
APACHE
kafka®

UNIT

Kafka Producers

Scrivere Messaggi Su Kafka

- Sia che Kafka venga usato come coda, bus di messaggi o piattaforma di archiviazione dei dati, alla fine tutto si riconduce a un producer che scrive i dati su Kafka, un consumer che legge i dati da Kafka o un'applicazione che ricopre entrambi i ruoli.

- Esempio:
  - in un sistema di elaborazione delle transazioni con carta di credito, ci sarà un'applicazione client, forse un negozio online, responsabile dell'invio di ogni transazione a Kafka immediatamente quando viene effettuato un pagamento.

  - Un'altra applicazione è responsabile del controllo immediato di questa transazione rispetto a un motore di regole e della determinazione dell'eventuale approvazione o rifiuto della transazione.

Scrivere Messaggi Su Kafka

- Apache Kafka viene fornito con API client integrate che gli sviluppatori possono utilizzare durante lo sviluppo di applicazioni che interagiscono con Kafka.

- Di seguito impareremo come utilizzare il producer Kafka, iniziando con una panoramica del suo design e dei suoi componenti.

- Vedremo come creare oggetti KafkaProducer e ProducerRecord, come inviare record a Kafka e come gestire gli errori che Kafka potrebbe restituire.

- Esamineremo quindi le opzioni di configurazione più importanti utilizzate per controllare il comportamento del producer.

- Concluderemo con uno sguardo più approfondito su come utilizzare diversi metodi di partizionamento e serializzatori e su come scrivere i propri serializzatori e partizionatori.

Third-Party Clients

- Oltre ai client integrati, Kafka ha un «binary wire protocol».
- Ciò significa che è possibile per le applicazioni leggere messaggi da Kafka o scrivere messaggi su Kafka semplicemente inviando le sequenze di byte corrette alla porta di rete di Kafka.
- Esistono diversi client che implementano il protocollo wire di Kafka in diversi linguaggi di programmazione, offrendo modi semplici per utilizzare Kafka non solo nelle applicazioni Java ma anche in linguaggi come C ++, Python, Go e molti altri.
- Quei client non fanno parte del progetto Apache Kafka, ma un elenco di client non Java è presente nel wiki del progetto (https://cwiki.apache.org/confluence/display/KAFKA/Clients)
- Il protocollo wire e i client esterni non rientrano nell'ambito del capitolo.

Producer Overview

- Vi sono molti motivi per cui un'applicazione potrebbe aver bisogno di scrivere messaggi su Kafka:

  - registrazione delle attività degli utenti per il controllo o l'analisi
  - registrazione delle metriche
  - memorizzazione dei messaggi di log
  - registrazione delle informazioni da dispositivi intelligenti
  - comunicazione asincrona con altre applicazioni
  - buffering delle informazioni prima di scrivere su un database
  - ….e molto altro

Producer Overview

- Questi diversi casi d'uso implicano anche requisiti diversi:
  - Ogni messaggio è critico o possiamo tollerare la perdita di messaggi?
  - Accettiamo la duplicazione accidentale di messaggi?
  - Esistono requisiti di latenza o throughput rigorosi che dobbiamo supportare?

Producer Overview

- Nell'esempio di elaborazione delle transazioni con carta di credito che abbiamo introdotto in precedenza, possiamo vedere che è fondamentale non perdere mai un singolo messaggio né duplicare alcun messaggio.

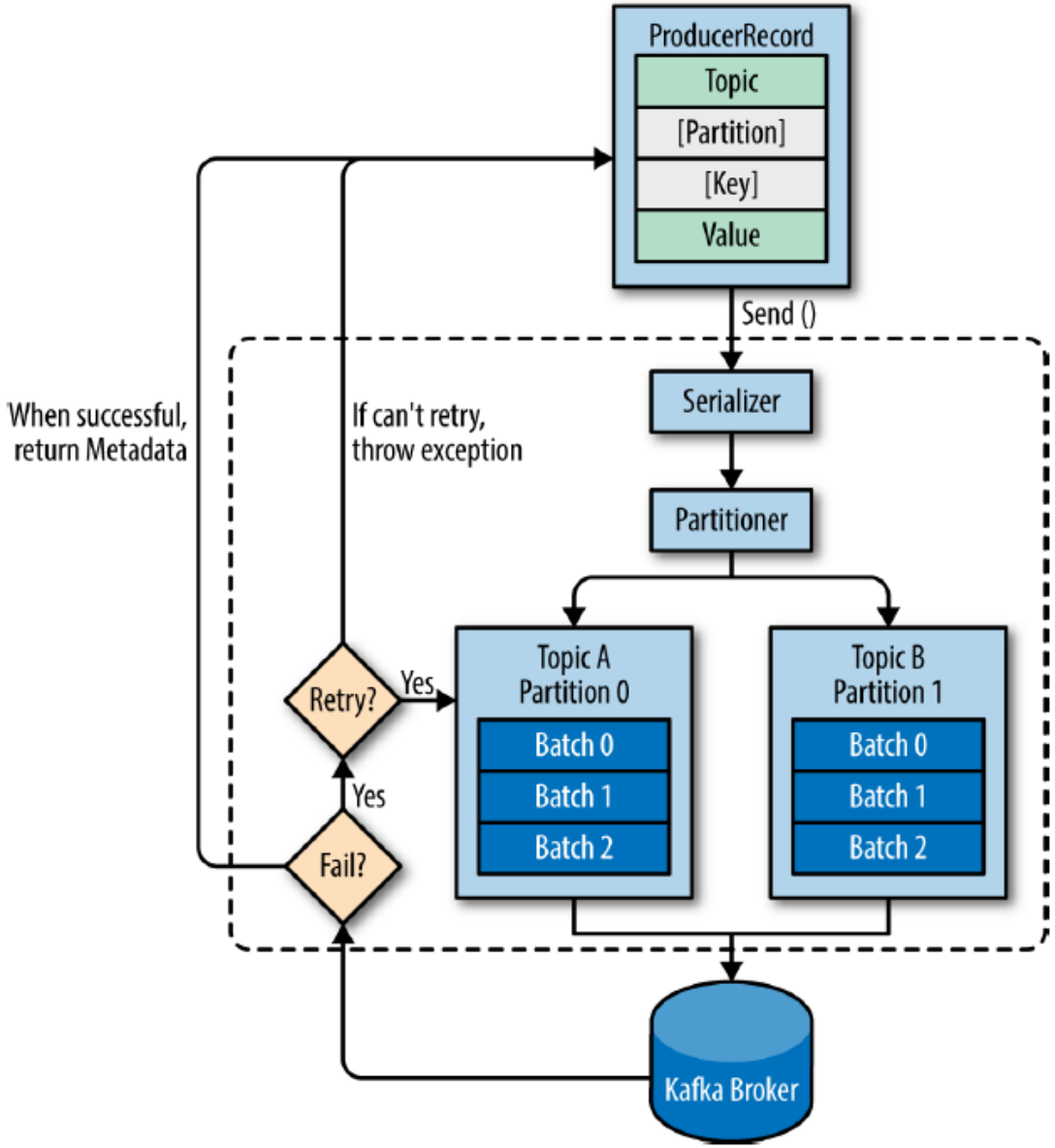
- La latenza dovrebbe essere bassa, ma è possibile tollerare latenze fino a 500 ms e la velocità effettiva dovrebbe essere molto elevata: prevediamo di elaborare fino a un milione di messaggi al secondo.

