## **Analogies**

Topic means project

Consumer means team member

## **Important points**

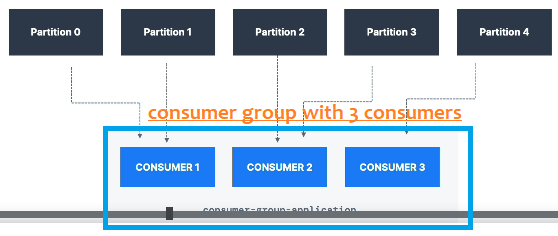
1. If 3 partitions are there in that topic, always we should have maximum of 1 consumer per partition in same consumer group , if we have 2 consumers for 1 partition, 2nd consumer will sit ideal

Let’s say if we have 10 partitions in a topic, to read data from all partitions simultaneously we should have 10 consumers in same consumer group so that there won’t be any load, 10 consumers of same group (same team ) will share the load by reading from 10 partitions

2)1 guy can work in 2 projects simultaneously, similarly 1 consumer can read from 2 partitions

3) Consumer can receive 1MB of data also at a time

Consumer groups



Each consumer can consume 2 partitions simultaneously like 1 guy working in 2 projects simutaneously

1. Sharing the messages in a partition by consumers of a group

Members in a team share their work , similarly consumers in a group share their messages

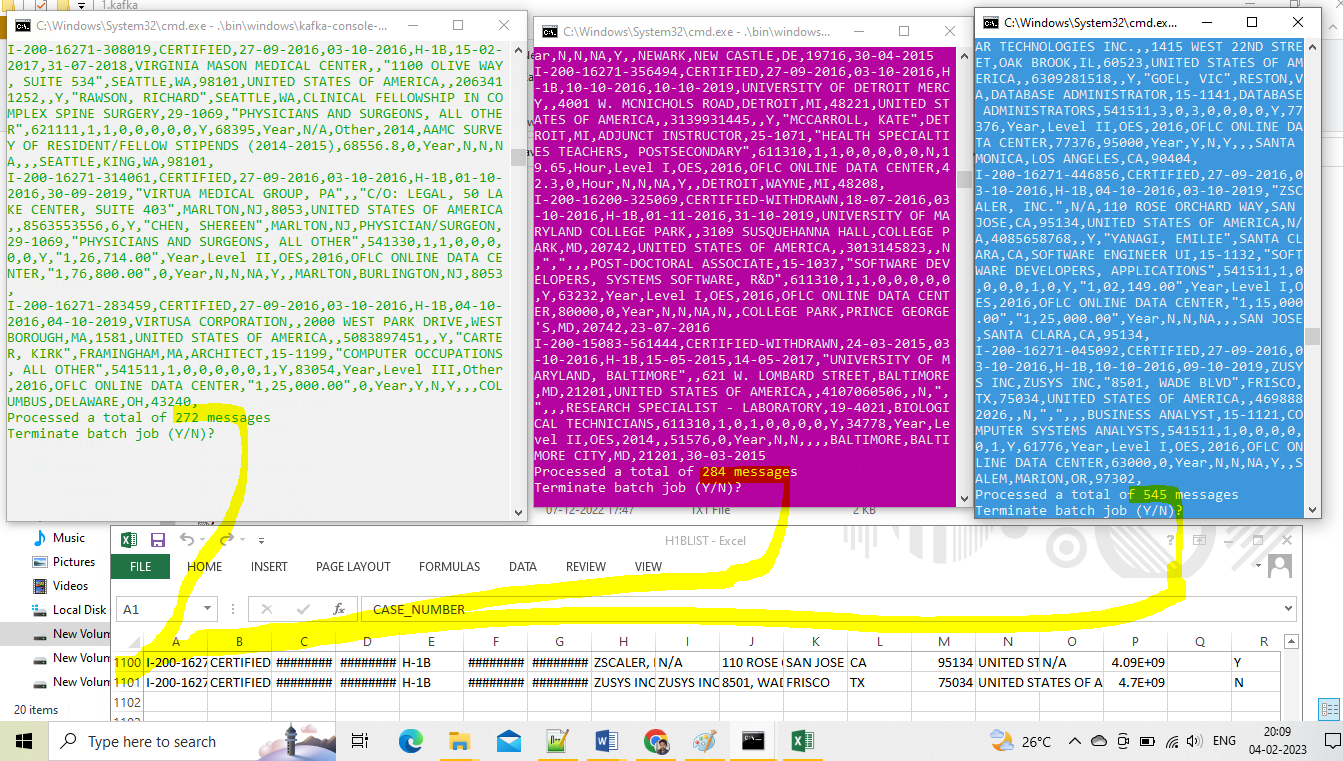
Above u can see in a topic if 10 messages came 2 messages in each partition

