## **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 and 1 consumer can read from 2 topics as below

KafkaConsumer cons=new KafkaConsumer();

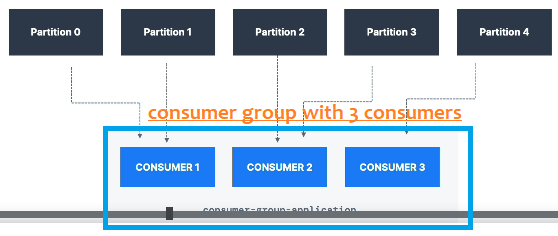
Cons.subscribe( List.of(“Topic1”,”topic2) );

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

References

<https://github.com/manideep-vv/Forked-spring-kafka/tree/main/samples>

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

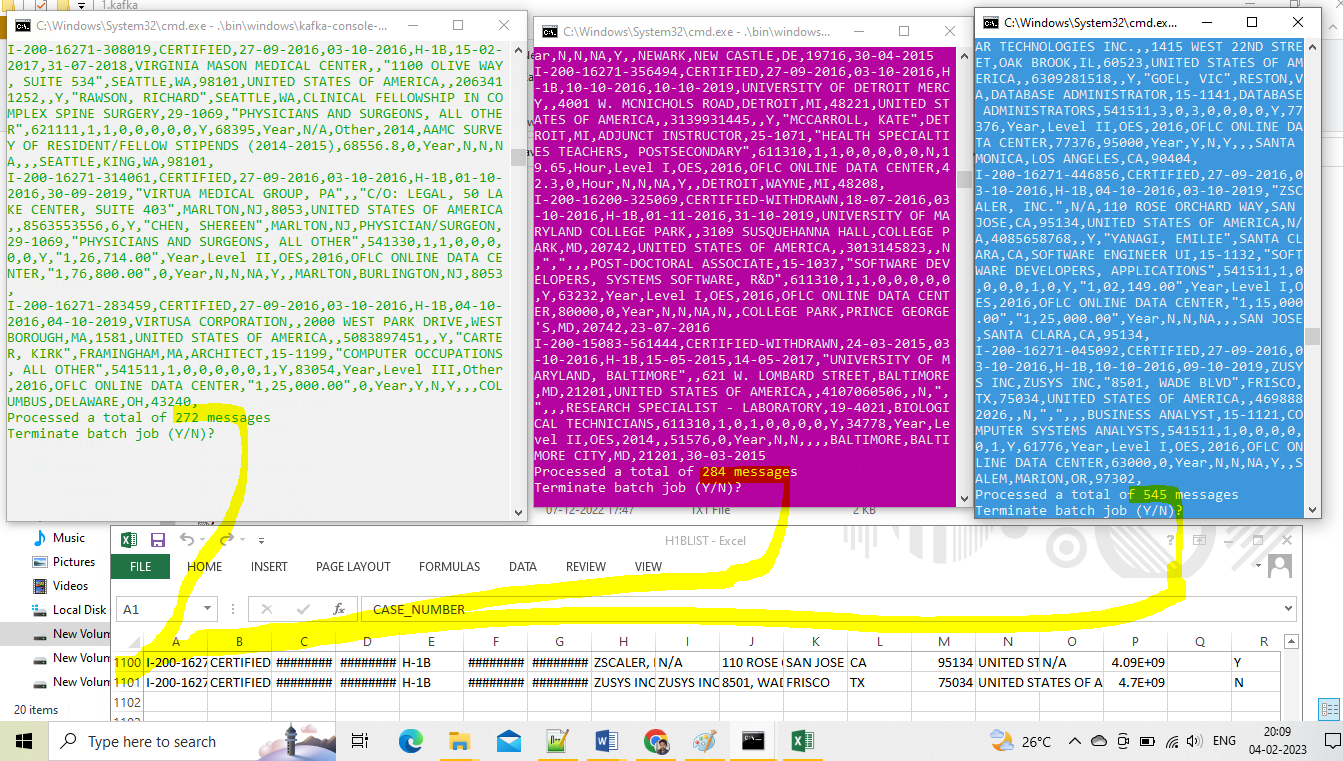
Above u can see in a topic if 10 messages came 2 messages in each partition

