
APACHE
kafka®

UNIT

Kafka Consumers

Consumers

- Le applicazioni che devono leggere i dati da Kafka utilizzano un KafkaConsumer per iscriversi agli argomenti di Kafka e ricevere messaggi da questi argomenti.

- La lettura dei dati da Kafka è leggermente diversa dalla lettura dei dati da altri sistemi di messaggistica e sono coinvolti alcuni concetti e idee unici.

- È difficile capire come utilizzare l'API consumer senza prima capire questi concetti.

- Inizieremo spiegando alcuni dei concetti importanti, quindi esamineremo alcuni esempi che mostrano i diversi modi in cui le API dei consumer possono essere utilizzate per implementare applicazioni con requisiti diversi

Kafka Consumer Concepts - Consumers e Consumer Groups

- Supponiamo di avere un'applicazione che deve leggere i messaggi da un argomento Kafka, eseguire alcune convalide su di essi e scrivere i risultati in un altro archivio dati.

- In questo caso, l'applicazione creerà un oggetto consumer, si iscriverà all'argomento appropriato e inizierà a ricevere messaggi, convalidandoli e scrivendo i risultati.

- Questo potrebbe funzionare bene per un po ', ma cosa succede se la velocità con cui i producer scrivono messaggi sull'argomento supera la velocità con cui l'applicazione può convalidarli?

- Se ci si limita a un singolo consumer che legge ed elabora i dati, la tua applicazione potrebbe rimanere sempre più indietro, incapace di tenere il passo con la velocità dei messaggi in arrivo.

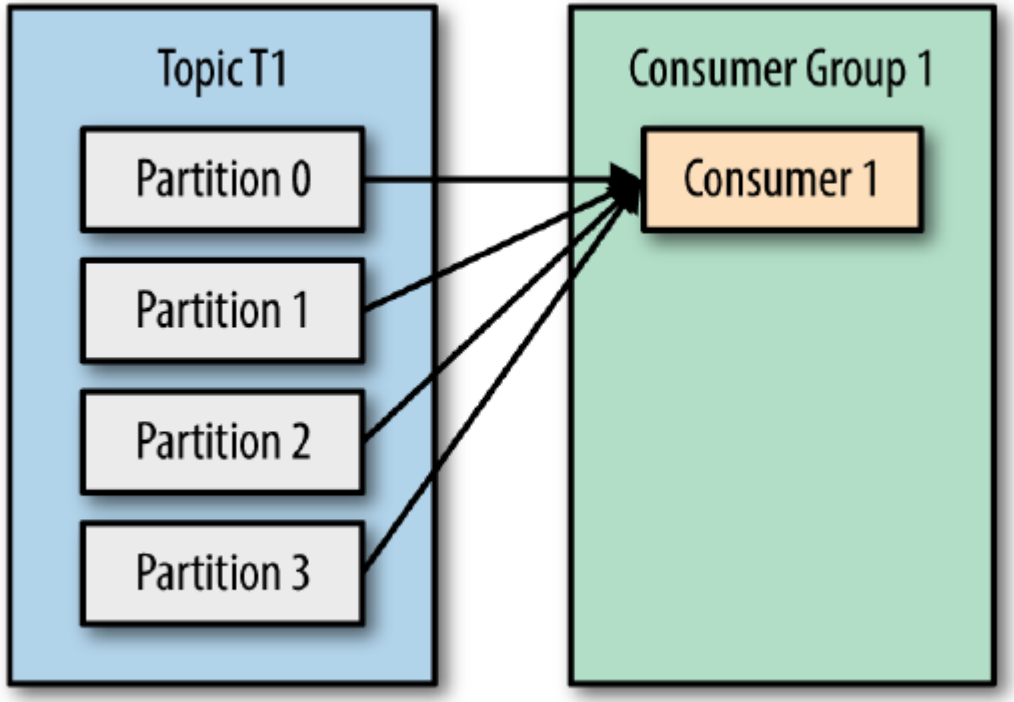
- Ovviamente è necessario ridimensionare il consumo in base agli argomenti.

- Proprio come più producer possono scrivere sullo stesso argomento, dobbiamo consentire a più consumer di leggere dallo stesso argomento, suddividendo i dati tra loro.

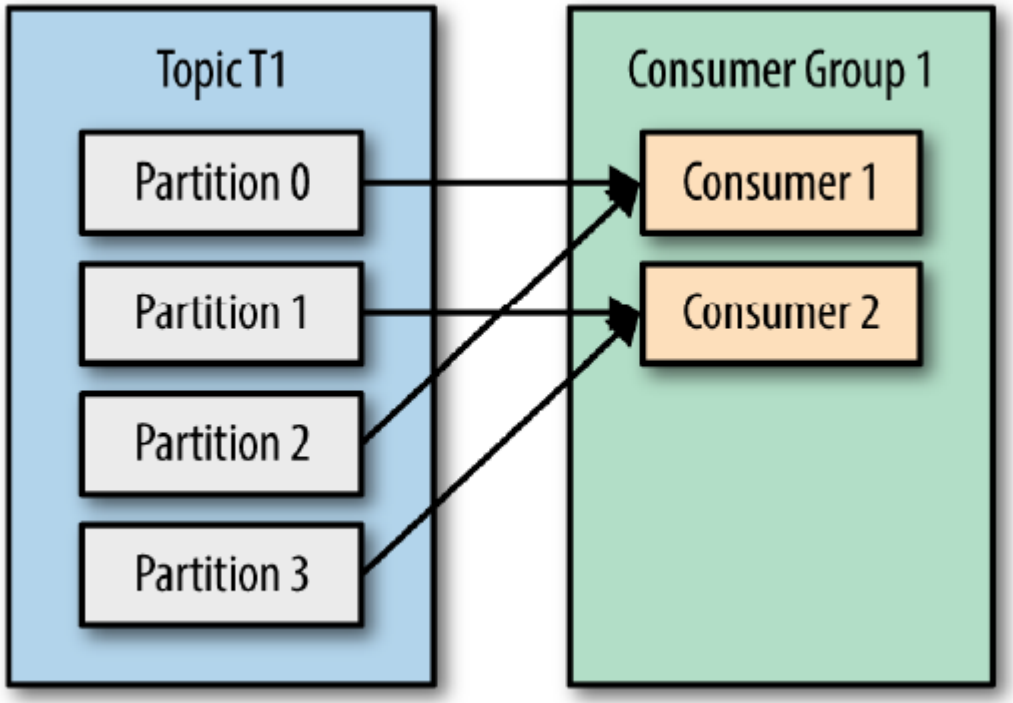
- I consumer di Kafka fanno generalmente parte di un gruppo di consumer.

- Quando più consumer sono «subscribed» a un argomento e appartengono allo stesso gruppo di consumer, ogni consumer nel gruppo riceverà messaggi da un sottoinsieme diverso delle partizioni nell'argomento.

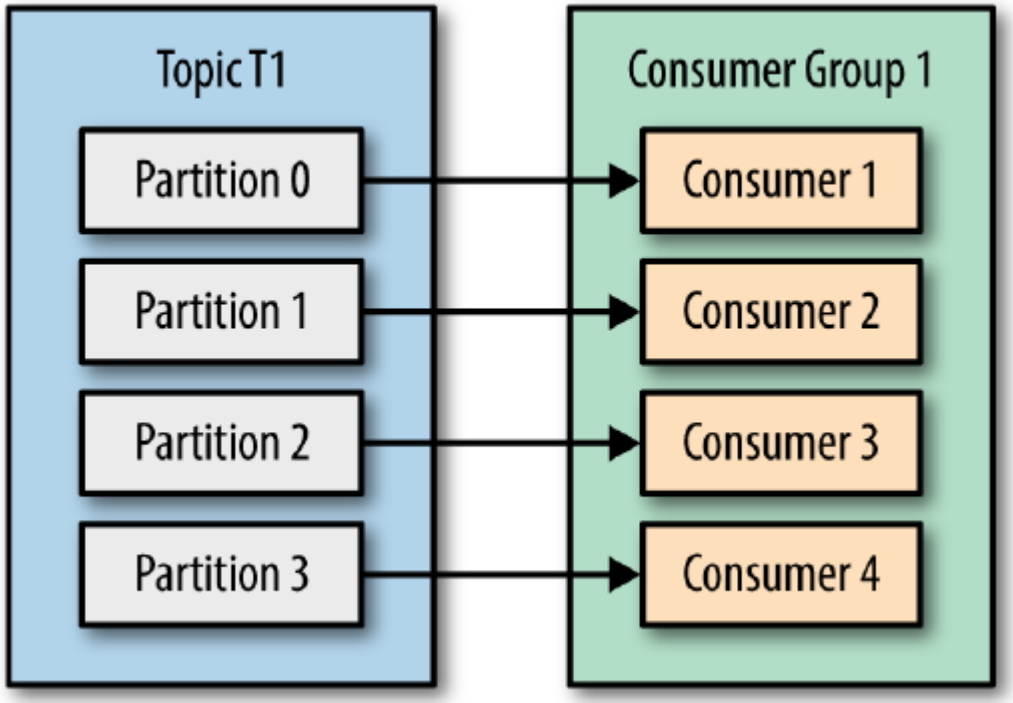
- Prendiamo l'argomento T1 con quattro partizioni. Supponiamo ora di aver creato un nuovo consumer C1 (unico consumer del gruppo G1) e di usarlo per iscriversi all'argomento T1. Il consumer C1 riceverà tutti i messaggi da tutte e quattro le partizioni t1.