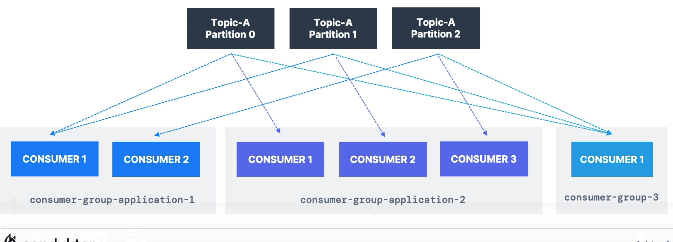
1 st consumer is responsible to consume from 2 partitions

2nd consumer is responsible to consume from 3rd,4th partitions

3rd consumer is responsible to consume from 5th,6th partitions

The below example I have sent 1101 messages to the topic with 3 partitions , and wantedly I created 3 consumers in same group they must and they will share the load because they are in same team , its like all developers in same team, since they are in same group each consumer will consumer from each partition.3 consumers will consume from 3 partitions , 1 from each partition.

All the code to create 3 console consumers and ztart zookeeper is available in another doc of sam e directory



Each consumer group is different like each consumer group is a different application

If we send 1000 messages to the topic , 1 st consumer group will consume 1000 messages and second consumer group will also consume 1000 messages and 3rd consumer group will also consume 1000 messages , because each consumer group is like a different application

Ex:- messages in the topic of 1CASM application wants by many other application

Like 1dstr application wants same data(so these guys are in separate consumer group), pops wants same data (so they are different consumer group)

Each consumer group is completely different and isolated from other consumer group

While starting a console consumer if u didn’t set the consumer group name , then it will be under temp consumer group

## **How to add multiple consumers in same group**

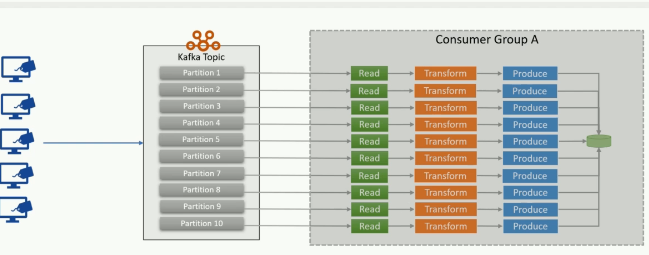
If Every consumer have same group id all consumers will join same group

Kafka will take care about that.

Always follow 1 consumer per partitions, 10 partitions means = we should have 10 consumers in same consumer group

But my doubt is if we have only 2 partitions in a topic, and if we have 1 consumer in our application and if we have 2 instances in each data center, 2- in Richardson, 2 in olathy , means totally 4 instances overall, means **then do we have 4 consumers in same consumer group for 2 partitions, 1)** Because we have only 2 topics and 4 consumers if we calculate 1 consumer for each partition ***are those 2 consumers are idle***?

So if we send a message based on the key it will go to certain partition,

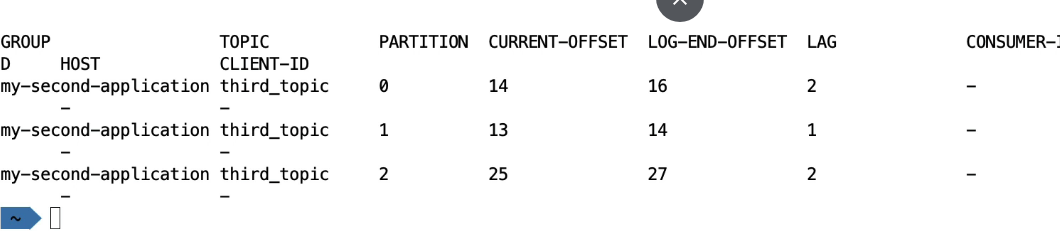


Lagging in a consumer group

I have stopped all my consumers in a consumer group and sent 5 messages to my topic and my all hence all my consumers were stopped he can’t consume those 5 messages hence my consumer group is 5 messages lagged behind the topic, then we say there is lagging