- Un caso d'uso diverso potrebbe essere quello di memorizzare le informazioni sui clic da un sito Web.
  - In tal caso, è possibile tollerare la perdita di alcuni messaggi o alcuni duplicati;
  - la latenza può essere elevata purché non vi sia alcun impatto sull'esperienza dell'utente.
  - In altre parole, non ci importa se ci vogliono alcuni secondi prima che il messaggio arrivi a Kafka, a condizione che la pagina successiva venga caricata immediatamente dopo che l'utente abbia cliccato su di un collegamento.
  - Il rendimento dipenderà dal livello di attività che prevediamo sul nostro sito Web.

Producer Overview

- I diversi requisiti influenzeranno:
  - il modo in cui vengono usale le API del producer per scrivere messaggi su Kafka
  - la configurazione da operare
- Mentre le API del producer sono molto semplici nell'utilizzo, ciò che succede sotto il «cofano» è un po' più complesso.
- Nella slide successiva vengono mostrati i passaggi principali coinvolti nell'invio di dati a Kafka.

Producer Overview



Producer Overview

- Iniziamo a produrre messaggi su Kafka creando un «ProducerRecord»; questi deve includere l'argomento a cui vogliamo inviare il record e un valore.

- Facoltativamente possiamo anche specificare una chiave e/o una partizione.

- Dopo aver inviato «ProducerRecord» la prima cosa che farà il producer è serializzare gli oggetti «chiave e valore» su ByteArrays in modo che possano essere inviati in rete

- Successivamente, i dati vengono inviati a un partizionatore.

Producer Overview

- Se abbiamo specificato una partizione in «ProducerRecord», il partizionatore non fa nulla e restituisce semplicemente la partizione specificata.
- In caso contrario, il partizionatore sceglierà una partizione per noi, in genere basata sulla chiave «ProducerRecord».
- Una volta selezionata una partizione, il produttore sa a quale argomento e partizione andrà il record.
- Quindi aggiunge il record a un batch di record che verranno inviati anche allo stesso argomento e partizione.
- Un thread separato è responsabile dell'invio di tali lotti di record ai broker Kafka appropriati.

Producer Overview

- Quando il broker riceve i messaggi, invia una risposta.
- Se i messaggi sono stati scritti correttamente su Kafka, restituirà un oggetto «RecordMetadata» con l'argomento, la partizione e l'offset del record all'interno della partizione.
- Se il broker non è riuscito a scrivere i messaggi, restituirà un errore.
- Quando il producer riceve un errore, potrebbe riprovare a inviare il messaggio alcune volte prima di rinunciare e restituire un errore.

UNIT

Costruire un produttore Kafka

Producer API

- L'API Producer consente alle applicazioni di inviare flussi di dati agli argomenti nel cluster Kafka.

- Per utilizzare il produttore, è possibile utilizzare la seguente dipendenza maven:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.3.0</version>
</dependency>
```

Costruire un produttore Kafka

- Il primo passo per scrivere messaggi su Kafka è creare un oggetto producer con le proprietà che si desidera passare al producer.

- Un producer di Kafka ha tre proprietà obbligatorie:

  - bootstrap.servers
  - key.serializer
  - value.serializer

bootstrap.servers

- Elenco «host:port» di broker che il producer utilizzerà per stabilire la connessione iniziale al cluster Kafka.

- Non è necessario che questo elenco includa tutti i broker, poiché il producer riceverà ulteriori informazioni dopo la connessione iniziale.

- Si consiglia tuttavia di includerne almeno due, quindi nel caso in cui un broker si interrompa, il produttore sarà comunque in grado di connettersi al cluster.

key.serializer

- Nome di una classe che verrà utilizzata per serializzare le chiavi dei record che produrremo in Kafka.
- I broker Kafka si aspettano che gli array di byte siano chiavi e valori dei messaggi.
- Tuttavia, l'interfaccia del producer consente, utilizzando tipi con parametri, di inviare qualsiasi oggetto Java come chiave e valore.
- Questo rende il codice molto leggibile, ma significa anche che il produttore deve sapere come convertire questi oggetti in array di byte.

key.serializer

- key.serializer deve essere impostato su un nome di una classe che implementa l'interfaccia org.apache.kafka.common.serialization.Serializer.
- Il producer utilizzerà questa classe per serializzare l'oggetto chiave in una matrice di byte.
- Il pacchetto client Kafka include ByteArraySerializer (che non fa molto), StringSerializer e IntegerSerializer, quindi se si usano tipi comuni, non è necessario implementare i propri serializzatori.
- È necessario impostare key.serializer anche se si intende inviare solo valori.

value.serializer

- Nome di una classe che verrà utilizzata per serializzare i valori dei record che produrremo su Kafka.

- Allo stesso modo in cui si imposta key.serializer su un nome di una classe che serializzerà l'oggetto chiave del messaggio su un array di byte, si imposta value.serializer su una classe che serializzerà l'oggetto valore del messaggio.

Producer

- Il frammento di codice seguente mostra come creare un nuovo produttore impostando solo i parametri obbligatori e usando i valori predefiniti per tutto il resto:

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");
KafkaProducer<String, String> producer = new KafkaProducer<String, String>(kafkaProps);
```

Producer

- Con un'interfaccia così semplice, è chiaro che la maggior parte del controllo sul comportamento del producer viene effettuata settando le proprietà di configurazione corrette.

- La documentazione di Apache Kafka copre tutte le opzioni di configurazione e analizzeremo quelle importanti più avanti.
  (http://kafka.apache.org/documentation.html#producerconfigs)

- Dopo aver creato un'istanza per un producer, è tempo di iniziare a inviare messaggi.

Producer

- Esistono tre metodi principali per l'invio di messaggi.