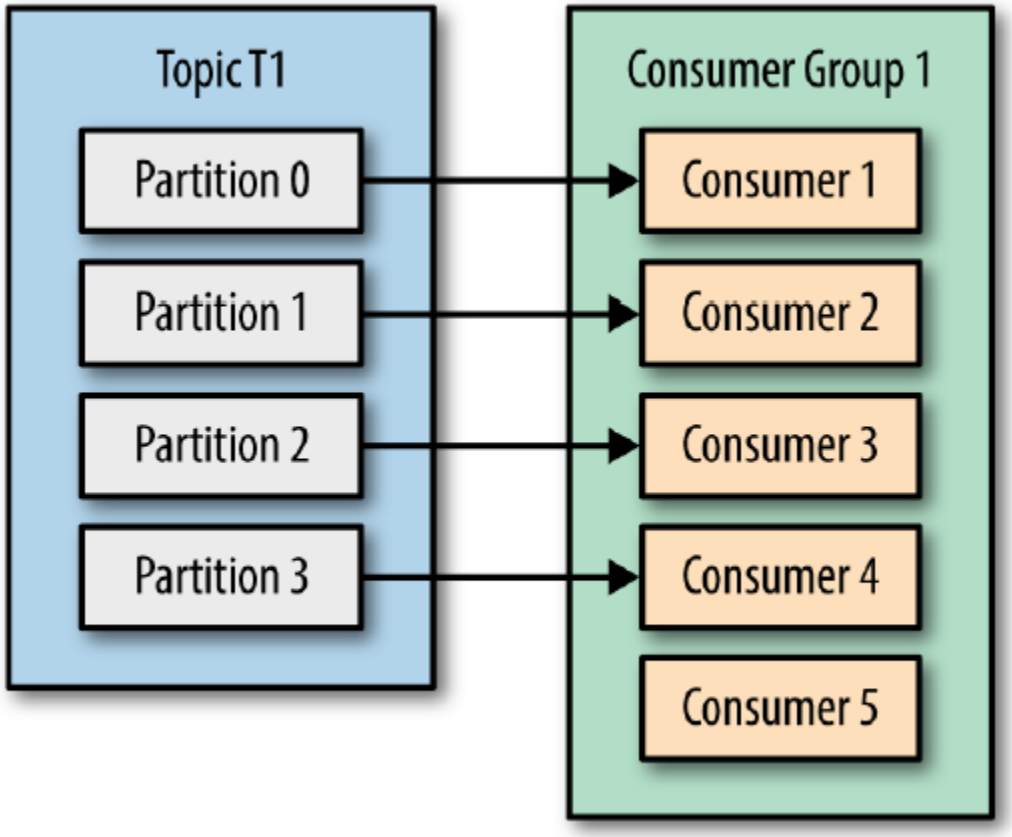
- Se aggiungiamo un altro consumer, C2, al gruppo G1, ogni consumer riceverà solo messaggi da due partizioni. Forse i messaggi dalle partizioni 0 e 2 vanno a C1 e i messaggi dalle partizioni 1 e 3 vanno al consumer C2.



- Se G1 ha quattro utenti, ognuno leggerà i messaggi da una singola partizione.



- Se aggiungiamo più consumer a un singolo gruppo con un singolo argomento rispetto alle partizioni, alcuni dei consumer resteranno inattivi e non riceveranno alcun messaggio.



- Il principale modo per ridimensionare il consumo di dati da un argomento di Kafka è aggiungere più consumer a un gruppo di consumer.

- È comune per i consumer Kafka eseguire operazioni ad alta latenza come scrivere su un database o un calcolo che richiede tempo sui dati.

- In questi casi, un singolo consumer non può probabilmente tenere il passo con i flussi di dati su un argomento e l'aggiunta di più consumer che condividono il carico avendo ogni consumer solo un sottoinsieme delle partizioni e dei messaggi è il nostro principale metodo di ridimensionamento.

- Questa è una buona ragione per creare argomenti con un gran numero di partizioni: consente di aggiungere più utenti quando aumenta il carico.

- E’ da tener presente che non ha senso aggiungere consumer in numero superiore alle partizioni presenti in un argomento: alcuni dei consumer saranno semplicemente inattivi.

Kafka Consumer Concepts - Consumers e Consumer Groups

- Oltre ad aggiungere utenti per ridimensionare una singola applicazione, è molto comune avere più applicazioni che devono leggere i dati dallo stesso argomento.

- In effetti, uno dei principali obiettivi di progettazione di Kafka era quello di rendere disponibili i dati prodotti sugli argomenti di Kafka per molti casi d'uso all'interno dell'organizzazione.

- In questi casi, vogliamo che ogni applicazione riceva tutti i messaggi, anziché solo un sottoinsieme.

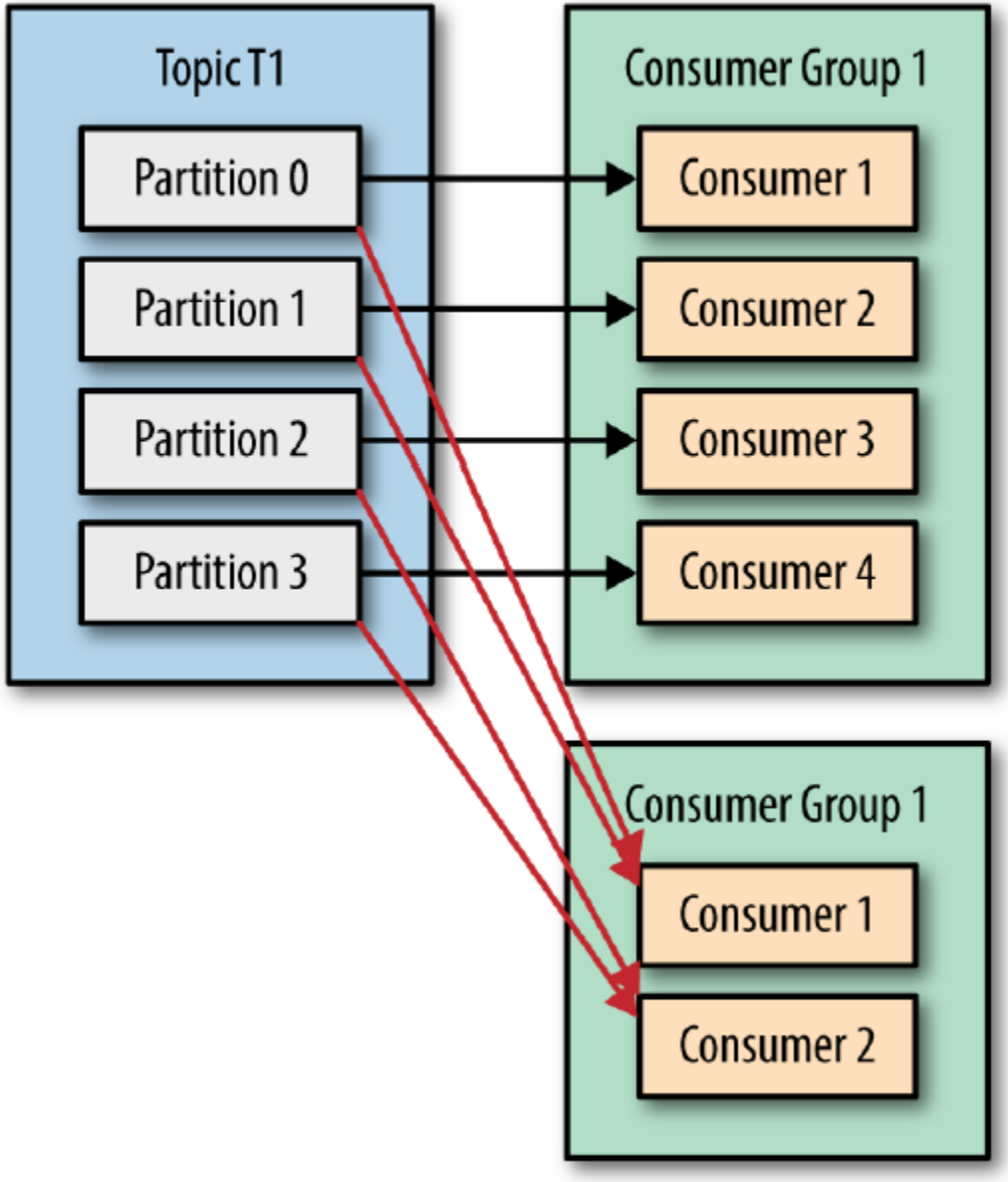
- Per assicurarsi che un'applicazione riceva tutti i messaggi in un argomento, assicurarsi che l'applicazione abbia il proprio gruppo di consumer.

- A differenza di molti sistemi di messaggistica tradizionali, Kafka si adatta a un gran numero di consumer e gruppi di consumer senza ridurre le prestazioni.

- Nell'esempio precedente, se aggiungiamo un nuovo gruppo di consumer G2 con un singolo consumer, questo consumer riceverà tutti i messaggi nell'argomento T1 indipendentemente da ciò che G1 sta facendo.

- G2 può avere più di un singolo consumer, nel qual caso ognuno riceverà un sottoinsieme di partizioni, proprio come abbiamo mostrato per G1, ma G2 nel suo insieme riceverà comunque tutti i messaggi indipendentemente dagli altri gruppi di consumer.

Kafka Consumer Concepts - Consumers e Consumer Groups



- Per riassumere, si crea un nuovo gruppo di consumer per ogni applicazione che necessita di tutti i messaggi di uno o più argomenti.

- Aggiungete i consumer a un gruppo di consumer esistente per ridimensionare la lettura e l'elaborazione dei messaggi dagli argomenti, così ogni consumer aggiuntivo in un gruppo riceverà solo un sottoinsieme dei messaggi

- I consumer di un gruppo di consumer condividono la proprietà delle partizioni negli argomenti a cui si abbonano.

- Quando aggiungiamo un nuovo consumer al gruppo, questo inizia a consumare messaggi da partizioni precedentemente consumate da un altro consumer.

- La stessa cosa accade quando un consumer si spegne o si blocca; lascia il gruppo e le partizioni che consumava saranno consumate da uno dei restanti consumer.

- La riassegnazione delle partizioni ai consumer si verifica anche quando vengono modificati gli argomenti che il gruppo di consumer sta consumando (ad esempio, se un amministratore aggiunge nuove partizioni).

- Lo spostamento della proprietà della partizione da un consumer a un altro si chiama riequilibrio.

- I riequilibri sono importanti perché forniscono al gruppo di consumer un'alta disponibilità e scalabilità (che ci consente di aggiungere e rimuovere facilmente e in modo sicuro i consumer), ma nel normale corso degli eventi sono abbastanza indesiderabili.

- Durante un riequilibrio, i consumer non possono consumare messaggi, quindi un riequilibrio è sostanzialmente una breve finestra di indisponibilità dell'intero gruppo di consumer.