Lagging means if a consumer didn’t consume the messages that are in a topic it is called lagging

If u describe the topic you can see the column called LAG , so if u sum u will get 5 means all 5 were not consumed



What is consumer group rebalance / partition rebalance

* Every message in a segment (file) of a partition(folder) will be uniquely is identified by its offset

When a consumer in a group dies group will automatically rebalanced, then existing consumers are only responsible to read from 3 partitions while rebalancing the consumer will not consume any data, old consumer would have been reading from offset number -22 ,1st segment file, 10th folder partition 10, then that died consumer would have committed offset , will be given to existing or newly assigned consumer then he will read from that offset, to make it happen first old consumer should commit the offset ,

Like while we are working in a team , when we are leaving the company we should commit all our code then only other can take and continue from there right

See here consumer -1 died so consumer-2 only responsible to read data from 2 partitions

Now consumer-2 is reading from 2 partitions earlier it was reading only from 1 partition

This is called consumer group or partition rebalance

Offsets

Always ensure the consumer on consuming the message should auto commit the offset

Once u read the message u should commit the offsets are generally committed to the broker

we have 2 offsets called 1)current offset -- tells current offset position which consumer is currently reading

1. Committed offset- generally offsets are committed to another topic in broker , consumer should commit the offset very frequently stating I have read messages till this position , if he didn’t commit when consumer went offline to take the work by another consumer if he didn’t tell / commit till which position he has read , then new consumer should read again from beginning

Analogy:- after reading a book before closing he should place a pencil/some thing at the last page he read ,so next day when he came back he will start reading from there

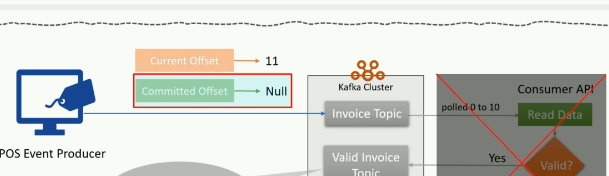
props.setProperty("enable.auto.commit", "true");

else if u didn’t remember you need to start reading from 1st page

if u set the above flag to false, then consumer after consuming the message he wont commit the offset automatically , he should issue a manual commit,

consumer.commit()

1. This committed offset is used to avoid duplicate reading
2. fetch.min.bytes: It is the min amount of bytes that a consumer will wait for server to give it will wait until min buffer capacity is reached
3. fetch.max.wait.ms It is the maximum amount of time the consumer has to wait for the broker to send back records.
4. session.timeout.ms,
5. partition.assignment.strategy: It is the strategy used by PartitionAssignor to split the partition among consumers in a consumer group.



Consumer vs consumer group

If all consumers are in same group they will share the work load,

means if there are 120 messages stored in 3 partitions (40 messages per partition) if there are 3 consumers in same consumer group , then each consumer will read from 1 partition

Since consumer -1 pointing to partition -1 , if there are 90 messages in partition-1 all 90 messages will be consumed by the consumer-1