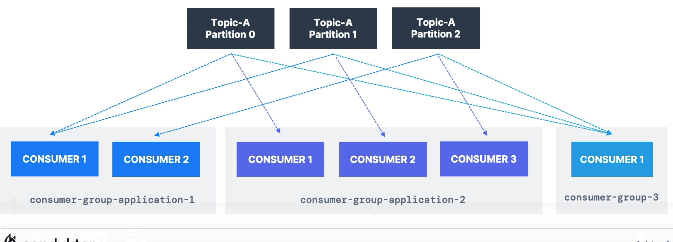
1 st consumer is responsible to consume from 2 partitions

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**

Ideally 1 consumer group is treated as 1 separate application, If Every consumer have same group id all consumers will join same group,

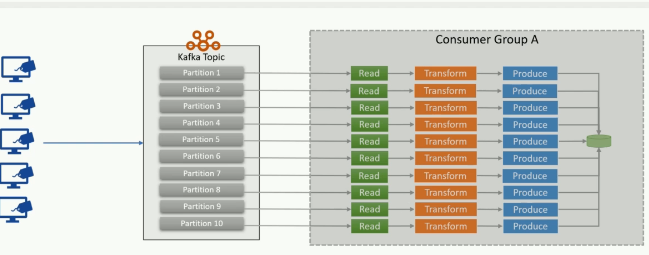
Kafka will take care about that.

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

If we have 2 consumers in same group for 1 partition, as mostly 1 consumer will read from 1 partition, another consumer will be idle

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***?yes

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

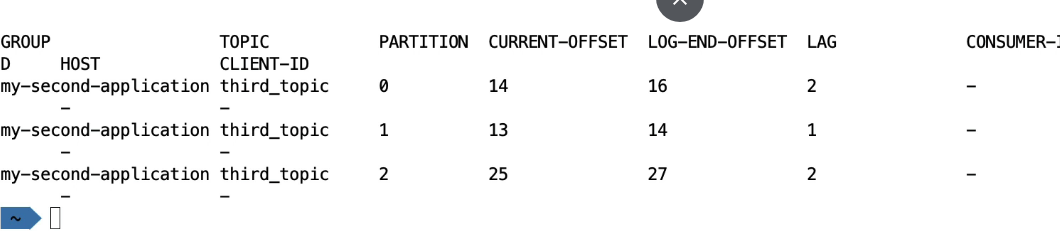


Lagging in a consumer group

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 means this is stored in the 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

As the value of committed offset is stored in broker, even if consumer restarts consumer will exactly read from where it is last read

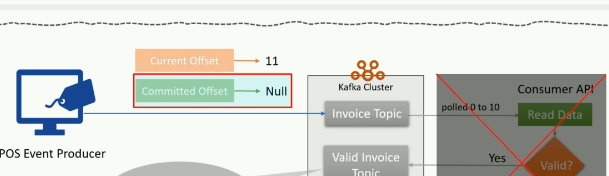
props.setProperty("enable.auto.commit", "true");

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

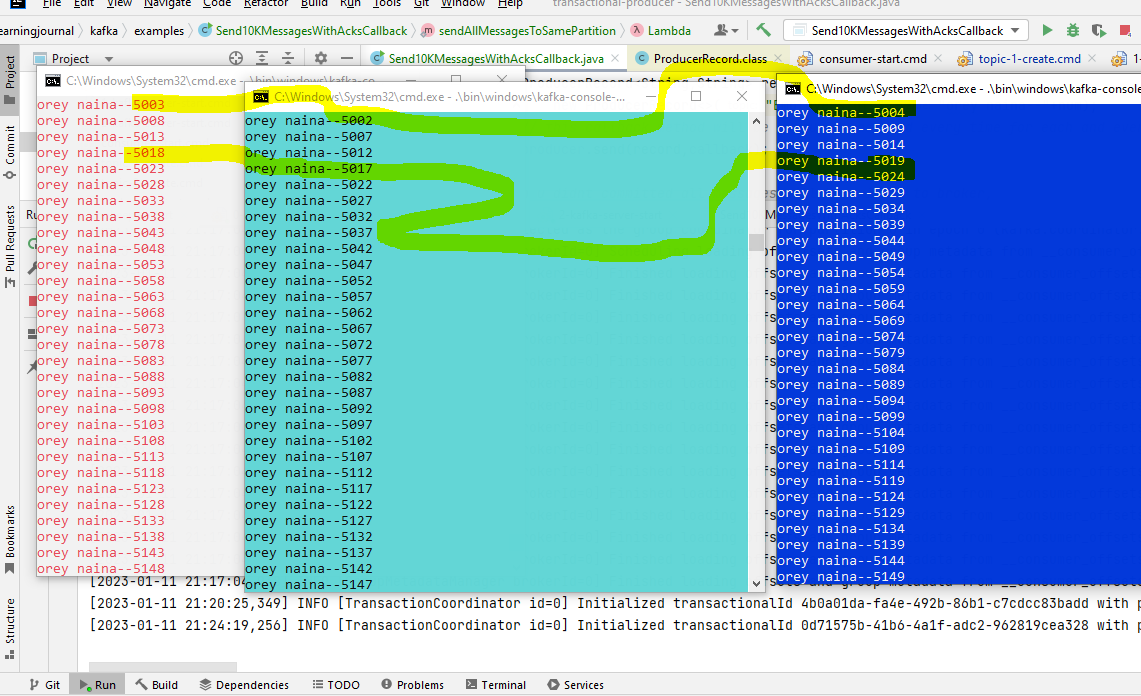
If all consumers are in different group they are treated as individual consumers