- Inoltre, quando le partizioni vengono spostate da un consumer a un altro, il consumer perde il suo stato attuale; se stava memorizzando nella cache dei dati, dovrà aggiornare le sue cache, rallentando l'applicazione fino a quando il consumer non ripristina il suo stato.

- Il modo in cui i consumer mantengono l'appartenenza a un gruppo di consumer (e la proprietà delle partizioni assegnate loro) è inviando «ping» a un broker Kafka designato come coordinatore del gruppo (questo broker può essere diverso per i diversi gruppi di consumer).

- Finché il consumer invia «ping» a intervalli regolari, si presume che sia vivo attivo e che elabori i messaggi dalle sue partizioni.

- I «ping» vengono inviati quando i consumer sondano il broker e quando questi committa i record che ha consumato.

- Se il consumer smette di inviare i «ping» abbastanza a lungo, la sua sessione scadrà e il coordinatore del gruppo lo considererà morto e scatenerà un riequilibrio.

- Se un consumer si arresta in modo anomalo e interrompe l'elaborazione dei messaggi, il coordinatore del gruppo impiegherà alcuni secondi senza «ping» per decidere che è inattivo e attivare il riequilibrio. Durante quei secondi, nessun messaggio verrà elaborato dalle partizioni di proprietà del consumer inattivo.

- Quando si chiude un consumer in modo pulito, il consumer informerà il coordinatore del gruppo che sta per andarsene e il coordinatore del gruppo attiverà immediatamente un riequilibrio, riducendo il divario nell'elaborazione.

- Più avanti in discuteremo le opzioni di configurazione che controllano la frequenza del «ping» e i timeout della sessione e come impostarli in base alle vostre esigenze.

Modifiche al comportamento del «ping» nelle recenti versioni di Kafka

- Nella versione 0.10.1, la community di Kafka ha introdotto un thread separato per il «ping» come modalità di sonda per il server.

- Ciò consente di separare la frequenza del «ping» (e quindi quanto tempo impiega il gruppo di consumer a rilevare che un consumer si è bloccato e non sta più inviando «ping») dalla frequenza del polling (che è determinata dal tempo impiegato per elaborare i dati tornato dai broker).

- Con le versioni più recenti di Kafka, è possibile configurare quanto tempo può durare l'applicazione senza polling prima che lasci il gruppo e attivi un riequilibrio

Modifiche al comportamento del «ping» nelle recenti versioni di Kafka

- Questa configurazione viene utilizzata per impedire un «livelock», in cui l'applicazione non si è arrestata in modo anomalo ma per qualche motivo non riesce a fare progressi.

- Questa configurazione è separata da session.time out.ms, che controlla il tempo necessario per rilevare un arresto anomalo del consumer e interrompere l'invio di «ping».

- Se si utilizza una nuova versione e bisogna gestire i record che richiedono più tempo per l'elaborazione, basta semplicemente ottimizzare max.poll.interval.ms in modo che gestisca ritardi più lunghi tra il polling per i nuovi record.

Come funziona il processo di assegnazione delle partizioni ai broker?

- Quando un consumer desidera unirsi a un gruppo, invia una richiesta JoinGroup al coordinatore del gruppo.

- Il primo consumer a unirsi al gruppo diventa il leader del gruppo.

- Il coordinatore riceve un elenco di tutti i consumer del gruppo dal coordinatore del gruppo (questo includerà tutti i consumer che hanno inviato un «ping» di recente e che sono quindi considerati vivi) ed è responsabile dell'assegnazione di un sottoinsieme di partizioni a ciascun consumer.

- Utilizza un'implementazione di PartitionAssignor per decidere quali partizioni devono essere gestite da un consumer.

Come funziona il processo di assegnazione delle partizioni ai broker?

- Kafka ha due criteri di assegnazione delle partizioni.
- Dopo aver deciso l'assegnazione della partizione, il leader del consumer invia l'elenco delle assegnazioni al GroupCoordinator, che invia queste informazioni a tutti i consumer.
- Ogni consumer vede solo il proprio incarico: il leader è l'unico processo client che ha l'elenco completo dei consumer nel gruppo e i loro incarichi.
- Questo processo si ripete ogni volta che si verifica un riequilibrio.

UNIT

Creare Kafka Consumers

Creare Kafka Consumers

- Il primo passo per iniziare a consumare i record è creare un'istanza di KafkaConsumer.

- La creazione di un KafkaConsumer è molto simile alla creazione di un KafkaProducer in cui si crea un'istanza di Proprietà Java con le proprietà che si desidera passare al consumer.

- Per iniziare, dobbiamo solo usare le tre proprietà obbligatorie:
  - bootstrap.servers
  - key.deserializer
  - value.deserializer

- La prima proprietà, bootstrap.servers, è la stringa di connessione a un cluster Kafka.

- È usato esattamente come in KafkaProducer.

- Le altre due proprietà, key.deserializer e value.deserializer, sono simili ai serializzatori definiti per il producer, ma anziché specificare le classi che trasformano gli oggetti Java in array di byte, è necessario specificare le classi che possono prendere un array di byte e deserializzarlo in un oggetto Java.

- C'è una quarta proprietà, che non è strettamente obbligatoria, ma che per ora faremo finta che lo sia.

- La proprietà è group.id e specifica il gruppo di consumer a cui appartiene l'istanza di KafkaConsumer.

- Sebbene sia possibile creare consumer che non appartengono a nessun gruppo di consumer, ciò non è comune, quindi per la maggior parte del prosieguo delle slide supponiamo che il consumer faccia parte di un gruppo.

- Il frammento di codice seguente mostra come creare un 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);
```

Creare Kafka Consumers

- Il frammento di codice seguente mostra come creare un 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);
```

- Partiamo dal presupposto che i record che consumiamo avranno oggetti String sia per la chiave sia per il valore del record.

- L'unica nuova proprietà è group.id corrispondente al nome del gruppo di consumer a cui appartiene questo consumer.

UNIT

Iscrizione agli argomenti

Iscrizione agli argomenti

- Una volta creato un consumer, il passo successivo è sottoscrivere uno o più argomenti.

- Il metodo subcribe () prende un elenco di argomenti come parametro, quindi è abbastanza semplice da usare:

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

- Qui creiamo semplicemente un elenco con un singolo elemento: il nome dell'argomento customerCountries.

Iscrizione agli argomenti

- È anche possibile iscriversi con un'espressione regolare.

- L'espressione può corrispondere a più nomi di argomenti e se qualcuno crea un nuovo argomento con un nome corrispondente, un riequilibrio avverrà quasi immediatamente e i consumer inizieranno a utilizzare il nuovo argomento.

- Ciò è utile per le applicazioni che devono consumare da più argomenti e in grado di gestire i diversi tipi di dati che gli argomenti conterranno.

- La sottoscrizione a più argomenti tramite un'espressione regolare viene utilizzata più comunemente nelle applicazioni che replicano i dati tra Kafka e un altro sistema.

- Per iscriversi a tutti gli argomenti di prova, possiamo chiamare:

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

UNIT

Il «Poll Loop»

Il «Poll Loop»

- Al centro dell'API consumer c'è un semplice ciclo per eseguire il polling del server per ulteriori dati.

- Una volta che il consumer si sottoscrive agli argomenti il «poll Loop» gestisce tutti i dettagli di coordinamento

  - ribilanciamento delle partizioni,
  - «ping»
  - recupero dei dati

- lasciando allo sviluppatore un'API pulita che restituisce semplicemente i dati disponibili dalle partizioni assegnate.

- Il corpo principale di un consumer sarà il seguente:

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord <String,String> record : records) {
            log.debug("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());
            int updatedCount = 1;
            if (custCountryMap.countainsValue(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
            custCountryMap.put(record.value(), updatedCount)
            JSONObject json = new JSONObject(custCountryMap);
            System.out.println(json.toString(4))
        }
    }
}
finally {
    consumer.close();
}
```

- Questo è un ciclo infinito

Il «Poll Loop»

- I consumer sono in genere applicazioni di lunga durata che eseguono continuamente il polling di Kafka per ulteriori dati.

- Mostreremo più avanti come uscire in modo pulito dal loop e chiudere il consumer.

- I consumer devono continuare a sondare Kafka o saranno considerati inattivi e le partizioni che stanno consumando saranno consegnate a un altro consumer del gruppo per continuare a consumare.

- Il parametro che passiamo a poll(), è un intervallo di timeout e controlla per quanto tempo poll() bloccherà la lettura se i dati non sono disponibili nel buffer del consumer.

- Se impostato su 0, poll() tornerà immediatamente; in caso contrario, attenderà il numero specificato di millisecondi affinché i dati arrivino dal broker.

Il «Poll Loop»

- poll () restituisce un elenco di record.
- Ogni record contiene l'argomento e la partizione da cui proviene il record, l'offset del record all'interno della partizione e, naturalmente, la chiave e il valore del record.
- In genere vogliamo scorrere l'elenco e elaborare i record singolarmente.
- Il metodo poll() accetta un parametro di timeout.
- Questo specifica quanto tempo ci vorrà per tornare al poll, con o senza dati.
- Il valore è in genere determinato dalle esigenze dell'applicazione per risposte rapide (quanto velocemente si desidera restituire il controllo al thread che esegue il polling?)

Il «Poll Loop»

- L'elaborazione di solito termina con la scrittura di un risultato in un archivio dati o l'aggiornamento di un record archiviato.

- Nell’esempio l'obiettivo è di mantenere un conteggio corrente dei clienti di ogni «nazione», quindi aggiorniamo una tabella hash e stampiamo il risultato come JSON. Un esempio più realistico memorizza il risultato degli aggiornamenti in un archivio dati.

- Il metodo close() deve essere sempre chiamato da parte del consumer prima di uscire. Ciò chiuderà le connessioni e i socket di rete e attiverà immediatamente un riequilibrio anziché attendere che il coordinatore del gruppo scopra che il consumer ha smesso di inviare i «ping» perché inattivo

- Non usare close() impegnerà il brocker e quindi si tradurrà in un periodo di tempo più lungo in cui i consumer non possono consumare messaggi da un sottoinsieme delle partizioni).

Il «Poll Loop»

- Il «poll Loop» fa molto di più che ottenere semplicemente dati.
- La prima volta che viene chiamato poll() con un nuovo consumer, il metodo si preoccupa di trovare un GroupCoordinator, unirsi al gruppo di consumer e ricevere un'assegnazione di partizione.
- Se viene attivato un ribilanciamento, verrà gestito anche all'interno del ciclo di polling.
- E, naturalmente, i «ping» che mantengono in vita i consumer vengono inviati all'interno del ciclo del sondaggio.
- Per questo motivo, cerchiamo di assicurarci che qualsiasi elaborazione che facciamo tra le iterazioni sia rapida ed efficiente.

- Non puoi avere più consumer che appartengono allo stesso gruppo in un thread e non puoi avere più thread in modo sicuro per utilizzare lo stesso consumer.

- Un consumer per thread è la regola.

- Per eseguire più consumer nello stesso gruppo in un'unica applicazione, dovrai eseguirne ciascuno nel proprio thread.

- È utile effettuare il «wrap» della logica del consumer nel proprio oggetto e quindi utilizzare ExecutorService di Java per avviare più thread ciascuno con il proprio consumer.

Il «Poll Loop» - Thread Safety