2nd consumer will read from partition-2

3rd consumer will read from partition-3

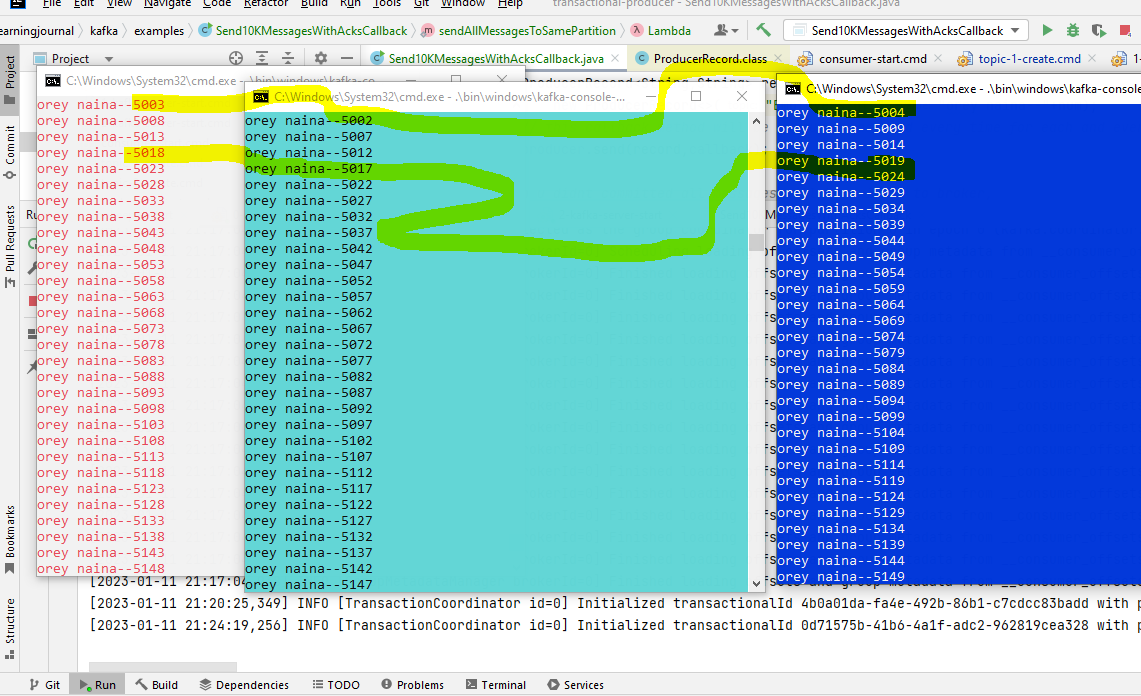
If all consumers are in different group they are treated as individual consumers

And each consumer will consume all the messages in the topic

IntStream.*range*(1,6000).forEach(num->{  
 *//Here we are sending all messages to without partition number , so that all messages will be sent to all partitions evenly*

*// and if key is there it may hash and decide the partition num , as I didn’t provide the key , my 6000 messages will be evenly distributed to all partitions*

ProducerRecord<String,String> record=  
 new ProducerRecord<>("EmployeesInfo","orey naina--"+num);  
 *// this callback method will be executed each and every time for each and every message* producer.send(record,callback);  
 });  
*//even though u didnt committed all messages will be written to broker,  
// once u committed it will change the flag to committed for each and every message* producer.commitTransaction();



See in the above I have sent 6000 messages to a topic that have 3 partitions and since all 3 consumers are in same group they started sharing the work load each like around 2000 messages

Resetting the offset to re consume same msg again

Consumer after consuming he must commit the offset to the broker, all the offsets will be stored in a separate dedicated topic inside broker, if he didn’t commit after sometime if he was down and if he came back again he will not remember how far he has read the topic so he will end up in reading the topic from starting,

solution:- commit the offsets frequently after consuming the messages and enable.auto.offset.commit=true

re consuming the same messages again

1. Don’t commit the offset ,disable auto commit , so that you can consume all messages and again and again even if u start kafka multiple times
2. If u commit the offset , stop it and reset the offset, not sure whether we can do it programmatically or not .stop the consumer and reset the offset –we can stop the consumer and read all messages again consProperties.put(“auto.offset.reset”,” earliest”); this is applicable only whenu don’t commit offset

if we keep earliest in local for debugging purpose everytime if we are starting and stopping if we already consumed all messages of u want to read all those again just start and stop so 2nd time while the server is restarting offset will be reset to the topic’s first message so u can consume same messages again and again on every restart, generally also we can reset the offset only once u stopped the consumer

this is very dangerous do do in production if we start and stop our app for deployment purpose it will consume all the messages again present in the topic

make sure this earliest is only for local development purpose

1. use new consumer group – which we cant do in production

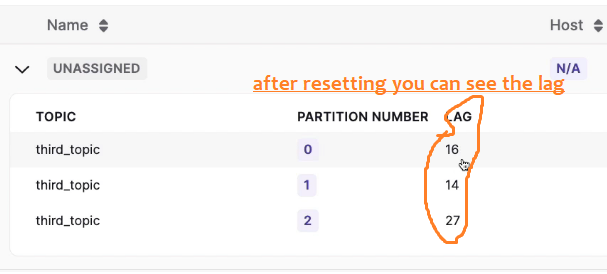
In local easiest way is to use new consumer group so that it will start freshly and it will read all messages in the topic