- Fire-and-forget:
  - Inviamo un messaggio al server e non ci interessa davvero se arrivi con successo o meno.
  - Il più delle volte arriverà con successo, poiché Kafka è altamente disponibile e il producer riproverà a inviare i messaggi automaticamente.
  - Tuttavia, utilizzando questo metodo alcuni messaggi potrebbero andare persi

- Synchronous send
  - Inviamo un messaggio, il metodo send() restituisce un oggetto Future e usiamo get() per attendere la risposta e verificare se send() ha avuto successo o meno.

Producer

- Asynchronous send
  - Chiamiamo il metodo send() con una funzione di callback, che viene attivata quando riceve una risposta dal broker Kafka.

Producer

- Negli esempi che seguiranno vedremo come inviare messaggi utilizzando questi metodi e come gestire i diversi tipi di errori che potrebbero verificarsi.

- Gli esempi (che hanno puro scopo didattico) sono a thread singolo; un oggetto producer può (e dovrebbe) essere utilizzato con tecniche multi-thread per inviare messaggi.

- Se è necessario un throughput migliore, è possibile aggiungere più thread che utilizzano lo stesso producer.

- Nella medesima applicazione è possibile utilizzare più oggetti producer per ottenere una velocità ancora maggiore.

UNIT

Invio Messaggio Fire-and-forget

Sending di un “Message” a Kafka

- Il modo più semplice per inviare un messaggio è il seguente:

```
ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", “Italia");
try {
  producer.send(record);
} catch (Exception e) {
  e.printStackTrace();
}
```

- L’oggetto record è di tipo ProducerRecord; usiamo come costruttore quello che richiede il nome dell'argomento a cui stiamo inviando i dati (è sempre una stringa), e la chiave e il valore che stiamo inviando a Kafka (in questo caso d’uso stringhe).

- I tipi di chiave e valore devono corrispondere ai nostri oggetti serializzatore e produttore.

Sending di un “Message” a Kafka

- Il metodo send() dell'oggetto producer invia l’oggetto di tipo ProducerRecord a Kafka.
- Il messaggio verrà inserito in un buffer e verrà inviato al broker in un thread separato.
- Il metodo send() restituisce un oggetto «Java Future» (riguarda la programmazione concorrete) con RecordMetadata; poiché ignoriamo un eventuale valore restituito, non abbiamo modo di sapere se il messaggio è stato inviato correttamente o meno.
- Questo metodo di invio dei messaggi può essere utilizzato quando, a livello di requisito funzionale, non ci interessa sapere che «fine abbia fatto il messaggio stesso».
- Ovviamente per i test va bene, «ma in produzione evitiamo di usare questo approccio» ☺

Sending di un “Message” a Kafka

- Mentre ignoriamo gli errori che possono verificarsi durante l'invio di messaggi ai broker Kafka o negli stessi broker, potremmo comunque ottenere un'eccezione se il producer ha riscontrato errori prima di inviare il messaggio a Kafka.

- Le eccezioni possono essere

  - SerializationException: quando non riesce a serializzare il messaggio
  - BufferExhaustedException o TimeoutException: se il buffer è pieno
  - InterruptException: se il thread di invio è stato interrotto.

UNIT

Invio Messaggio Sincrono

Sending di un “Message” sincrono a Kafka

- Il modo più semplice per inviare un messaggio in modo sincrono è il seguente:

```
ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", “Italia");
try {
  producer.send(record).get();
} catch (Exception e) {
  e.printStackTrace();
}
```

- Qui, stiamo usando Future.get() per attendere una risposta da Kafka.

- Questo metodo genererà un'eccezione se il record non viene inviato correttamente a Kafka:
  - Se non ci sono errori, otterremo un oggetto RecordMetadata che possiamo usare per recuperare l'offset in cui è stato scritto il messaggio.
  - Riscontreremo un'eccezione se si sono verificati errori prima di inviare dati a Kafka, durante l'invio, se i broker Kafka hanno restituito eccezioni non attendibili o se abbiamo esaurito i tentativi disponibili.

Sending di un “Message” sincrono a Kafka

- KafkaProducer ha due tipi di errori.

- Gli errori reversibili sono quelli che possono essere risolti inviando di nuovo il messaggio; ad esempio, un errore di connessione può essere risolto perché la connessione potrebbe essere ristabilita.

- Un errore "nessun leader" può essere risolto quando un nuovo leader viene eletto per la partizione; KafkaProducer può essere configurato per gestire automaticamente il reinvio dei messaggi; in questo scenario le eventuali eccezioni sono intercettabili solo quando il numero di tentativi è stato esaurito e l'errore non è stato risolto.

- Alcuni tipi di errori non prevedono il rinvio automatico anche in presenza di tale configurazione; questo è il caso dell’errore «dimensione del messaggio troppo grande». In questo scenario KafkaProducer non effettuerà ulteriori tentativi ma restituirà immediatamente l'eccezione.

UNIT

Invio Messaggio Asincrono

Sending di un “Message” asincrono a Kafka

- Supponiamo che il tempo di andata e ritorno (roundtrip) di rete tra la nostra applicazione e il cluster Kafka sia di 10 ms; se attendiamo una risposta dopo aver inviato ciascun messaggio, l'invio di 100 messaggi richiederà circa 1 secondo.

- D'altra parte, se inviamo tutti i nostri messaggi e non aspettiamo alcuna risposta, l'invio di 100 messaggi richiederà a malapena il tempo previsto.

- Nella maggior parte dei casi non abbiamo davvero bisogno di una risposta: Kafka restituisce l'argomento, la partizione e l'offset del record dopo che è stato scritto, che di solito non è richiesto dall'app di invio.

- D'altra parte, dobbiamo sapere quando non siamo riusciti a inviare un messaggio completamente in modo da poter generare un'eccezione, registrare un errore o tracciare l’informazione in un file «di log degli errori» per un'analisi successiva.

Sending di un “Message” asincrono a Kafka

- Per inviare messaggi in modo asincrono e gestire ancora scenari di errore, il producer supporta l'aggiunta di un callback durante l'invio di un record.

- Ecco un esempio di come utilizziamo un callback:

```
private class DemoProducerCallback implements Callback {
     @Override
     public void onCompletion(RecordMetadata recordMetadata, Exception e) {
     if (e != null) {
       e.printStackTrace();
     }
}
}
ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
producer.send(record, new DemoProducerCallback());
```

Sending di un “Message” asincrono a Kafka

- Per utilizzare i callback, è necessaria una classe che implementi l'interfaccia org.apache.kafka.clients.producer.Callback, che ha un singolo metodo virtual: onCompletion().

- Se Kafka restituisce un errore e il metodo di callbacke onCompletion () avrà un'eccezione non nulla; nel corpo del metodo «gestiremo» l’eventuale eccezione.