```
public static void main(String[] args) {
  int numConsumers = 3;
  String groupId = "consumer-tutorial-group"
  List<String> topics = Arrays.asList("consumer-tutorial");
  ExecutorService executor = Executors.newFixedThreadPool(numConsumers);

  final List<ConsumerLoop> consumers = new ArrayList<>();
  for (int i = 0; i < numConsumers; i++) {
    ConsumerLoop consumer = new ConsumerLoop(i, groupId, topics);
    consumers.add(consumer);
    executor.submit(consumer);
  }

  Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
      for (ConsumerLoop consumer : consumers) { consumer.shutdown(); }
      executor.shutdown();
      try {
        executor.awaitTermination(5000, TimeUnit.MILLISECONDS);
      } catch (InterruptedException e) {
        e.printStackTrace;
      }
    }
  });
}
```

- Questo è un ciclo infinito

UNIT

Configurazione Consumer

Configurazione Consumer

- Finora ci siamo concentrati sull'apprendimento dell'API consumer, ma abbiamo esaminato solo alcune delle proprietà di configurazione (quelle obbligatorie: solo i bootstrap.servers, group.id, key.deserializer e value.deserializer).

- La completa configurazione del consumer è descritta nella documentazione di Apache Kafka.

- La maggior parte dei parametri presenta valori predefiniti ragionevoli e non richiede modifiche, ma alcuni hanno implicazioni sulle prestazioni e sulla disponibilità dei consumer.

- Diamo un'occhiata ad alcune delle proprietà più importanti.

Configurazione Consumer: fetch.min.bytes

- Questa proprietà consente a un consumer di specificare la quantità minima di dati che desidera ricevere dal broker durante il recupero dei record.

- Se un broker riceve una richiesta di record da un consumer ma i nuovi record ammontano a meno byte di min.fetch.bytes, il broker attenderà fino a quando non saranno disponibili più messaggi prima di rispedire i record al consumer.

- Ciò riduce il carico sia per il consumer che per il broker in quanto devono gestire un minor numero di messaggi «back-and-forth» (avanti e indietro) nei casi in cui gli argomenti non hanno molte nuove attività (o per le ore di attività più basse della giornata).

- Bisognerà settare questo parametro su un valore superiore a quello predefinito se il consumer utilizza troppa CPU quando non ci sono molti dati disponibili o riduciamo il carico sui broker quando abbiamo un gran numero di consumer.

Configurazione Consumer: fetch.max.wait.ms

- Settando fetch.min.bytes, diciamo a Kafka di attendere fino a quando non ha abbastanza dati da inviare prima di rispondere al consumer. fetch.max.wait.ms consente di controllare il tempo di attesa. Per impostazione predefinita, Kafka attenderà fino a 500 ms (corrispondente alla latenza aggiuntiva nel caso in cui non vi siano dati sufficienti passati dai producer all'argomento Kafka per soddisfare la quantità minima di dati da restituire alla nostra richiesta).

- Se si desidera limitare la latenza potenziale (in genere a causa degli SLA che controllano la latenza massima dell'applicazione), è possibile impostare fetch.max.wait.ms su un valore inferiore.

- Se settiamo fetch.max.wait.ms su 100 ms e fetch.min.bytes su 1 MB, Kafka riceverà una richiesta di recupero dal consumer e risponderà con i dati quando ha 1 MB di dati da restituire o dopo 100 ms, qualunque cosa accada per prima.

Configurazione Consumer: max.partition.fetch.bytes

- Questa proprietà controlla il numero massimo di byte che il server restituirà per partizione. Il valore predefinito è 1 MB, il che significa che quando KafkaConsumer.poll() restituisce ConsumerRecords, l'oggetto record utilizzerà al massimo n byte (quelli indicati in max.partition.fetch.bytes) per partizione assegnata al consumer.

- Quindi, se un argomento ha 20 partizioni e vi sono 5 consumer, ogni consumer dovrà avere a disposizione 4 MB di memoria per Consumer Records.

- In pratica, andremo ad allocare più memoria poiché ogni consumer dovrà gestire più partizioni se altri consumer nel gruppo falliscono. max.partition.fetch.bytes deve essere maggiore del messaggio più grande che un broker accetterà (determinato dalla proprietà max.message.size nella configurazione del broker) oppure il broker potrebbe contenere messaggi che il consumer non sarà in grado di consumare, in tal caso il consumer si bloccherà nel tentativo di leggerli.

Configurazione Consumer: max.partition.fetch.bytes

- Un'altra considerazione importante quando si imposta max.partition.fetch.bytes è la quantità di tempo impiegata dal consumer per elaborare i dati.

- Come ricorderete, il consumer deve chiamare poll() abbastanza frequentemente per evitare il timeout della sessione e il successivo riequilibrio.

- Se la quantità di dati restituiti da un singolo poll() è molto elevata, l'elaborazione del consumer potrebbe richiedere più tempo, il che significa che non arriverà alla successiva iterazione di poll loop in tempo per evitare un timeout della sessione.

- In questo caso, le due opzioni consentono di ridurre max.partition.fetch.bytes o di aumentare il timeout della sessione

Configurazione Consumer: session.timeout.ms

- Il periodo di tempo in cui un consumer può rimanere in contatto con i broker mentre è ancora considerato attivo è, per default, pari a 3 secondi.

- Se più di session.timeout.ms passano senza che il consumer invii un ping al coordinatore del gruppo, viene considerato inattivo e il coordinatore del gruppo attiverà un riequilibrio del gruppo di consumer per allocare le partizioni dal consumer inattivo agli altri consumer del gruppo .

- Questa proprietà è strettamente correlata a heartbeat.interval.ms il quale controlla la frequenza con cui il metodo poll() di KafkaConsumer invierà un ping al coordinatore del gruppo, mentre session.timeout.ms controlla quanto tempo può trascorrere un consumer senza inviare un ping.

Configurazione Consumer: session.timeout.ms

- Pertanto, queste due proprietà vengono in genere modificate insieme: heatbeat.interval.ms deve essere inferiore a session.timeout.ms e di solito è impostato su un terzo del valore di timeout.

- Quindi se session.timeout.ms è di 3 secondi, heartbeat.interval.ms dovrebbe essere di 1 secondo.

- L'impostazione di session.timeout.ms su un valore inferiore a quello predefinito consentirà ai gruppi di consumer di rilevare e recuperare un guasto prima, ma può anche causare riequilibri indesiderati a causa del tempo impiegato dai consumer per completare il ciclo di polling o la garbage collection.

- Un settings alto di session.timeout.ms ridurrà le possibilità di ribilanciamento accidentale, ma significa anche che occorrerà più tempo per rilevare un errore reale.

Configurazione Consumer: auto.offset.reset

- Questa proprietà controlla il comportamento del consumer quando inizia a leggere una partizione per la quale non ha un offset impegnato o se lo offset inviato non è valido (di solito perché il consumer è stato giù per così tanto tempo che il record con tale offset è scaduto per il broker).

- Il valore predefinito è «latest»; il che significa che in mancanza di un offset valido, il consumer inizierà a leggere dai record più recenti (record che sono stati scritti dopo che il consumer è stato avviato).

- L'alternativa è «earliest» che significa che senza un offset valido, il consumer leggerà tutti i dati nella partizione, a partire dall'inizio.

Configurazione Consumer: enable.auto.commit

- Questo parametro controlla se il consumer committerà automaticamente gli offset; il valore predefinito è true.

- Possiamo settarlo su false se si preferisce controllare quando vengono committati offset, il che è necessario per ridurre al minimo i duplicati ed evitare la perdita di dati.

- Se si imposta enable.auto.commit su true, è possibile che si desideri controllare anche la frequenza con cui verranno committati gli offset utilizzando auto.commit.interval.ms.

Configurazione Consumer: partition.assignment.strategy

- Abbiamo appreso che le partizioni sono assegnate ai consumer in un gruppo di consumer.

- Un PartitionAssignor è una classe che, dati i consumer e gli argomenti a cui si sono sottoscritti, decide quali partizioni verranno assegnate a quale consumer.

- Per impostazione predefinita, Kafka ha due strategie di assegnazione:

  - Range
  - RoundRobin

Configurazione Consumer: partition.assignment.strategy → Range

- Assegna a ciascun consumer un sottoinsieme consecutivo di partizioni da ciascun argomento a cui si sottoscrive.