Let’s say if u have already consumed 500 messages inside a partition topics and you would have committed the offsets for those messages already, if u want to consume all those again best option is we should reset the offset , you can reset offset only when consumer is stopped ,

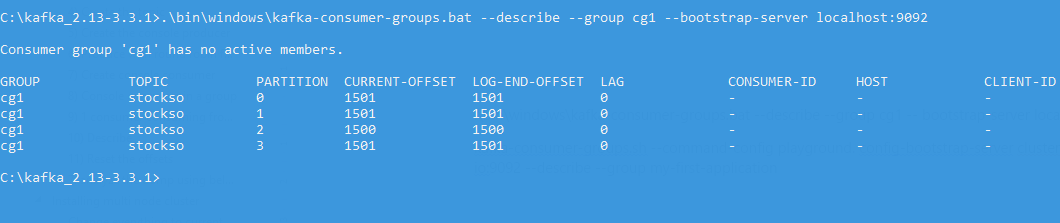
If consumer is running u should be committing the offsets for the consumed messages and parallelly u cant reset the offset so better stop the consumer and then reset the offsets

Script to reset the offset in given in another doc

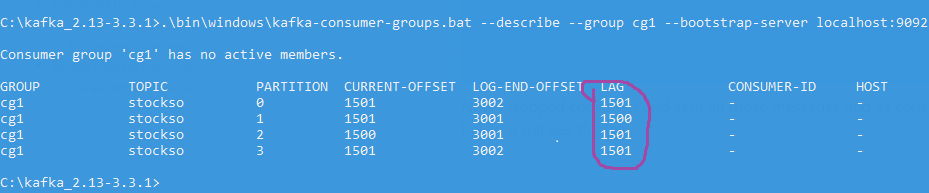
Before reset there is no lag ,because consumer is always consuming all the messages present inside partition and hence there is no lag, but after resetting the offsets are set to starting position and consumer didn’t consume all those messages hence u are seeing the lag in below snip



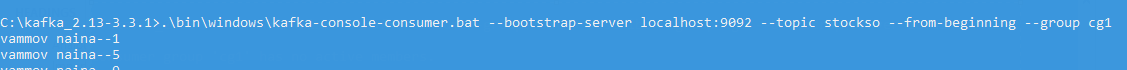
Before these is no lag, because 1st time I sent 6000 messages and consumed all of them



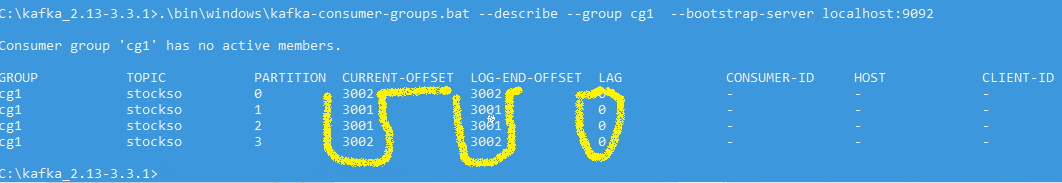
Now I stopped consumer and sent 6000 messages and as consumer was offline he didn’t consumed, now in below u will see the lag for 6000 messages I have sent them programmatically



Now I have started consumer and all those above lag messages will be consumed



Now requirement is so far I have sent 12,000 messages to the topic and consumed all of them so lag is zero and offsets are also committed and now and if u want to re consume all of those 12,000 messages again there are 2options 1 is start a new consumer group or **2nd option if u don’t want to change the consumer group then** Reset the offset, to reset the offset now see the current offset is 12,000 3k\*4



Now issued reset



After reset all the new offset are pointing to zero now start consuming with same consumer group

Challenges for producer –solution for it

**Duplicate messages from producer side** If producer wanted to send 100 messages and broker failed at 65th message or if acknowledgement was lost for 1 message then kafka will resend all messages for which acknowledgement is not received .

***Solution:- send them in a kafka transaction*** , if exception raised at 65th message then tx will not be committed , so even broker received the messages until those were committed those will be in aborted mode.