UNIT

Configurare il Producer

Configurazione Producer

- Finora abbiamo visto pochissimi parametri di configurazione per i producer:
  - l'URI bootstrap.servers
  - i serializzatori obbligatori.

- Il produttore ha un gran numero di parametri di configurazione; la maggior parte sono riportati nella documentazione di Apache Kafka e molti hanno impostazioni predefinite ragionevoli, quindi non c'è motivo di armeggiare con ogni singolo parametro.

- Tuttavia, alcuni dei parametri hanno un impatto significativo sull'utilizzo della memoria, sulle prestazioni e sull'affidabilità dei producer.

Configurazione Producer: acks

- Il parametro «acks» controlla quante repliche di partizioni devono ricevere il record prima che il produttore possa considerare la scrittura corretta; questa opzione ha un impatto significativo sulla probabilità di perdere i messaggi.

- Esistono tre valori consentiti per il parametro acks:

Configurazione Producer: acks = 0

- Il producers non attenderà una risposta dal broker prima di ritenere che il messaggio sia stato inviato correttamente.

- Ciò significa che se qualcosa è andato storto e il broker non ha ricevuto il messaggio, il producer non lo saprà e il messaggio andrà perso.

- Tuttavia, poiché il producer non è in attesa di alcuna risposta dal server, può inviare messaggi con la stessa velocità supportata dalla rete, quindi questa impostazione può essere utilizzata per ottenere un throughput molto elevato.

Configurazione Producer: acks = 1

- il producer riceverà una risposta positiva dal broker nel momento in cui la replica del leader ha ricevuto il messaggio.

- Se il messaggio non può essere scritto sul leader (ad es. Se il leader si è arrestato in modo anomalo e non è stato ancora eletto un nuovo leader), il producer riceverà una risposta di errore e potrà riprovare a inviare il messaggio evitando la potenziale perdita di dati.

- Il messaggio può comunque andare perso se il leader si arresta in modo anomalo e una replica senza questo messaggio viene eletta come nuovo leader (tramite elezione del leader «impuro»).

- In questo caso, la velocità effettiva dipende dal fatto che inviamo messaggi in modo sincrono o asincrono.

Configurazione Producer: acks = 1

- Se il nostro codice client attende una risposta dal server (chiamando il metodo get () dell'oggetto Future restituito quando si invia un messaggio), aumenterà ovviamente la latenza in modo significativo (mediante un roundtrip di rete).

- Se il client utilizza i callback, la latenza sarà nascosta, ma il throughput sarà limitato dal numero di messaggi in «invio» (ovvero da quanti messaggi il producer invierà prima di ricevere risposte dal server).

Configurazione Producer: acks = all

- Il producer riceverà una risposta positiva dal broker una volta che tutte le repliche sincronizzate avranno ricevuto il messaggio.

- Questa è la modalità più sicura poiché assicura che più di un broker abbia il messaggio e che il messaggio sopravviverà anche in caso di crash.

- Tuttavia, la latenza di cui abbiamo discusso nel caso acks = 1 sarà ancora più elevata, poiché aspetteremo che più di un broker riceva il messaggio.

Configurazione Producer: buffer.memory

- Il parametro setta la quantità di memoria che il producer utilizzerà per bufferizzare i messaggi in attesa di essere inviati ai broker.

- Se i messaggi vengono inviati dall'applicazione più velocemente di quanto possano essere consegnati al server, il produttore potrebbe esaurire lo spazio e ulteriori chiamate send() bloccheranno o genereranno un'eccezione, in base al parametro block.on.buffer.full (sostituito con max.block.ms nella versione 0.9.0.0, che consente il blocco per un certo tempo e quindi l'eccezione).

Configurazione Producer: compression.type

- Per impostazione predefinita, i messaggi vengono inviati non compressi. Con l'uso del parametro «compression.type» il quale può assumere i seguenti valori, snappy, gzip o lz4, verranno utilizzati gli algoritmi di compressione corrispondenti per comprimere i dati prima di inviarli ai broker.

- La «snappy compression» è stata inventata da Google per fornire rapporti di compressione decenti con sovraccarico della CPU basso e buone prestazioni; è consigliata nei casi in cui le prestazioni e la larghezza di banda rappresentino un problema.

- La compressione Gzip in genere utilizza più CPU e tempo ma comporta migliori rapporti di compressione, quindi è consigliata nei casi in cui la larghezza di banda della rete è più limitata.

- Abilitando la compressione, si riduce l'utilizzo e l'archiviazione della rete, che spesso costituisce un collo di bottiglia durante l'invio di messaggi a Kafka.

Configurazione Producer: retries

- Quando il producer riceve un messaggio di errore dal server, l'errore potrebbe essere temporaneo (ad esempio, una mancanza di leader per una partizione).

- In questo caso, il valore del parametro retries controllerà quante volte il producer riproverà a inviare il messaggio prima di rinunciare e notificare al client un problema.

- Per impostazione predefinita, il producer attenderà 100 ms tra i tentativi, ma è possibile controllarlo utilizzando il parametro retry.backoff.ms.

- Si consiglia di testare il tempo necessario per il ripristino da un broker in crash (ovvero, quanto tempo prima che tutte le partizioni ottengano nuovi leader) e impostare il numero di tentativi e il ritardo tra di essi in modo tale che il tempo totale impiegato per riprovare sia più lungo del tempo che impiega il cluster Kafka per riprendersi dallo schianto, altrimenti il producer si «arrenderà» troppo presto.

Configurazione Producer: batch.size

- Quando più record vengono inviati alla stessa partizione, il producer li raggrupperà insieme.
- Questo parametro controlla la quantità di memoria in byte («non i messaggi!») che verrà utilizzata per ciascun batch. Quando il batch è pieno, verranno inviati tutti i messaggi nel batch.
- Tuttavia, ciò non significa che il producer attenderà che il batch si riempia. Il producer invierà lotti completi a metà e persino lotti con un solo messaggio al loro interno.
- Pertanto, l'impostazione di una dimensione batch troppo grande non causerà ritardi nell'invio dei messaggi; utilizzerà solo più memoria per i batch.
- Se si imposta una dimensione del batch troppo piccola, si aggiungeranno costi generali perché il producer dovrà inviare i messaggi più frequentemente.