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

p.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*,false);

1. 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

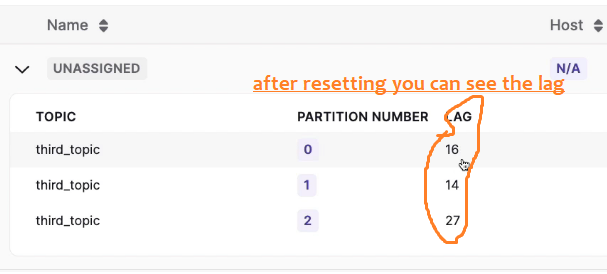
In local easiest way is to use new consumer group so that it will start freshly and it will read all messages in the topic

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 ,

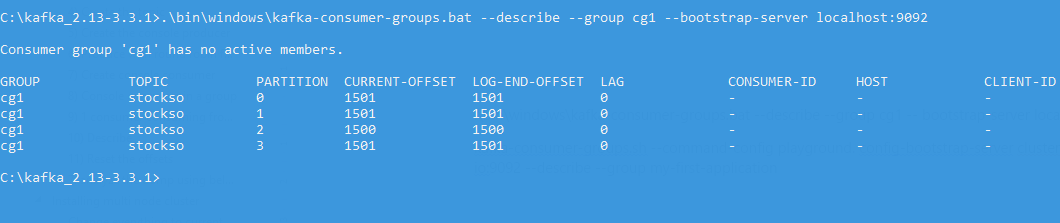
If consumer is running u should be committing the offsets for the consumed messages and parallelly u cant reset the offset so better stop the consumer and then reset the offsets

Script to reset the offset in given in another doc

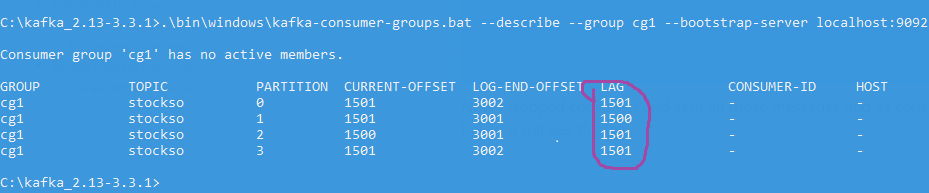
Before reset there is no lag ,because consumer is always consuming all the messages present inside partition and hence there is no lag, but after resetting the offsets are set to starting position and consumer didn’t consume all those messages hence u are seeing the lag in below snip



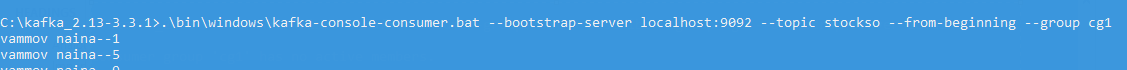
Before these is no lag, because 1st time I sent 6000 messages and consumed all of them