Challenges for consumer -solution

Problem:- if consumer consumed 15 messages and didn’t committed the offset and if it went down and came back after some time since nothing were committed all those messages will be read again

1 solution:- commit offsets frequently

the reason for duplicate processing is –it is not committing the offsets frequently set the flags accordingly to commit the offsets frequently, because if it would have committed the offset and went down and if consumer came back after some time it will take latest offset position and it will read from there

2nd solution : -use kafka streams

Properties consumerProps = new Properties();

consumerProps.put(ConsumerConfig.CLIENT\_ID\_CONFIG, AppConfigs.applicationID);

consumerProps.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, AppConfigs.bootstrapServers);

consumerProps.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

consumerProps.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.class);

consumerProps.put(JsonDeserializer.VALUE\_CLASS\_NAME\_CONFIG, PosInvoice.class);

consumerProps.put(ConsumerConfig.GROUP\_ID\_CONFIG, AppConfigs.groupID);

consumerProps.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

KafkaConsumer<String, PosInvoice> consumer = new KafkaConsumer<>(consumerProps);

consumer.subscribe(Arrays.asList(AppConfigs.sourceTopicNames));

while (true) {

ConsumerRecords<String, PosInvoice> records = consumer.poll(Duration.ofMillis(100));

This is like in that 100 milli seconds it will fetch how many records it can receive from the broker

for (ConsumerRecord<String, String> record : records)

{

System.out.println("topic = "+record.topic()+

"partition = "+record.partition()+

"offset = "+record.offset()+

"customer = "+record.key()+

"country = "+record.value());

}

}

* The poll loop handles coordination between consumers in the consumer group, partition rebalancing, heartbeats and fetching data from servers.
* The poll() method returns data from the topic partitions assigned to it.

What happens if a consumer suddenly went down

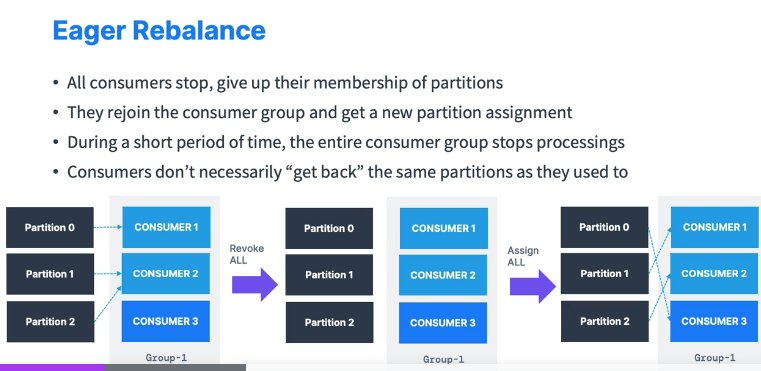
If initially 3 consumers are there in a consumer group reading from 3 partitions 1st consumer might have read 40 offsets positions and second consumers might have read 300 offset positions , after reading if consumer went down that and if we gave that work to new consumer or existing consumer he will read from that offset position

Cons group rebalancing stratagies

1. Eager rebalance

The main problem with Eager rebalance is here when a consumer suddenly joins the group

* 1. The group will rebalance and during this time consumer will not accept any message this is called “stop the world“ which is not good in this case, once after rebalancing completes all consumers will get new partitions , even existing consumer who are reading from some partitions even those will be assigned with new partitions

--

### **2) co-operative rebalance**

This is latest and This will avoid “Stop the world”, it is incremental rebalance , existing consumer will still point to the

Same partitions , new consumers will get new partition slowy re assignment will happen

Kafka Consumer:this is the key “**partition.assignment.strategy”**

• RangeAssignor: assign partitions on a per-topic basis (can lead to imbalance) this is eager rebalance and causes stop the world

• Round Robin: assign partitions across all topics in round-robin fashion, optimal balance this is eager rebalance and causes stop the world

• StickyAssignor: this is eager rebalance and causes stop the world, and this is balanced like Round Robin, and then minimises partition movements when consumer join/leave the group in order to minimize movements

• CooperativeStickyAssignor: This is a newer strategy rebalance strategy is identical to StickyAssignor but supports cooperative rebalances and therefore consumers can keep on consuming from the topic

• The default assignor is [RangeAssignor, CooperativeStickyAssignor], which will use the RangeAssignor by default, but allows upgrading to the Cooperative StickyAssignor with just a single rolling bounce that removes the RangeAssignor from the list.