Configurazione Producer: linger.ms

- linger.ms controlla il tempo di attesa per ulteriori messaggi prima di inviare il batch corrente.
- KafkaProducer invia un batch di messaggi quando il batch corrente è pieno o quando viene raggiunto il limite linger.ms.
- Per impostazione predefinita, il producer invierà messaggi non appena è disponibile un thread mittente per inviarli, anche se nel batch è presente un solo messaggio.
- Impostando linger.ms su un valore superiore a 0, chiediamo al producer di attendere qualche millisecondo per aggiungere ulteriori messaggi al batch prima di inviarlo ai broker.
- Ciò aumenta la latenza ma aumenta anche la velocità effettiva (poiché inviamo più messaggi contemporaneamente vi è un minor sovraccarico per messaggio).

Configurazione Producer: client.id

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

- Viene utilizzato nella registrazione e nelle metriche e per le quote.

Configurazione Producer: max.in.flight.requests.per.connection

- Controlla quanti messaggi il producer invierà al server senza ricevere risposte.

- L'impostazione su un valore alto può aumentare l'utilizzo della memoria migliorando al contempo la velocità effettiva, ma impostandola su un valore troppo alto potrebbe ridurre la velocità in quanto il batch diventa meno efficiente.

- L'impostazione su 1 garantirà che i messaggi vengano scritti nel broker nell'ordine in cui sono stati inviati, anche quando si verificano nuovi tentativi.

Configurazione Producer: timeout.ms, request.timeout.ms, and metadata.fetch.timeout.ms

- Controllano per quanto tempo il producer attenderà una risposta dal server quando invierà i dati (request.timeout.ms) e quando richiederà metadati come i leader attuali per le partizioni su cui stiamo scrivendo (metadata.fetch.timeout.ms) .

- Se il timeout viene raggiunto senza risposta, il producer riproverà a inviare o risponderà con un errore (tramite eccezione o invio callback).

- timeout.ms controlla il tempo in cui il broker attenderà che le repliche sincronizzate riconoscano il messaggio per soddisfare la configurazione di acks: il broker restituirà un errore se il tempo trascorre senza i necessari riconoscimenti.

Configurazione Producer: max.block.ms

- Questo parametro controlla per quanto tempo il producer bloccherà quando chiama send () e quando richiede esplicitamente metadati tramite partitionsFor ().

- Tali metodi si bloccano quando il buffer di invio del producer è pieno o quando i metadati non sono disponibili.

- Quando viene raggiunto max.block.ms, viene generata un'eccezione di timeout.

Configurazione Producer: max.request.size

- Controlla le dimensioni di una richiesta di messaggio inviato dal producer.
- Limita sia la dimensione del messaggio più grande che può essere inviato sia il numero di messaggi che il producer può inviare in una richiesta.
- Ad esempio, con una dimensione di richiesta massima predefinita di 1 MB, il messaggio più grande che è possibile inviare è 1 MB o il produttore può raggruppare 1.000 messaggi di dimensione 1 K ciascuno in una richiesta.
- Inoltre, il broker ha il proprio limite sulla dimensione del messaggio più grande che accetterà (message.max.bytes).
- Di solito è una buona idea far corrispondere queste configurazioni in modo tale che il producer non tenti di inviare messaggi di dimensioni che verranno rifiutate dal broker.

Configurazione Producer: receive.buffer.bytes and send.buffer.bytes

- Gestiscono le dimensioni dei buffer di invio e ricezione TCP utilizzati dai socket durante la scrittura e la lettura dei dati.

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

- È una buona idea aumentarli quando i producer o i consumer comunicano con i broker in un centro dati diverso perché tali collegamenti di rete hanno in genere una latenza più elevata e una larghezza di banda inferiore.

UNIT

Garanzie dell’ordine dei messaggi

Garanzia dell'ordine dei messaggi

- Apache Kafka conserva l'ordine dei messaggi all'interno di una partizione.
- Ciò significa che se i messaggi sono stati inviati dal producer in un ordine specifico, il broker li scriverà su una partizione in quell'ordine e tutti i consumatori li leggeranno in quell'ordine.
- Per alcuni casi d'uso, l'ordine è molto importante. C'è una grande differenza tra depositare 100€ in un conto e successivamente prelevarlo, e viceversa!
- Tuttavia, alcuni casi d'uso sono meno sensibili.
- Settando il parametro retries su diverso da zero e la sessione max.in.flights.requests.per.session > 1 significa che è possibile che il broker non riesca a scrivere il primo batch di messaggi, riesca a scrivere il secondo (che era già in «volo»), quindi riprovare il primo batch e riuscire, invertendo così l'ordine.

Garanzia dell’ordine dei messaggi

- Di solito, impostare il numero di tentativi su zero non è un'opzione in un sistema affidabile, quindi se garantire l'ordine è fondamentale, si consiglia di impostare in.flight.requests.per.session = 1 per assicurarsi che mentre un gruppo di messaggi sono in fase di rinvio (causa precedenti fallimenti), non verranno inviati messaggi aggiuntivi (poiché potrebbe potenzialmente invertire l'ordine corretto).

- Ciò limiterà notevolmente il rendimento del producer, quindi va utilizzato solo quando l'ordine è importante.

UNIT

Serializers

Serializer

- Come visto negli esempi precedenti, la configurazione del producer include serializzatori obbligatori.
- Abbiamo visto come utilizzare il serializzatore di stringhe predefinito.
- Kafka include anche serializzatori per numeri interi e ByteArrays, ma questo non copre la maggior parte dei casi d'uso.
- Alla fine però vorremmo essere in grado di serializzare record più generici.
- Inizieremo mostrando come scrivere il proprio serializzatore e quindi introdurre il serializzatore «Avro» come alternativa consigliata.

Custom Serializers

- Quando l'oggetto da inviare a Kafka non è una stringa o un numero intero semplici, si può optare per l'utilizzare di una libreria di serializzazione generica come Avro, Thrift o Protobuf per creare record o creare una serializzazione personalizzata per gli oggetti che si intende utilizzare.

- Consigliamo vivamente di utilizzare una libreria di serializzazione generica.

- Per capire come funzionano i serializzatori e perché è una buona idea utilizzare una libreria di serializzazione, vediamo cosa serve per scrivere un serializzatore personalizzato.