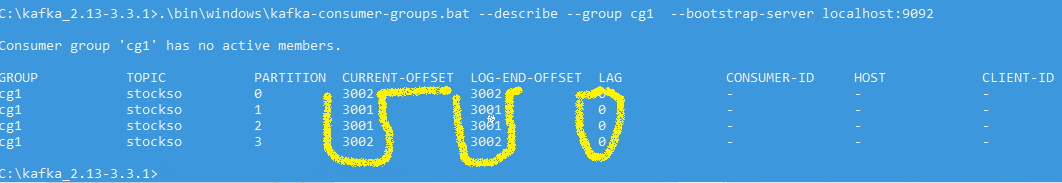
Now I stopped consumer and sent 6000 messages and as consumer was offline he didn’t consumed, now in below u will see the lag for 6000 messages I have sent them programmatically



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

Resetting to Earliest vs latest

1. If u are going to production newly with a new consumer group

and if u want to consume all those messages that are there in the production topic(because generally as per retention policy messages will stay in the topic for a 7-10 days

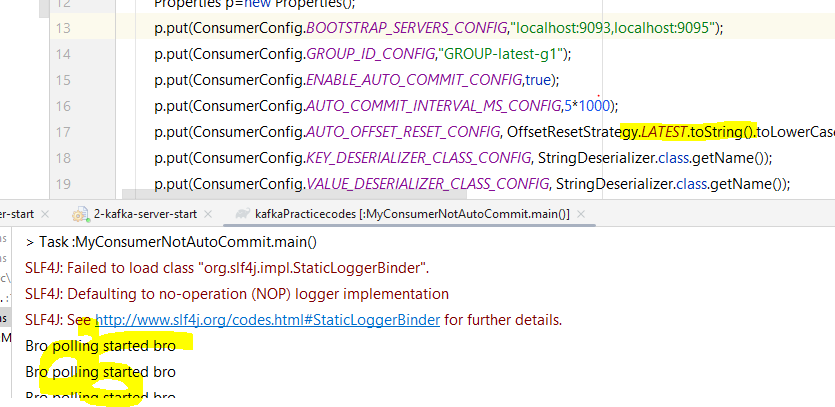
so in this case use earliest, it will consume all messages in the topic, if u keep latest means it will ignore all existing messages present in topic, it will be consuming only those messages which were sent after that consumer started

Latest means- it wont consider older messages already present, once if started after that if any message comes only those it will consume

**Demo for resetting to latest –losing existing msgs**

p.put(ConsumerConfig.*GROUP\_ID\_CONFIG*,"tumbri GROUP");  
p.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*,true);  
p.put(ConsumerConfig.*AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG*,5\*1000);  
p.put(ConsumerConfig.*AUTO\_OFFSET\_RESET\_CONFIG*, OffsetResetStrategy.*LATEST*.toString().toLowerCase());

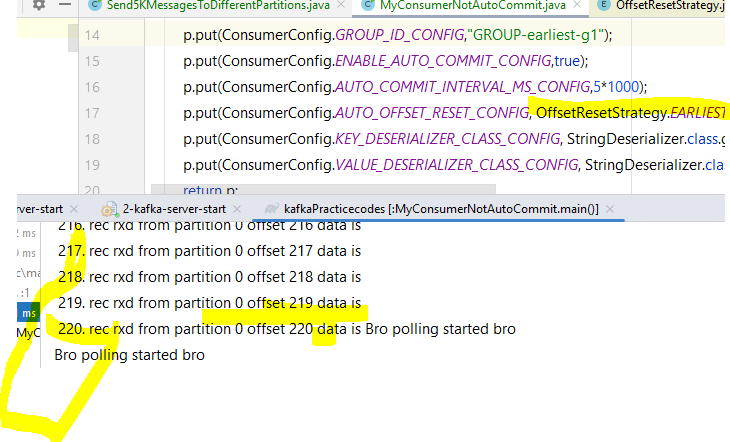
Before starting the consumer I have pushed 200 messages in the topic ,since offset reset is to latest as its starting for the first time it will reset the offset to latest position 201 and it will not consume the existing messages in the topic



**Demo for resetting to earliest-**

When reset to earliest as u see all 200 messages are received which were kept before starting this consumer,I mean first I pushed all messages to the topic and later I have started this new consumer group with earliest

I could see all messages present in the topic were read as auto.offset.reset is earliest