### **3. Static group membership**

If u specify “group.instance.id”=”some UUID” it makes a consumer static member

The advantage is for static members partitions will not be re assigned immediately

If u configure “session,timeout.ms”=200 until that 200 ms partition rebalance /consumer group rebalance will not trigger , if the consumer didn’t join in that specified time then rebalance will trigger immediately

Sample code for consumer

Properties props = new Properties();

props.setProperty("bootstrap.servers", "localhost:9092");

props.setProperty("group.id", "test");

props.setProperty("enable.auto.commit", "false");

props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

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

consumer.subscribe(Arrays.asList("foo", "bar"));

final int minBatchSize = 200;

List<ConsumerRecord<String, String>> buffer = new ArrayList<>();

while (true) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

buffer.add(record);

}

if (buffer.size() >= minBatchSize) {

insertIntoDb(buffer);

consumer.commitSync();

buffer.clear();

}

}

Precautions while consuming follow these

* 1. Validate the incoming messages before u process them
  2. Make sure you handle the poison pill scenario (De serialization exception) like when u configure the value deserializer as string deserializer and when avro object came it might throw de serialization exception , so to avoid configure proper spring error handling deserializer and delegate class deserializer
  3. Store the poison pills/ undeserializable messages into dead letter topics
  4. Proper exception handling
  5. Retry mechanisms with proper exponential backoff’s and

Consumer properties

Even consumer.close() will commit the offsets

Properties consProperties=new consumerProperties();

kafkaConsumer kc=new kafkaconsumer(consProperties);

1. consProperties.put(“key.deserializer”,StringDeserializer.class.getName());
2. consProperties.put(“value.deserializer”,StringDeserializer.class.getName());
3. consProperties.put(“auto.offset.reset”,”none/earliest/latest”);

earliest is very dangerous , everytime we start the consumer earliest means its like read from the beginning , suppose in production for the 1st time we went and already consumed millions of messages for a minor enhancement if we went down and came back if we keep this property that’s it , eventhough we already processed all messages it will start reading again all the messages present in the topic, so it’s a duplicate processing of all messages

1. “group.instance.id”=”some UUID” if u give this then it will become a static group member and for static group member u can give session,timeout.ms”=200 means partition rebalance will not trigger untill for that 200 ms,

Configuring consumer

* fetch.min.bytes: It is the minimum amount of data in topic partitions in a broker that a consumer can request for. If the amount of data in broker's topic partition is **less than fetch.min.bytes**, the broker will wait till enough data flows into the topic before sending the records to the consumer. This helps in reducing the load on both consumer and broker.
* fetch.max.wait.ms: It is the maximum amount of time the consumer has to wait for the broker to send back records. The consumer will wait till **fetch.max.wait.ms** for the broker to send back the record if there is not enough data. After that, it will again request broker for the record.
* max.partition.fetch.bytes: This property is used to control the maximum number of bytes the server will return per partition.
* session.timeout.ms: It is the maximum amount of time a consumer can stay connected to the broker without sending heartbeats to the group coordinator. When session.timeout.ms passes without consumer sending heartbeats to group coordinator, it is considered to be dead, and group coordinator will trigger a partition re-balance.
* partition.assignment.strategy: It is the strategy used by PartitionAssignor to split the partition among consumers in a consumer group. It can be given with four values.

1. **Range**: Here, a consecutive set of partitions of a topic are split among the consumers in the consumer group. Example, if topic1 and topic2 have 3 partitions respectively. And, c1 and c2 are two consumers in the consumer group cg1 that subscribed to both topic1 and topic2. Using range strategy, c1 will receive two partitions from topic1 and topic2, and c2 will receive one from topic1 and topic2.
2. **Round Robin**: Assigns partitions to consumers in consumer group sequentially. Example as in the above one, partition 0 and partition 2 of topic1 will be assigned to c1 and partition1 to c2. Also, partition 0 and partition2 of topic2 will be assigned to c2 and partition1 of topic2 will be assigned to c1.
3. **StickyAssignor**: Guarantees an assignment that is maximally balanced while preserving as many existing partition assignments as possible.
4. **CooperativeStickyAssignor**: Follows the same StickyAssignor logic, but allows for cooperative rebalancing.

## Facts about consumer

* 1. If you don’t commit the offsets and if u start and stop u will consume all messages once again, to avoid this scenario only we should commit the offsets frequently
  2. Not sure what is the use of giving unique cousmer in for each consumer instance