Custom Serializers

- Supponiamo che invece di registrare solo il nome del cliente si voglia registrare un oggetto creato a partire da una classe semplice per rappresentare i clienti:

```
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;
  }
}
```

Custom Serializers

- Supponiamo ora di voler creare un serializzatore personalizzato per questa classe:

```
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer < Customer > {
     @ Override
    public void configure(Map configs, boolean isKey) {
        // nothing to configure
    }
     @ Override
    /**
    We are serializing Customer as:
    4 byte int representing customerId
    4 byte int representing length of customerName in UTF-8 bytes (0 if name is
    Null)
    N bytes representing customerName in UTF-8
     */
```

Custom Serializers

```
public byte[]serialize(String topic, Customer data) {
        try {
            byte[]serializedName; int stringSize;
            if (data == null)  return null;
            else {
                if (data.getName() != null) {
                    serializeName = data.getName().getBytes("UTF-8");
                    stringSize = serializedName.length;
                } else {
                    serializedName = new byte[0]; stringSize = 0;
                }
            }
            ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
            buffer.putInt(data.getID());
            buffer.putInt(stringSize);
            buffer.put(serializedName);
            return buffer.array();
        } catch (Exception e) {
            throw new SerializationException("Error when serializing Customer to
                byte[] " + e);
        }
    }
     @ Override
    public void close() { // nothing to close
    }
}
```

Custom Serializers

- La configurazione di un producer con CustomerSerializer consentirà di definire ProducerRecord <String, Customer> e di inviare i dati del cliente e di passare gli oggetti del cliente direttamente al produttore.

- Questo esempio è piuttosto semplice e mostra quanto sia «debole» il codice; se, ad esempio, vi sono troppi clienti e nel mentre abbiamo cambiato l'ID cliente in Long, o se avessimo mai deciso di aggiungere un campo startDate al Cliente, avremo un serio problema nel mantenere la compatibilità tra vecchi e nuovi messaggi.

- Il debug dei problemi di compatibilità tra diverse versioni di serializzatori e deserializzatori è piuttosto impegnativo: è necessario confrontare array di byte grezzi.

Custom Serializers

- A peggiorare le cose, se più team nella stessa azienda finiscono per scrivere i dati dei clienti su Kafka, dovranno tutti utilizzare gli stessi serializzatori e modificare il codice allo stesso tempo.

- Per questi motivi, si consiglia di utilizzare serializzatori e deserializzatori esistenti come JSON, Apache Avro, Thrift o Protobuf.

- Nella slide seguenti descriveremo Apache Avro e quindi mostreremo come serializzare i record Avro e inviarli a Kafka.

UNIT

Serializing Using Apache Avro

Serializing Using Apache Avro

- Apache Avro è un formato di serializzazione dei dati «language-neutral». Il progetto è stato creato da Doug Cutting per fornire un modo per condividere file di dati con alta numerosità di utenti in accesso.

- I dati di Avro sono descritti in uno schema «language-independent”. Lo schema è generalmente descritto in JSON e la serializzazione è di solito su file binari, sebbene sia supportata anche la serializzazione su JSON.

- Avro presume che lo schema sia presente durante la lettura e la scrittura di file, generalmente incorporando lo schema nei file stessi.

- Una delle funzionalità più interessanti di Avro, e ciò che lo rende idoneo all'uso in un sistema di messaggistica come Kafka, è che quando l'applicazione che scrive messaggi passa a un nuovo schema, le applicazioni che leggono i dati possono continuare a elaborare i messaggi senza richieste di cambiamento o aggiornamento.

- Partiamo dalla seguente isola dati JSON:

```
{
    "namespace": "customerManagement.avro",
    "type": "record",
    "name": "Customer",
    "fields": [{
            "name": "id",
            "type": "int"
        }, {
            "name": "name",
            "type": "string"
        }, {
            "name"   : "faxNumber",
            "type"   : ["null", "string"],
            "default": "null"
        }
    ]
}
```

- i campi id e name sono obbligatori, mentre il numero di fax è facoltativo e il valore predefinito è null.

- Supponiamo di aver utilizzato questo schema per alcuni mesi e generato alcuni terabyte di dati in questo formato.

- Supponiamo ora che decidiamo che nella nuova versione aggiorneremo al ventunesimo secolo e non includeremo più un campo di numero di fax e utilizzeremo invece un campo di posta elettronica.

```
{
    "namespace": "customerManagement.avro",
    "type": "record",
    "name": "Customer",
    "fields": [{
            "name": "id",
            "type": "int"      }, {
            "name": "name",
            "type": "string"   }, {
            "name": "email",
            "type": ["null", "string"],
            "default": "null"
        }
    ]
}
```

- Ora, dopo l'aggiornamento alla nuova versione, i vecchi record conterranno "faxNumber" e i nuovi record conterranno "e-mail".

- In molte organizzazioni, gli aggiornamenti vengono eseguiti lentamente e per molti mesi.

- Pertanto dobbiamo considerare come le applicazioni di pre-aggiornamento che utilizzano ancora i numeri di fax e le applicazioni di post-aggiornamento che utilizzano la posta elettronica saranno in grado di gestire tutti gli eventi in Kafka.

- Name () e getId () continueranno a funzionare senza alcuna modifica, ma getFaxNumber () restituirà null poiché il messaggio non conterrà un numero di fax.

- Supponiamo ora di aggiornare la nostra applicazione di lettura e che non abbia più il metodo getFaxNumber () ma piuttosto getEmail ().

- Se incontra un messaggio scritto con il vecchio schema, getEmail () restituirà null perché i messaggi più vecchi non contengono un indirizzo e-mail.

- Questo esempio illustra i vantaggi dell'utilizzo di Avro: anche se abbiamo modificato lo schema nei messaggi senza modificare tutte le applicazioni che leggono i dati, non ci saranno eccezioni o errori di interruzione e non saranno necessari costosi aggiornamenti dei dati esistenti.

- Tuttavia, ci sono due avvertenze in questo scenario:
  - Lo schema utilizzato per la scrittura dei dati e lo schema previsto dall'applicazione di lettura devono essere compatibili. La documentazione di Avro include regole di compatibilità. (https://avro.apache.org/docs/1.7.7/spec.html#Schema+Resolution)
  - Il deserializzatore dovrà accedere allo schema utilizzato durante la scrittura dei dati, anche quando è diverso dallo schema previsto dall'applicazione che accede ai dati. Nei file Avro, lo schema di scrittura è incluso nel file stesso, ma esiste un modo migliore di gestirlo per i messaggi di Kafka.

UNIT

Uso degli Avro Records con Kafka

Uso degli Avro Records con Kafka

- A differenza dei file Avro, in cui la memorizzazione dell'intero schema nel file di dati è associata a un sovraccarico abbastanza ragionevole, la memorizzazione dell'intero schema in ciascun record di solito supera il doppio della dimensione del record.

- Tuttavia, Avro richiede ancora che l'intero schema sia presente durante la lettura del record, quindi è necessario locare lo schema altrove.

- Per raggiungere questo obiettivo, seguiamo un modello di architettura comune e utilizziamo un «Schema Registry».