- Quindi se i consumer C1 e C2 sono iscritti a due argomenti, T1 e T2, e ciascuno degli argomenti ha tre partizioni,

  - a C1 verranno assegnate le partizioni 0 e 1 dagli argomenti T1 e T2
  - a C2 verrà assegnata la partizione 2.

- Poiché ogni argomento ha un numero irregolare di partizioni e l'assegnazione viene eseguita per ogni argomento in modo indipendente, il primo consumer finisce con più partizioni rispetto al secondo.

- Ciò accade ogni volta che viene utilizzata l'assegnazione Range; il numero di consumer non divide in modo ordinato il numero di partizioni in ciascun argomento.

Configurazione Consumer: partition.assignment.strategy → RoundRobin

- Prende tutte le partizioni da tutti gli argomenti sottoscritti e le assegna ai consumer in sequenza, una per una.

- Rifacendosi all'esempio precedente portato per il caso «range» se C1 e C2 sono in RoundRobin,

  - C1 avrebbe partizioni 0 e 2 dall'argomento T1 e partizione 1 dall'argomento T2.
  - C2 avrebbe partizione 1 dall'argomento T1 e partizioni 0 e 2 dall'argomento T2.

- In generale, se tutti i consumer sono iscritti agli stessi argomenti (uno scenario molto comune), l'assegnazione RoundRobin fara sì che tutti i consumer abbiano lo stesso numero di partizioni (o al massimo 1 differenza di partizione – ovviamente per gestire il caso di nr partizioni totali dispare ☺).

Configurazione Consumer: partition.assignment.strategy

- partition.assignment.strategy consente di scegliere una strategia di assegnazione delle partizioni.
- L'impostazione predefinita è org.apache.kafka.clients.consumer.RangeAssignor, che implementa la strategia di intervallo sopra descritta.
- Si può sostituire con org.apache.kafka.clients.consumer.RoundRobinAssignor.
- Un'opzione più avanzata è quella di implementare la propria strategia di assegnazione, nel qual caso partition.assignment.strategy dovrebbe puntare al nome della tua classe.

Configurazione Consumer: client.id

- Può trattarsi di qualsiasi stringa e verrà utilizzata dai broker per identificare i messaggi inviati dal client.

- Viene utilizzato per:
  - I log
  - Per le metriche
  - Per le quote.

Configurazione Consumer: max.poll.records

- Controlla il numero massimo di record restituiti da una singola chiamata al poll ().
- Ciò è utile per controllare la quantità di dati che l'applicazione dovrà elaborare nel ciclo di polling.

Configurazione Consumer: receive.buffer.bytes e send.buffer.bytes

- Con questi parametri si indicano le dimensioni dei buffer di invio e ricezione TCP utilizzati dai socket durante la scrittura e la lettura dei dati.

- Se sono settati su -1, verranno utilizzati i valori predefiniti del sistema operativo.

- Può essere una buona idea aumentarli quando i producer o i consumer comunicano con i broker in un centro dati diverso, poiché tali collegamenti di rete hanno generalmente una latenza più elevata e una larghezza di banda inferiore.

UNIT

Commits e Offsets

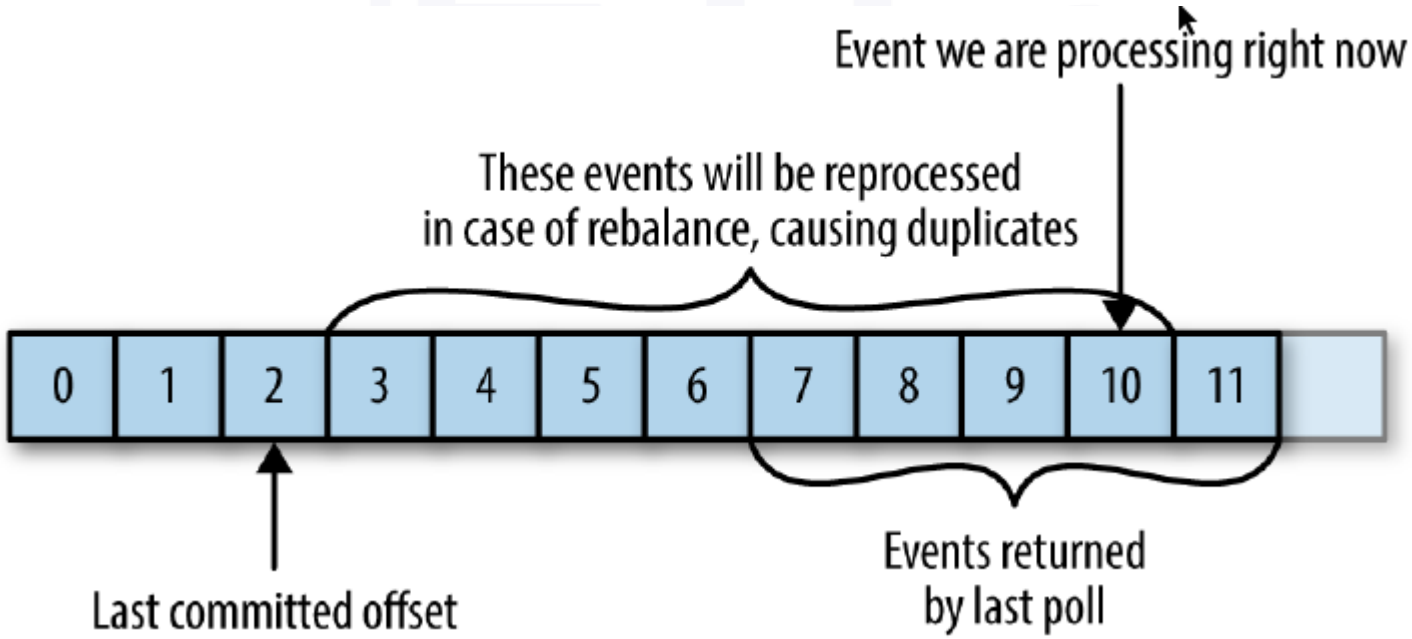
Commits e Offsets

- Ogni volta che chiamiamo poll() questi restituisce i record scritti su Kafka che i consumer del nostro gruppo non hanno ancora letto.
- Ciò significa che abbiamo un modo per rintracciare quali record sono stati letti da un consumer del gruppo.
- Come discusso in precedenza, una delle caratteristiche uniche di Kafka è che non tiene traccia dei riconoscimenti dei consumer come fanno molte code JMS.
- Consente però ai consumer di utilizzare Kafka per tenere traccia della loro posizione (offset) in ciascuna partizione.
- Chiamiamo l'azione di aggiornamento della posizione corrente nella partizione «commit».

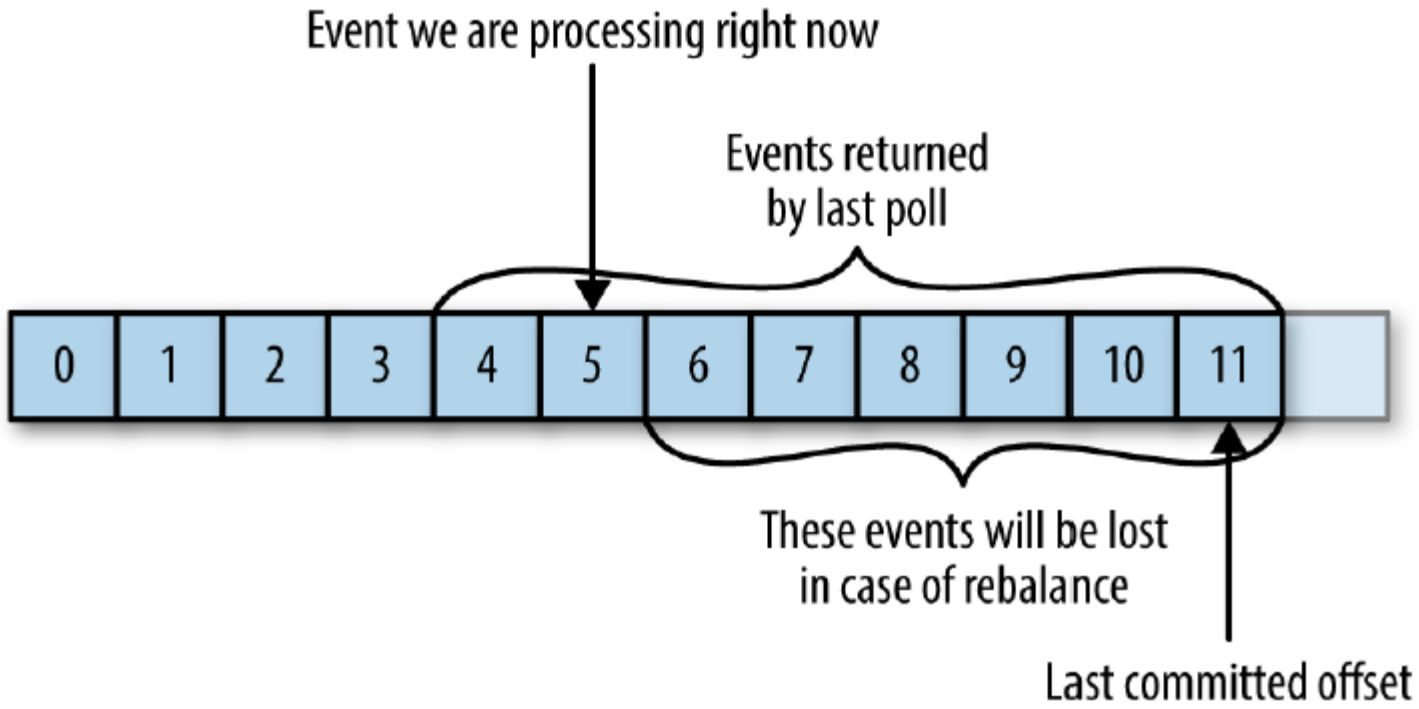
Commits e Offsets

- In che modo un consumer esegue il commit di un offset?
- Il cosumer produce un messaggio su Kafka passando un argomento speciale «__consumer_offsets», con l'offset committato per ogni partizione.
- Finché tutti i consumer sono attivi, girano e «fanno cose», ciò non avrà alcun impatto.
- Tuttavia, se un consumer si arresta in modo anomalo o un nuovo consumer si unisce al gruppo di consumer, ciò determinerà un riequilibrio.
- Dopo un ribilanciamento, a ciascun consumer può essere assegnato un nuovo set di partizioni rispetto a quello elaborato in precedenza.

- Al fine di sapere dove riprendere il lavoro, il consumer leggerà l'ultimo offset committato di ogni partizione per ripartire da lì.



- Se l’offset committato è maggiore dell’offset dell’ultimo messaggio processato dal client, tutti i messaggi tra l'ultimo offset elaborato e l'offset del commit verranno persi dal gruppo di consumer.



Commits e Offsets

- Chiaramente, la gestione degli offset ha un grande impatto sull'applicazione client.

- L'API di KafkaConsumer offre diversi modi per eseguire gli offset:

  - Automatic Commit
  - Commit Current Offset
  - Asynchronous Commit
  - Combinazione di Synchronous e Asynchronous Commits
  - Commit Specified Offset

Commits e Offsets - Automatic Commit

- Il modo più semplice per eseguire gli offset è consentire al consumer di farlo per noi.
- Se si configura enable.auto.commit = true, ogni cinque secondi il consumer committerà l'offset maggiore ricevuto dal client dal pool ().
- L'intervallo di cinque secondi è l'impostazione predefinita ed è controllato impostando auto.commit.interval.ms.
- Proprio come tutto il resto nel consumer, i commit automatici sono guidati pool loop.
- Ogni volta che esegui il polling, il consumer verifica se è il momento di eseguire il commit e, in caso affermativo, committerà gli offset restituiti nell'ultimo polling.
- Prima di utilizzare questa utile opzione, tuttavia, è importante comprendere le conseguenze.

Commits e Offsets - Automatic Commit

- Consideriamo che, per impostazione predefinita, i commit automatici avvengono ogni 5 secondi.
- Supponiamo che siamo 3 secondi dopo l'attivazione più recente e che venga attivato un riequilibrio.
- Dopo il riequilibrio, tutti i consumer inizieranno a consumare dall'ultimo offset impegnato.
- In questo caso, per l'offset sono già trascorsi 3 secondi, quindi tutti gli eventi arrivati in quei tre secondi verranno elaborati due volte.
- È possibile configurare l'intervallo di commit per eseguire il commit più frequentemente e ridurre la finestra in cui verranno duplicati i record, ma è impossibile eliminarli completamente.

Commits e Offsets - Automatic Commit

- Con autocommit abilitato, una chiamata al pool impegnerà sempre l'ultimo offset restituito dal pool precedente.
- Non sa quali eventi sono stati effettivamente elaborati, quindi è fondamentale elaborare sempre tutti gli eventi restituiti da poll () prima di chiamare nuovamente poll ().
- Proprio come poll (), close () committa automaticamente gli offset.
- Di solito ciò non è un problema, ma bisogna fare attenzione quando gestiamo le eccezioni o chiudiamo prematuramente il ciclo di polling.
- I commit automatici sono convenienti, ma non offrono agli sviluppatori un controllo sufficiente per evitare messaggi duplicati.

- Molti sviluppatori esercitano un maggiore controllo sul momento in cui vengono committati gli offset (per eliminare la possibilità di messaggi mancanti e ridurre il numero di messaggi duplicati durante il riequilibrio).

- L'API consumer ha la possibilità di eseguire il commit dell'offset corrente in un punto sensato per lo sviluppatore piuttosto che basato su un timer.

- Impostando auto.commit.offset = false, gli offset verranno impegnati solo quando l'applicazione decide esplicitamente di farlo.

- La più semplice e affidabile delle API di commit è commitSync ().

- Questa API eseguirà il commit dell'ultimo offset restituito da poll () e effettuerà il «return» solo dopo aver eseguito il commit dell'offset (generando un'eccezione se il commit fallisce per qualche motivo).

- È importante ricordare che commitSync () eseguirà il commit dell'ultimo offset restituito da poll (), quindi è necessario assicurarsi di chiamare commitSync () dopo aver terminato l'elaborazione di tutti i record della collection restituita o si rischia di perdere messaggi così come descritto in precedenza.

- Quando viene attivato il ribilanciamento, tutti i messaggi dall'inizio del batch più recente fino al momento del riequilibrio verranno elaborati due volte.

- Ecco come utilizzare commitSync per eseguire il commit degli offset dopo aver terminato l'elaborazione dell'ultimo batch di messaggi:

```
While (true) {
    ConsumerRecords<String,String> records = consumer.poll(100);
    for (ConsumerRecord<String,String > record: records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(), record.key(), record.value());
    }
    try {
            consumer.commitSync();
    } catch (CommitFailedException e) {
        log.error("commit failed", e)
    }
}
```

- Supponiamo che stampando il contenuto di un record, abbiamo finito di elaborarlo.

- Probabilmente l'applicazione farà molto di più con i record:
  - Modificarli
  - Arricchirli
  - Aggregarli
  - Visualizzarli su una dashboard
  - Notificare agli utenti eventi importanti
  - Etc. Etc.

- Bisogna determinare, in funzione del caso d'uso quando il processo di un messaggio può essere considerao «Done».

Commits e Offsets - Commit Current Offset

- Una volta terminata l'elaborazione di tutti i record nel batch corrente, chiamiamo commitSync per eseguire il commit dell'ultimo offset nel batch, prima di eseguire il polling per ulteriori messaggi.

- commitSync riprova a eseguire il commit a condizione che non vi siano errori che non possono essere recuperati.

- Se ciò accade ovvero viene riscontrato un errore, non c'è molto che possiamo fare tranne che terne conto dell'errore stesso tracciandolo da qualche parte.

Commits e Offsets - Asynchronous Commit

- Uno svantaggio del commit manuale è che l'applicazione viene bloccata fino a quando il broker non risponde alla richiesta di commit.
- Ciò limiterà il throughput dell'applicazione.
- Il rendimento può essere migliorato impegnandosi meno frequentemente, ma poi stiamo aumentando il numero di potenziali duplicati che creerà un eventuale riequilibrio.
- Un'altra opzione è l'API di commit asincrono.

- Invece di attendere che il broker risponda a un commit, inviamo semplicemente la richiesta:

```
while (true) {
    ConsumerRecords < String,
    String > records = consumer.poll(100);
    for (ConsumerRecord < String, String > record: records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(), record.key(), record.value());
    }
    consumer.commitAsync();
}
```

- In pratica committiamo l'ultimo offset e poi proseguiamo con le restanti attività.

- Lo svantaggio è che mentre commitSync () riproverà il commit fino a quando non avrà esito positivo o non incontrerà un errore irreversibile, commitAsync () non effettuerà ulteriori.

- La ragione per cui non riprova è che al momento in cui commitAsync () riceve una risposta dal server, potrebbe esservi stato un commit successivo che aveva già avuto esito positivo.

- Immaginiamo di aver inviato una richiesta di commit offset 2000. C'è un problema di comunicazione temporanea, quindi il broker non riceve mai la richiesta e quindi non risponde mai.

- Nel frattempo, abbiamo elaborato un altro batch e abbiamo committato correttamente l'offset 3000.

- Se commitAsync () ora riprova il commit precedentemente fallito, potrebbe riuscire a eseguire il commit dell'offset 2000 dopo che l'offset 3000 è già stato elaborato e sottoposto a commit.

- Nel caso di un riequilibrio, ciò causerà più duplicati.

- Evidenziamo questa complicazione e l'importanza del corretto ordine dei commit, perché commitAsync () dà anche un'opzione per passare un callback che verrà attivato quando il broker risponde. È comune utilizzare il callback per registrare gli errori di commit o contarli in una metrica, ma se si desidera utilizzare il callback per i tentativi, è necessario essere consapevoli del problema con l'ordine di commit:

```
while (true) {
    ConsumerRecords<String,String> records = consumer.poll(100);
    for (ConsumerRecord<String,String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(), record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition, OffsetAndMetadata > offsets, Exception exception) {
            if (e != null) log.error("Commit failed for offsets {}", offsets, e);
        }
    });
}
```

- Inviamo il commit e proseguiamo, ma se il commit fallisce, l'errore e gli offset verranno registrati.

Commits e Offsets - Asynchronous Commit

- Un modello semplice per ottenere l'ordine di commit corretto per i tentativi asincroni consiste nell'utilizzare un numero progressivo monotonicamente crescente.

- Aumenta il numero di sequenza ogni volta che esegui il commit e aggiungi il numero di sequenza al momento del commit nel callback commitAsync.

- Quando ti stai preparando a inviare un nuovo tentativo, controlla se il numero di sequenza di commit ottenuto dal callback è uguale alla variabile di istanza;

- se lo è, non è stato eseguito un commit più recente ed è possibile riprovare in modo sicuro.

- Se il numero di sequenza dell'istanza è superiore, non riprovare perché è già stato inviato un nuovo commit.

Commits e Offsets – Combinazione Synchronous e Asynchronous Commits

- Normalmente, gli errori occasionali di commit senza ulteriori tentativi non sono un grosso problema perché se il problema è temporaneo, il seguente commit avrà esito positivo.

- Ma se sappiamo che questo è l'ultimo commit prima di chiudere il consumer, o prima di un riequilibrio, vogliamo assicurarci che il commit abbia successo.

- Pertanto, un modello comune è quello di combinare commitAsync () con commitSync () appena prima dell'arresto.

Commits e Offsets – Combinazione Synchronous e Asynchronous Commits

```
try {
    while (true) {
        ConsumerRecords<String,String> records = consumer.poll(100);
        for (ConsumerRecord < String, String > record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());
        }
        consumer.commitAsync();
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
}
finally {
    try { consumer.commitSync(); }
    finally { consumer.close(); }
}
```

- Eseguiamo commitAsync perché è più veloce e se un commit fallisce, il commit successivo fungerà da nuovo tentativo. Ma se stiamo chiudendo la connessione, non vi sarà il «prossimo commit». A questo punto chiamiamo commitSync () perché riproverà fino a quando avrà successo o subirà un errore irreversibile assicurandoci il commit di tutto ciò che è in «pending».

- Il commit dell'offset più recente consente di eseguire il commit solo ogni volta che si completa l'elaborazione dei batch.

- E se volessimo effettuare commit più frequenti di così?

- Che cosa succede se poll () restituisce un batch enorme e si desidera eseguire degli offset nel mezzo del batch per evitare di dover elaborare nuovamente tutte quelle righe mentre si verifica un ribilanciamento?

- Non possiamo semplicemente chiamare commitSync () o commitAsync () poiché impegnerà l'ultimo offset restituito che non abbiamo ancora elaborato.

- Fortunatamente, l'API consumer consente di chiamare commitSync () e commitAsync () e passare una mappa delle partizioni e degli offset di cui si desidera eseguire il commit.

- Se si sta elaborando un batch di record e l'ultimo messaggio ricevuto dalla partizione 3 nell'argomento «clienti» che ha offset 5000, è possibile chiamare commitSync () per eseguire il commit dell'offset 5000 per la partizione 3 nell'argomento "clienti".

- Poiché il consumer potrebbe consumare più di una singola partizione, sarà necessario tenere traccia degli offset su tutti, il che aumenta la complessità del codice.

Commits e Offsets – Commit Specified Offset