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

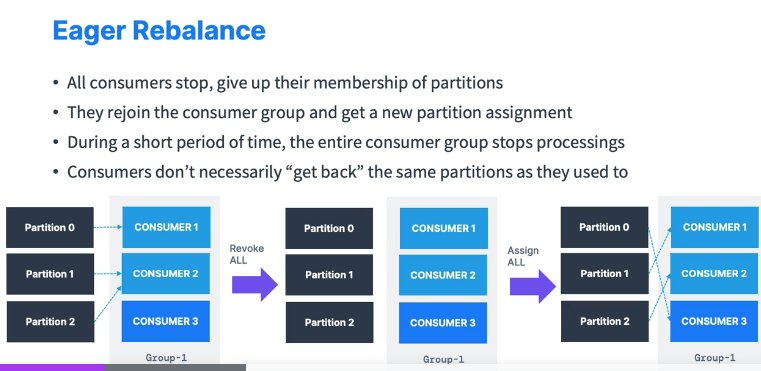
If initially 3 consumers are there in a consumer group reading from 3 partitions 1st consumer might have read 40 offsets positions and second consumers might have read 300 offset positions , after reading if consumer went down that and if we gave that work to new consumer or existing consumer he will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

Batch Processing in Kafka

<https://medium.com/@shrutishrm17/kafka-batch-processing-in-spring-boot-fc6c58f857fa>

<https://hevodata.com/learn/kafka-batch-processing/>

1. Create a consumerConfigs with these properties

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, "batch");

props.put(ConsumerConfig.MAX\_POLL\_RECORDS\_CONFIG, "4");

return props;

1. Modify the factory as

factory.setBatchListener(true);

1. Modify the consumer as listening to list of consumerRecords

@KafkaListener(id = "kafka-batch", topics = "${kafka.topic.app}")

public void receive(ConsumerRecords<String> messages,

While doing batch processing better disable auto.commit

Because lets say we have received a batch of 1000 records, and after processing 900 records if consumer went down, if u would have enabled auto commit=true and auto commit interval as 5 seconds ,then by the time u process those 900 records if it crossed 5 seconds then for that entire batch commit would have happened and we have lost 100 records ,so to avoid this one better go with manual commit

Retry mechanisms

<https://betterprogramming.pub/spring-boot-kafka-non-blocking-retries-a-hands-on-tutorial-a0c425acc3dd>

Sample code for consumer

Properties props = new Properties();

p.put(ConsumerConfig.*GROUP\_ID\_CONFIG*,"infosys-group");// every consumer belong to one group

p.put(ConsumerConfig.*CLIENT\_ID\_CONFIG*,"cl1");//   
p.put(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*,"localhost:9092,localhost:9093");  
p.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*,true);

p.put(ConsumerConfig.*AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG*,30\*1000);

p.put(ConsumerConfig.*AUTO\_OFFSET\_RESET\_CONFIG*,"latest");/earliest  
p.put(ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class.getName());  
p.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class.getName());

p.put(JSON\_Deserializer.*VALUE\_CLASS\_NAME\_CONFIG*, Employee.class);//means after deserializing the data will be binded to this class

prop.put(Consumer\_Config.MAX\_POLL\_RECORDS\_CONFIG, "5");//means in single fetch only max this much size of records will be fetched

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();

}

}

Using spring annotations

@KafkaListener(id = "qux", topicPattern = "myTopic1")

public void listen(@Payload String foo,

@Header(name = KafkaHeaders.RECEIVED\_KEY, required = false) Integer key,

@Header(KafkaHeaders.RECEIVED\_PARTITION) int partition,

@Header(KafkaHeaders.RECEIVED\_TOPIC) String topic,

@Header(KafkaHeaders.RECEIVED\_TIMESTAMP) long ts

) {

...

}

* KafkaHeaders.OFFSET
* KafkaHeaders.RECEIVED\_KEY
* KafkaHeaders.RECEIVED\_TOPIC
* KafkaHeaders.RECEIVED\_PARTITION
* KafkaHeaders.RECEIVED\_TIMESTAMP
* KafkaHeaders.TIMESTAMP\_TYPE

Kafka exception handling

When error occurred, While storing the error messages into mongodb ,

Store details with application name, topic name, partition number, offset number and detailed stack trace

Features

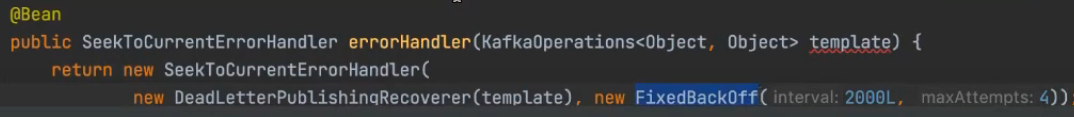
1. Default exception handler-retry 9 times :-- If any exception occurred or if any unknown magic byte came? while processing the message, Out of the box code says ,it will additionally retry 9 times total 10 times it will retry, even though it’s a null pointer exception it will retry 10 times