- Lo «Schema Registry» non fa parte di Apache Kafka ma vi sono diverse opzioni open source tra cui scegliere ma si consiglia di utilizzar lo «Schema Registry» di Confluet

Uso degli Avro Records con Kafka

- È possibile trovare il codice su GitHub oppure installarlo come parte della piattaforma Confluent.
  (https://github.com/confluentinc/schema-registry)

- Se si decide di utilizzare uno «Schema Registry» si consiglia di controllare e studiare la relativa documentazione.

- L'idea è di memorizzare tutti gli schemi utilizzati per scrivere i dati su Kafka nel registro.

- Quindi memorizziamo semplicemente l'identificatore per lo schema nel record che produciamo su Kafka.

Uso degli Avro Records con Kafka

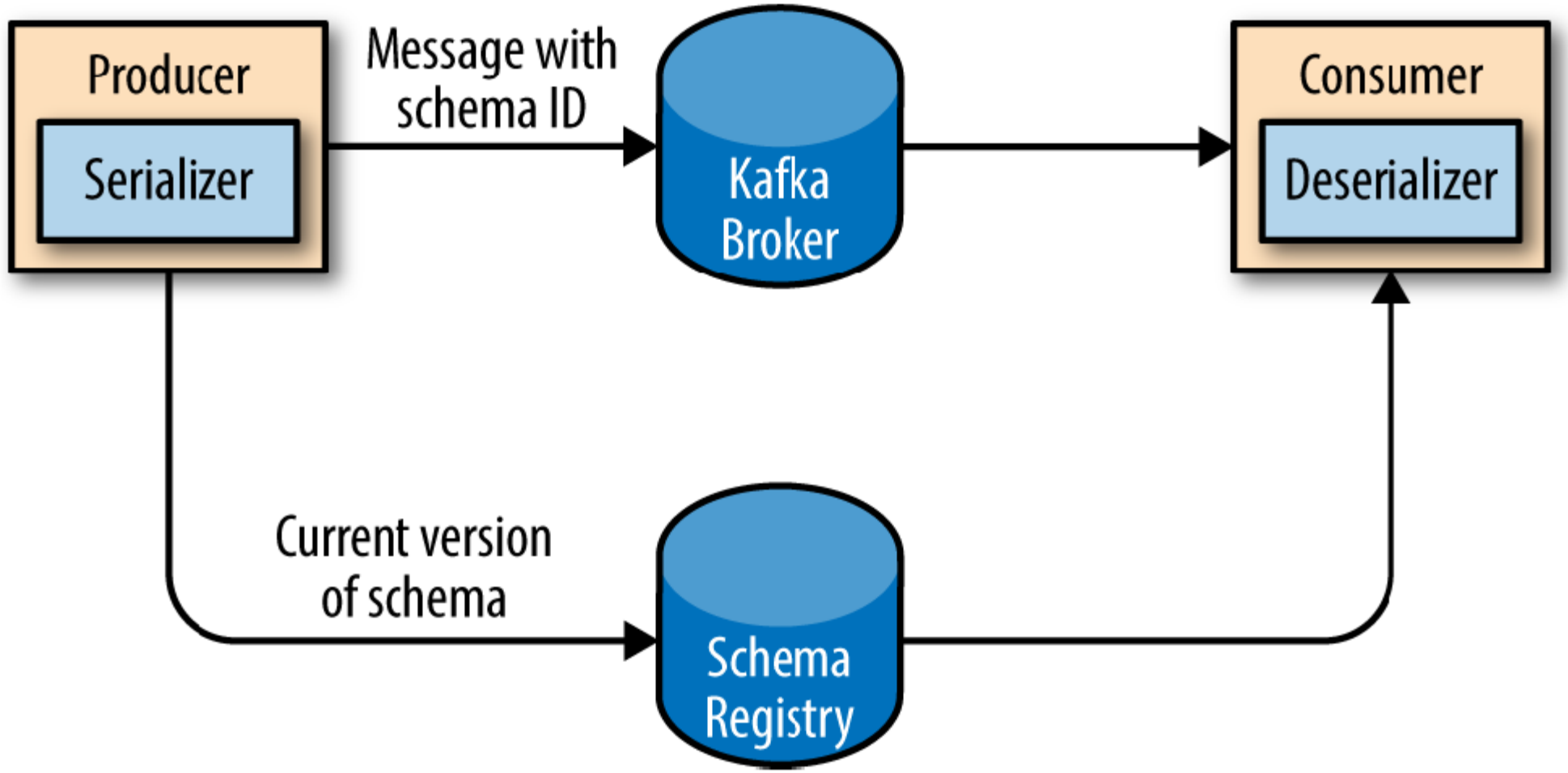
- I consumer possono quindi utilizzare l'identificatore per estrarre il record dal registro dello schema e deserializzare i dati.

- La chiave è che tutto questo lavoro - la memorizzazione dello schema nel registro e il pull up quando richiesto - viene eseguito nei serializzatori e nei deserializzatori.

- Il codice che produce dati per Kafka utilizza semplicemente il serializzatore Avro proprio come farebbe con qualsiasi altro serializzatore.

- Per Avro consultare la relativa documentazione. (http://avro.apache.org/docs/current/)

Flow diagram of serialization and deserialization of Avro records



Uso degli Avro Records con Kafka

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", schemaUrl);
String topic = "customerContacts";
int wait = 500;
Producer < String, Customer > producer = new KafkaProducer < String,Customer > (props);
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer " +  customer.toString());
    ProducerRecord < String,
    Customer > record = new ProducerRecord <  > (topic, customer.getId(), customer);
    producer.send(record);
}
```

Uso degli Avro Records con Kafka

- Usiamo KafkaAvroSerializer per serializzare i nostri oggetti con Avro. Si noti che AvroSerializer può anche gestire le primitive, motivo per cui in seguito possiamo usare String come chiave record e l'oggetto Customer come valore.

- schema.registry.url è un nuovo parametro che indica semplicemente dove archiviamo gli schemi.

- Il cliente è il nostro oggetto generato. Diciamo al producer che i nostri record conterranno il Cliente come valore.

- Istituiamo anche ProducerRecord con Cliente come tipo di valore e passiamo un oggetto Cliente durante la creazione del nuovo record. Inviamo il record con il nostro oggetto Cliente e KafkaAvroSerializer gestirà il resto.

- Cosa succede se si preferisce utilizzare oggetti Avro generici piuttosto che gli oggetti Avro generati? Nessun problema. In questo caso, bisogna solo fornire lo schema:

Uso degli Avro Records con Kafka

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url);
String schemaString = "{\"namespace\": \"customerManagement.avro\",
    \"type\": \"record\", " + "\"name\": \"Customer\"," + "\"fields\": [" +
    "{\"name\": \"id\", \"type\": \"int\"}," + "{\"name\": \"name\", \"type\": \"string\"}," +
    "{\"name\": \"email\", \"type\": [\"null\",\"string\"], \"default\":\"null\" }" + "]}";
Producer < String, GenericRecord > producer = new KafkaProducer < String, GenericRecord > (props);
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(schemaString);
for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
    String name = "exampleCustomer" + nCustomers;
    String email = "example " + nCustomers + "@example.com";
    GenericRecord customer = new GenericData.Record(schema);
    customer.put("id", nCustomer);
    customer.put("name", name);
    customer.put("email", email);
    ProducerRecord < String,
    GenericRecord > data = new ProducerRecord < String,
    GenericRecord > ("customerContacts", name, customer);
    producer.send(data);
}
```