```
private Map<TopicPartition,OffsetAndMetadata> currentOffsets = new HashMap <  > ();
int count = 0;
....
while(true) {
    ConsumerRecords<String,String > records = consumer.poll(100);
    for (ConsumerRecord<String,String > record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %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, "no metadata"));
        if (count % 1000 == 0)
           consumer.commitAsync(currentOffsets, null);
        count++;
    }
}
```

Commits e Offsets – Commit Specified Offset

- Dopo aver letto ogni record, aggiorniamo la mappa degli offset con l'offset del messaggio successivo che prevediamo di elaborare. È da qui che inizieremo a leggere al prossimo avvio.
- Qui, decidiamo di impegnare gli offset correnti ogni 1.000 record.
- Nella tua applicazione, puoi eseguire il commit in base al tempo o forse al contenuto dei record.
- Abbiamo chiamato commitAsync (), ma può anche essere usato commitSync().
- Naturalmente, quando si committano offset specifici, è comunque necessario eseguire tutta la gestione degli errori che abbiamo visto nelle slide precedenti.

- Un consumer vorrà fare un lavoro di «Cleanup» prima di uscire e anche prima del riequilibrio delle partizioni.

- Se stiamo chiudendo la connessione (e quindi il consumer sta per perdere la proprietà di una partizione) vorremmo committare offset dell'ultimo evento che abbiamo elaborato.

- Se il consumer ha gestito un buffer con eventi che elabora solo occasionalmente vorremmo elaborare gli eventi accumulati prima di perdere la proprietà della partizione.

- Se dobbiamo svolgere attività nostre come chiudere handle di file, connessioni al database e così via l'API consumer consente di eseguire il proprio codice quando le partizioni vengono aggiunte o rimosse dal consumer passando un ConsumerRebalanceListener quando chiamiamo il metodo subscribe()

- ConsumerRebalance Listener ha due metodi che possiamo implementare:

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

- Chiamato prima dell'inizio del riequilibrio e dopo che il consumer ha smesso di consumare i messaggi. È qui che vuoi eseguire l'offset, quindi chiunque ottenga questa partizione successiva saprà da dove iniziare.

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

- Chiamato dopo che le partizioni sono state riassegnate al broker, ma prima che il consumer inizi a consumare i messaggi.

- Di seguito un esempio che mostrerà come utilizzare onPartitionsRevoked () per eseguire il commit degli offset prima di perdere la proprietà di una partizione.

Commits e Offsets – Rebalance Listeners

```
private Map < TopicPartition, OffsetAndMetadata > currentOffsets = new HashMap <  > ();
private class HandleRebalance implements ConsumerRebalanceListener {
    public void onPartitionsAssigned(Collection < TopicPartition > partitions) {}
    public void onPartitionsRevoked(Collection < TopicPartition > partitions) {
        System.out.println("Lost partitions in rebalance. Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets);
    }
}
try {
    consumer.subscribe(topics, new HandleRebalance());
    while (true) {
        ConsumerRecords < String, String > records = consumer.poll(100);
        for (ConsumerRecord < String, String > record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %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, "no metadata"));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) { // ignore, we're closing
}
catch (Exception e) { log.error("Unexpected error", e); }
finally {
    try { consumer.commitSync(currentOffsets); } finally { consumer.close();
System.out.println("Closed consumer and we are done");
}}
```

- In questo esempio non è necessario fare nulla quando si ottiene una nuova partizione; inizieremo a consumare messaggi. Tuttavia, quando stiamo per perdere una partizione a causa del ribilanciamento, è necessario commettere offset.

- Teniamo presente che stiamo eseguendo gli ultimi offset che abbiamo elaborato, non gli ultimi offset nel batch che stiamo ancora elaborando. Questo perché una partizione potrebbe essere revocata mentre siamo ancora nel mezzo di un batch.

- Stiamo committando offset per tutte le partizioni, non solo per quelle che stiamo per perdere: poiché gli offset sono per eventi che sono già stati elaborati, non vi è alcun danno. E stiamo usando commitSync () per assicurarci che gli offset siano committati prima che il riequilibrio proceda.

- La parte più importante: passare ConsumerRebalanceListener al metodo subscribe() in modo che venga invocato dal consumer.

- Finora abbiamo visto come utilizzare poll () per iniziare a consumare i messaggi dall'ultimo offset impegnato in ciascuna partizione e procedere nell'elaborazione di tutti i messaggi in sequenza.

- Tuttavia, a volte si desidera iniziare a leggere con un offset diverso.

- Se vogliamo iniziare a leggere tutti i messaggi dall'inizio della partizione, o vogliamo saltare fino alla fine della partizione e iniziare a consumare solo nuovi messaggi, ci sono API specifiche per questo:

  - seekToBeginning (TopicPartition tp)
  - seekToEnd (TopicPartition tp).

- Tuttavia, l'API di Kafka ti consente anche di cercare un offset specifico.

- Questa abilità può essere usata in vari modi; ad esempio, per tornare indietro di alcuni messaggi o saltare alcuni messaggi (forse un'applicazione sensibile al tempo che sta andando indietro vorrà saltare su messaggi più pertinenti).

- Il caso d'uso più «stimolante» per questa capacità è quando gli offset sono memorizzati in un sistema diverso da Kafka.

- Pensiamoa a questo scenario comune: la tua applicazione sta leggendo eventi da Kafka (forse un clickstream di utenti in un sito Web), elabora i dati (forse rimuove i record che indicano i clic dai programmi automatizzati anziché dagli utenti) e quindi archivia i risultati in un database , Negozio NoSQL o Hadoop.

- Supponiamo che non vogliamo davvero perdere dati, né vogliamo archiviare gli stessi risultati nel database due volte. In questi casi, il ciclo consumer potrebbe essere così:

```
while (true) {
    ConsumerRecords < String,
    String > records = consumer.poll(100);
    for (ConsumerRecord < String, String > record: records) {
        currentOffsets.put(new TopicPartition(record.topic(),
                record.partition()), record.offset());
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

- In questo esempio, siamo molto «paranoici» per cui committiamo offset dopo l'elaborazione di ogni record.

- Tuttavia, esiste ancora la possibilità che la nostra applicazione si arresti in modo anomalo dopo che il record è stato archiviato nel database ma prima che abbiamo committato degli offset; ciò causa la rielaborazione del record che potrebbe avere come effetto la creazione di duplicati nel database (se non gestito).

- Ciò potrebbe essere evitato se esistesse un modo per memorizzare sia il record che l'offset in un'azione atomica. Sia il record che l'offset sono committati, oppure nessuno dei due è stato committato.

- Finché i record sono scritti in un database e gli offset su Kafka, questo è ovviamente impossibile perché gli ambienti sono separati.

- Ma cosa succede se abbiamo scritto sia il record che l'offset nel database, in una transazione?

- Quindi sapremo che o abbiamo finito con il record e la compensazione è impegnata oppure non lo siamo e il record verrà rielaborato.

- Ora l'unico problema è se il record è archiviato in un database e non in Kafka, come farà il nostro consumer a sapere da dove iniziare a leggere quando gli viene assegnata una partizione?

- Questo è esattamente ciò che seek () ci consente di fare. Quando il consumer inizia o quando vengono assegnate nuove partizioni, può cercare l'offset nel database e effettuare il seek() in quella posizione.

- Sulla slide successiva un esempio di utilizzo.

```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
    public void onPartitionsRevoked(Collection < TopicPartition > partitions) {
        commitDBTransaction();
    }
    public void onPartitionsAssigned(Collection < TopicPartition > partitions) {
        for (TopicPartition partition: partitions)
            consumer.seek(partition, getOffsetFromDB(partition));
    }
}
}
consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);
for (TopicPartition partition: consumer.assignment())
    consumer.seek(partition, getOffsetFromDB(partition));
while (true) {
    ConsumerRecords < String,
    String > records = consumer.poll(100);
    for (ConsumerRecord < String, String > record: records) {
        processRecord(record);
        storeRecordInDB(record);
        storeOffsetInDB(record.topic(), record.partition(), record.offset());
    }
    commitDBTransaction();
}
```

- L'idea è che i record e gli offset del database verranno inseriti nel database mentre elaboriamo i record e dobbiamo solo eseguire il commit delle transazioni quando stiamo per perdere la partizione per assicurarci che queste informazioni siano persistenti.

- Abbiamo anche un metodo immaginario per recuperare gli offset dal database e quindi tramite seek() cerchiamo tali record quando acquisiamo la proprietà di nuove partizioni.

- Quando il consumer è avviato, dopo la sottoscrizione agli argomenti, chiamiamo poll() una volta per assicurarci di unirci a un gruppo di consumer e ottenere partizioni assegnate, quindi cerchiamo immediatamente con seek() l'offset corretto nelle partizioni a cui siamo assegnati.

- seek () aggiorna solo la posizione da cui stiamo consumando, quindi il prossimo poll () prenderà i messaggi giusti. Se si è verificato un errore in seek () (ad esempio, l'offset non esiste), l'eccezione verrà generata da poll ().

- Vi sono molti modi diversi per implementare la semantica «exactly-once» memorizzando offset e dati in uno store esterno, ma tutti avranno bisogno di utilizzare il ConsumerRebalance Listener e seek() per assicurarsi che gli offset siano memorizzati in tempo e che il consumer cominci a leggere i messaggi dalla posizione corretta.

Commits e Offsets – Ma come possiamo uscire?

- Quando decididiamo di uscire dal ciclo di polling, avremo bisogno di un altro thread per chiamare con sumer.wakeup (). Se stiamo eseguendo il ciclo consumer nel thread principale, possiamo farlo da ShutdownHook.

- Si noti che consumer.wakeup () è l'unico metodo consumer che è sicuro chiamare da un thread diverso.