This is the nature of kafka / it s the out of the box feature

If anything is not worth retrying, then don’t retry

Like – Classcast Exception, MessageConversionException all these are not retryable, not worth retrying

<https://github.com/manideep-vv/Forked-spring-kafka/tree/main/samples>



Retry mechanism

If message processing fails, or if system encounters an issue the message is forwarded to a retry topic with a back off timestamp

**@RetryableTopic(attempts = 5, kafkaTemplate = "myRetryableTopicKafkaTemplate"**

**backoff = @Backoff(delay = 1000, multiplier = 2, maxDelay = 5000))**

@KafkaListener(topics = "my-annotated-topic")

The above is an exponential backoff first time it will retry after 1 second

2nd time it will retry after 2 (because multiplier is 2),next time delay is 2\*2=4 seconds

Next time it will retry after 8 seconds,but maximum delay is 5 seconds ,it will stop retry incase max delay is >5 seconds

When a record fails it will be pushed to retry topic and the listener of that topic will listen after certain delay and it will retry for certain times

Check if it automatically creates topics on its own or not?

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,

#### Enable.auto.commit

p.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG,false);

when consumer enables auto commit , for every message he consumes he will give the acknowledgement back and he will commit the offset Generally when the consumer commits the offsets , all those offsets will be stored in the kafka broker topic

, if he doesn’t commit the offset means he is saying I have not received the message, so when consumer was down and came back all the messages will be send back again

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/Guidelines

* 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
  3. Set proper data retention period & proper offset retention period
  4. Ensure auto offset reset period is either latest,earliest,none
  5. Incase of unexpected behaviour use replay capability

Consumer properties

1. Auto commit

p.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG,false);

p.put(ConsumerConfig.AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG,5\*1000);

if u enable the auto commit , automatically after above mentioned time offsets will be committed automatically

Modify the auto commit interval time to longer so that it should commit after processing all the records in a batch

Auto.commit.interval.ms=millis to process the batch

If u don’t know then prefer manual commit

1. auto.offset.reset

Assume we are consuming data from topic for first time in production, ours will be a new consumer group

When we gave auto.offset.reset=earliest , Then it will read entire data from starting

When we gave auto.offset.reset=latest , Then it will data from last

Assume u are consuming from some time suddenly if u down the application, then as kafka frequently commits the offsets, when it is up after 1 or 2 days it will automatically finds the latest offset commit and it will resume after that

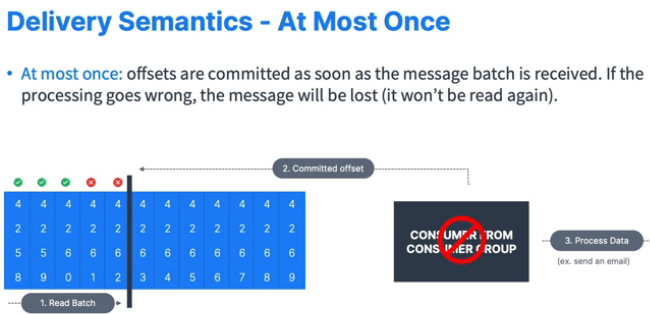
1. Delivery semantics

In case of atmost once –There will be data loss-Never twice ,sometimes 0

In case of atleast once- There will be duplicate processing of messages

At most once: this is due to the auto commit enabled and auto commit interval is 5 seconds

After 5 seconds for that entire batch all those messages will be auto committed



We can implement atMostOnce by enabling auto commit=true, with this loosing messages is possible, if we received 100 messages and we have enabled auto commit interval as 5000 ms /5sec

Within this 5 sec u should process 100 messages whether u process or not It will be auto committed and if consumer went down after auto committing that’s it that’s it messages are lost

There should be atmost care while using batch processing, while processing a batch if u choose auto commit, lets say u have received a batch of 5, after successfully processing 3/5 records in a batch if consumer went down after 10 seconds and due to “auto.commit.interval.ms=5 milliseconds” p.put(ConsumerConfig.*AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG*,5\*1000); without processing those 2 records itself the consumer went down and for those entire batch of 5 offsets were committed even after consumer came back online since the offsets were committed those 2 records will not be consumed again