Uso degli Avro Records con Kafka

- Usiamo ancora lo stesso KafkaAvroSerializer e forniamo l'URI dello stesso registro dello schema.
- Ma ora dobbiamo anche fornire lo schema di Avro, poiché non è fornito dall'oggetto generato da Avro.
- Il nostro tipo di oggetto è un Avro GenericRecord, che inizializziamo con il nostro schema e i dati che vogliamo scrivere.
- Il valore di ProducerRecord è semplicemente un GenericRecord che coordina il nostro schema e i nostri dati.
- Il serializzatore saprà come ottenere lo schema da questo record, memorizzarlo nel registro dello schema e serializzare i dati dell'oggetto.

UNIT

Partitions

Partitions

- Negli esempi precedenti, gli oggetti ProducerRecord che abbiamo creato includevano un nome argomento, una chiave e un valore.

- I messaggi Kafka sono coppie chiave-valore e mentre è possibile creare un ProducerRecord con solo un argomento e un valore (con la chiave impostata su null per impostazione predefinita) la maggior parte delle applicazioni produce record con chiavi.

- Le chiavi hanno due obiettivi:
  - sono informazioni aggiuntive che vengono archiviate con il messaggio
  - vengono anche utilizzate per decidere in quale partizione dell'argomento verrà scritto il messaggio.

- Tutti i messaggi con la stessa chiave andranno sulla stessa partizione.

Partitions

- Ciò significa che se un processo legge solo un sottoinsieme delle partizioni in un argomento tutti i record per una singola chiave saranno letti dallo stesso processo.

- Per creare un record di valore-chiave, è sufficiente creare un ProducerRecord come segue:

```
ProducerRecord<Integer, String> record = new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "USA");
```

Quando si creano messaggi con una chiave null, è possibile semplicemente lasciare la chiave fuori:

```
ProducerRecord<Integer, String> record = new ProducerRecord<>("CustomerCountry", "USA");
```

In questo caso, la chiave verrà semplicemente impostata su null, il che potrebbe indicare che il nome di un cliente mancava in un modulo.

Partitions

- Quando la chiave è nulla e viene utilizzato il partizionatore predefinito, il record verrà inviato a una delle partizioni disponibili dell'argomento in modo casuale. Un algoritmo round robin verrà utilizzato per bilanciare i messaggi tra le partizioni.

- Se esiste una chiave e viene utilizzato il partizionatore predefinito, Kafka eseguirà l'hashing della chiave (utilizzando il proprio algoritmo hash, quindi i valori di hash non cambieranno quando Java viene aggiornato) e utilizzerà il risultato per mappare il messaggio su una partizione specifica.

- Poiché è importante che una chiave sia sempre mappata sulla stessa partizione, usiamo tutte le partizioni nell'argomento per calcolare la mappatura, non solo le partizioni disponibili.

- Ciò significa che se una specifica partizione non è disponibile quando si scrivono dati su di essa, è possibile che venga visualizzato un errore.

Partitions

- La mappatura delle chiavi sulle partizioni è coerente solo finché il numero di partizioni in un argomento non cambia.
- Quindi, finché il numero di partizioni è costante, si può essere certi che, ad esempio, i record relativi all'utente 045189 verranno sempre scritti nella partizione 34.
- Ciò consente tutti i tipi di ottimizzazione durante la lettura dei dati dalle partizioni.
- Tuttavia, nel momento in cui vengano aggiunte nuove partizioni all'argomento, questo non è più garantito: i vecchi record rimarranno nella partizione 34 mentre i nuovi record verranno scritti in una partizione diversa.
- Quando le chiavi di partizionamento sono importanti, la soluzione più semplice è creare argomenti con partizioni sufficienti e non aggiungere mai partizioni.

Implementing a custom partitioning strategy

- Finora, abbiamo discusso i tratti del partizionatore predefinito, che è quello più comunemente usato.
- Tuttavia, Kafka non ti limita a solo partizioni di hash, e talvolta ci sono buoni motivi per partizionare i dati in modo diverso.
- Ad esempio, supponiamo che tu sia un fornitore B2B e il tuo maggiore cliente sia un'azienda che produce dispositivi portatili chiamati Banane.
- Supponiamo che tu faccia così tanti affari con il cliente "Banana" che oltre il 10% delle tue transazioni quotidiane avvenga con questo cliente.
- Se si utilizza il partizionamento hash predefinito, i record Banana verranno allocati sulla stessa partizione degli altri account, con il risultato che una partizione è circa il doppio rispetto al resto.
- Ciò può causare l'esaurimento dello spazio dei server, il rallentamento dell'elaborazione,

Implementing a custom partitioning strategy

- Ciò può causare l'esaurimento dello spazio dei server, il rallentamento dell'elaborazione, ecc.

- Quello che vogliamo davvero è dare a Banana la propria partizione e quindi utilizzare il partizionamento hash per mappare il resto degli account alle partizioni.

Implementing a custom partitioning strategy

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;
public class BananaPartitioner implements Partitioner {
    public void configure(Map < String,  ?  > configs) {}
    public int partition(String topic, Object key, byte[]keyBytes,
        Object value, byte[]valueBytes,
        Cluster cluster) {
        List < PartitionInfo > partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        if ((keyBytes == null) || (!(key instanceOf String)))
            throw new InvalidRecordException("We expect all messages to have customer name as key")
                  if (((String)key).equals("Banana"))
                return numPartitions;
        // Banana will always go to last partition
        // Other records will get hashed to the rest of the   partitions
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
    }
    public void close() {}
}
```

Implementing a custom partitioning strategy

- L'interfaccia del partizionatore include i metodi di configurazione, partizione e chiusura.
- Qui implementiamo solo la partizione, anche se avremmo davvero dovuto passare il nome del cliente speciale tramite configure invece di codificarlo nella partizione.
- Ci aspettiamo solo chiavi String, quindi lanciamo un'eccezione se non è così.