkedIn.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9320-2
9 788328 393202
Cena: 99,00 zł