- La chiamata a wakeup provocherà la chiusura di poll () con WakeupException o se è stato chiamato consumer.wakeup () mentre il thread non era in attesa di polling; l'eccezione verrà generata alla successiva iterazione quando viene chiamato poll ().

- Non è necessario gestire WakeupException, ma prima di uscire dal thread è necessario chiamare consumer.close ().

Commits e Offsets – Ma come possiamo uscire?

- La chiusura del consumer committerà degli offset (se necessario), e invierà al coordinatore del gruppo un messaggio che il consumer sta lasciando il gruppo.

- Il coordinatore del consumer attiverà immediatamente il riequilibrio e non sarà necessario attendere il timeout della sessione prima che le partizioni del consumer che si sta chiudendo vengano assegnate a un altro consumer del gruppo.

- Ecco come apparirà il codice di uscita se il consumer è in esecuzione nel thread principale dell'applicazione.

Commits e Offsets – Ma come possiamo uscire?

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup();
        try { mainThread.join(); } catch (InterruptedException e) { e.printStackTrace();
    }}});
... ...
try {
    // looping until ctrl-c, the shutdown hook will cleanup on exit
    while (true) {
        ConsumerRecords < String, String > records = movingAvg.consumer.poll(1000);
        System.out.println(System.currentTimeMillis() + "-- waiting for data...");
        for (ConsumerRecord < String, String > record: records) {
            System.out.printf("offset = %d, key = %s, value = %s\n",
                        record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            System.out.println("Committing offset at position:" + consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) { // ignore for shutdown
}
finally {
    consumer.close();
    System.out.println("Closed consumer and we are done");
}}
```

Commits e Offsets – Ma come possiamo uscire?

- ShutdownHook viene eseguito in un thread separato, quindi l'unica azione sicura che possiamo intraprendere è chiamare wakeup per uscire dal ciclo di polling.

- Un altro thread che chiama wakeup provocherà un WakeupException.

- E’ consigliabile catturare l'eccezione per assicurare che la tua applicazione non venga chiusa in modo imprevisto

- Prima di uscire dal consumer, assicurati di chiuderlo in modo pulito.

UNIT

Deserializers

Deserializzazione

- Nelle slide dedicate ai producer di Kafka, abbiamo visto come serializzare tipi personalizzati e come utilizzare Avro e AvroSerializer per generare oggetti Avro dalle definizioni dello schema e quindi serializzarli quando si producono messaggi a Kafka.

- Vedremo ora come creare deserializzatori personalizzati per i tuoi oggetti e come utilizzare Avro e i suoi deserializzatori.

- E' ovvio che il serializzatore utilizzato per produrre eventi deve corrispondere al deserializzatore che verrà utilizzato quando si consumano eventi.

- Ciò significa che come sviluppatore è necessario tenere traccia di quali serializzatori sono stati utilizzati per scrivere in ciascun argomento e assicurarsi che ogni argomento contenga solo dati che i deserializzatori utilizzati possono interpretare.

Deserializzazione

- Questo è uno dei vantaggi dell'utilizzo di Avro e del repository di schemi per la serializzazione e la deserializzazione

- AvroSerializer può assicurarsi che tutti i dati scritti su un argomento specifico siano compatibili con lo schema dell'argomento, il che significa che possono essere deserializzati con la corrispondenza deserializzatore/schema.

- Eventuali errori di compatibilità, dal lato producer o lato consumer, verranno colti facilmente con un messaggio di errore appropriato, il che significa che non sarà necessario provare a eseguire il debug di array di byte per errori di serializzazione.

- Inizieremo mostrando rapidamente come scrivere un deserializzatore personalizzato, anche se questo è il metodo meno comune, e poi passeremo ad un esempio di come utilizzare Avro per deserializzare chiavi e valori dei messaggi.

Deserializzazione

```
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;
    }
}
```

Deserializzazione

```
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;
public class CustomerDeserializer implements Deserializer < Customer > {
    @ Override
    public void configure(Map configs, boolean isKey) { // nothing to configure
    }
    @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("Size of data received by IntegerDeserializer is shorter than expected");
            ByteBuffer buffer = ByteBuffer.wrap(data);
            id = buffer.getInt();
            String nameSize = buffer.getInt();
            byte[]nameBytes = new Array[Byte](nameSize);
            buffer.get(nameBytes);
            name = new String(nameBytes, 'UTF-8');
            return new Customer(id, name);
        } catch (Exception e) { throw new SerializationException("Error when serializing Customer to byte[] " + e); }
    }
    @Override
    public void close() {// nothing to close
    }
}
```

Deserializzazione

- Il consumer ha anche bisogno dell'implementazione della classe Customer e sia la classe che il serializzatore devono corrispondere alle applicazioni producer e consumer.

- In una grande organizzazione con molti consumer e producer che condividono l'accesso ai dati, questo può diventare difficile.

- Qui stiamo semplicemente invertendo la logica del serializzatore: otteniamo l'ID cliente e il nome dall'array di byte e li usiamo per costruire l'oggetto di cui abbiamo bisogno.

- Il codice di consumo che utilizza questo serializer sarà simile al seguente esempio:

Deserializzazione

```
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.CustomerDeserializer");
KafkaConsumer < String, Customer > consumer =     new KafkaConsumer <  > (props);
consumer.subscribe("customerCountries")while (true) {
    ConsumerRecords < String, Customer > records = consumer.poll(100);
    for (ConsumerRecord < String, Customer > record: records) {
        System.out.println("current customer Id: " +
                record.value().getId() + " and current customer name: " +
                record.value().getName());
    }
}
```

- Ancora una volta, è importante notare che non è consigliabile implementare un serializzatore e un deserializzatore personalizzati (forte accoppiamento tra produttori e consumatori).
- Una soluzione migliore sarebbe quella di utilizzare un formato di messaggio standard come JSON, Thrift, Protobuf o Avro. Vedremo ora come utilizzare i deserializzatori Avro con il consumatore Kafka.

UNIT

Utilizzo di un deserializzazione Avro con Kafka consumer

Deserializzazione Avro

- Supponiamo che stiamo utilizzando l'implementazione della classe Customer

- Per consumare quegli oggetti da Kafka, si desidera implementare un'applicazione di consumo simile a questa:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", schemaUrl);
String topic = "customerContacts";
KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId, url));
consumer.subscribe(Collections.singletonList(topic));
System.out.println("Reading topic:" + topic);
while (true) {
    ConsumerRecords < String,Customer > records = consumer.poll(1000);
    for (ConsumerRecord < String, Customer > record: records) {
        System.out.println("Current customer name is: " +
            record.value().getName());
    }
    consumer.commitSync();
}
```

Deserializzazione Avro

- Usiamo KafkaAvroDeserializer per deserializzare i messaggi Avro.

- schema.registry.url è un nuovo parametro. Questo indica semplicemente dove archiviamo gli schemi. In questo modo il consumatore può utilizzare lo schema registrato dal produttore per deserializzare il messaggio.

- Specifichiamo la classe generata, Customer , come tipo per il valore del record.

- record.value () è un'istanza Customer e possiamo usarla di conseguenza

UNIT

Standalone Consumer

Pperché e come utilizzare un consumer senza un gruppo

Standalone Consumer

- Finora abbiamo discusso dei gruppi di consumatori, che sono quelli in cui le partizioni vengono assegnate automaticamente ai consumatori e vengono riequilibrate automaticamente quando i consumatori vengono aggiunti o rimossi dal gruppo.

- In genere, questo comportamento è proprio quello desiderato, ma in alcuni casi si desidera qualcosa di molto più semplice.

- A volte si sa di avere un singolo consumatore che ha sempre bisogno di leggere i dati da tutte le partizioni in un argomento, o da una partizione specifica in un argomento.

- In questo caso, non c'è motivo di gruppi o riequilibri. Basta assegnare l'argomento specifico per il consumatore e/o partizioni, consumare messaggi, e commit offset a volte.

Standalone Consumer

- Quando sai esattamente quali partizioni il consumatore dovrebbe leggere, non ti iscrivi ad un argomento

- Vi assegnate alcune partizioni.

- Un consumatore può sia iscriversi ad argomenti (e far parte di un gruppo di consumatori), o assegnarsi partizioni, ma non entrambi contemporaneamente.

- Ecco un esempio di come un consumatore può assegnarsi tutte le partizioni di un argomento specifico e consumarle da esse:

Standalone Consumer

```
List < PartitionInfo > partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");
if (partitionInfos != null) {
    for (PartitionInfo partition: partitionInfos)
        partitions.add(new TopicPartition(partition.topic(), partition.partition()));
    consumer.assign(partitions);
    while (true) {
        ConsumerRecords < String,
        String > records = consumer.poll(1000);
        for (ConsumerRecord < String, String > record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(), record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

Standalone Consumer

- Iniziamo chiedendo al cluster le partizioni disponibili nell'argomento.
- Una volta che sappiamo quali partizioni vogliamo, chiamiamo assign() con la lista.
- A parte la mancanza di riequilibri e la necessità di trovare manualmente le partizioni, tutto il resto è come al solito.
- Ricordate che se qualcuno aggiunge nuove partizioni per l'argomento, il consumatore non sarà informato.
- Sarà necessario gestire il controllo di consumer.partitionsFor() periodicamente o semplicemente facendo rimbalzare l'applicazione ogni volta che le partizioni vengono aggiunte.

Standalone Consumer

- Abbiamo iniziato con una spiegazione approfondita dei gruppi di consumatori di Kafka e il modo in cui consentono a più consumatori di condividere il lavoro di lettura di eventi da argomenti.

- Abbiamo seguito la discussione teorica con un esempio pratico di un consumatore che si iscrive ad un argomento e genera continuamente gli eventi di lettura.

- Abbiamo poi esaminato i più importanti parametri di configurazione dei consumatori e come essi influenzano il comportamento dei consumatori.

- Abbiamo discusso degli offsete di come i consumatori ne tengono traccia.

- Capire come i consumatori commettono compensazioni è fondamentale quando scrivono i consumatori affidabili, così abbiamo preso il tempo per spiegare i diversi modi in cui questo può essere fatto.

- Abbiamo poi discusso di ulteriori parti delle API dei consumatori, la gestione dei riequilibri e la chiusura del consumatore.