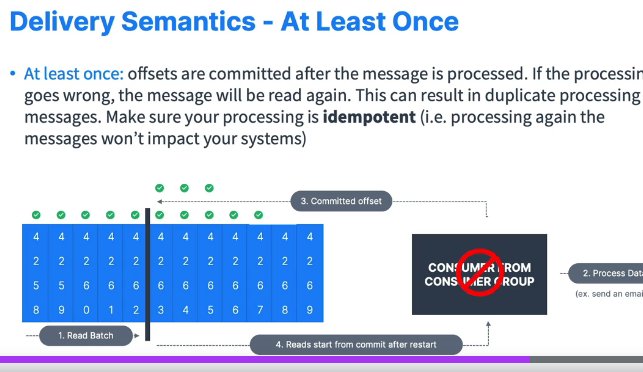
So this is how we can lose the data in atmost once

We can implement at Least Once by enabling auto commit=false

Or enabling manual commit

Here if u disable auto commit and enable manual commit, lets say u received 900 records and processed 800 records and all those were successfully saved in the database and now it went offline, we have written the logic of commit only after processing all 900 records

since those 100 rec processing didn’t got completed we didn’t committed so now when consumer is back offline then we will receive all 900 records this is called atLeast once scenario



But I don’t know how to make consuming process idempotent

Delivery semantics for consumers - Summary

• At most once: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).

At least once (preferred): offsets are committed after the message is processed. If the

processing goes wrong, the message will be read again. This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won't impact your systems)

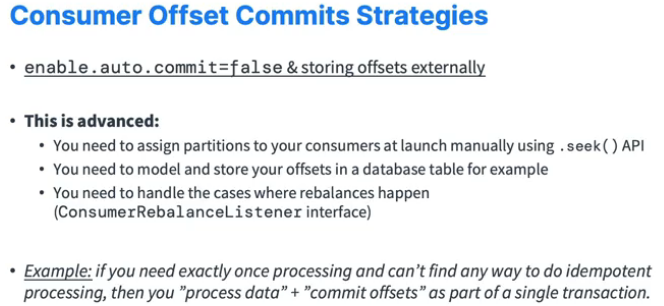
Exactly once: Can be achieved for Kafka Kafka workflows using the Transactional

=>

API (easy with Kafka Streams API). For Kafka => Sink workflows, use an idempotent

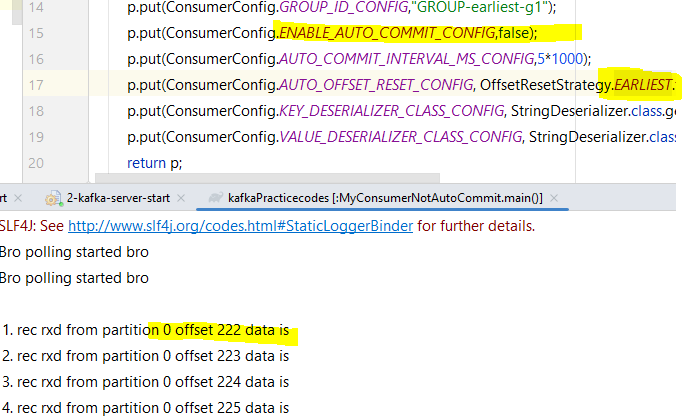
consumer.

**Bottom line: for most applications you should use at least once processing (we'll see in practice how to do it) and ensure your transformations / processing are idempotent**



1. Disabling the auto commit

p.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*,false);



in my program I have disabled auto commit from offset 222, means initially when I ran the program I ran with enable auto commit then every message for each message automatically offsets are committed till 221 th message ,later I disabled and ran the program ,

I stopped consumer and pushed 60 messages from offset 220 i.e., they have occupied till offset 280

now everytime I start the program with auto commit as false ,it will consume from offset 220 till 280 since auto commit is false I wont acknowledge, and keeps on polling and I will terminate & restarts again and again, it will start reading from offset 220 as for those messages 220 -280offsets are not committed

since offsets are not committed due to disabling auto commit ,after consuming messages offsets wont be committed automatically, every time we stop & start application freshly since for those 60 messages offsets are not committed it will start reading those 60 msgs each and every timeuntill u manually commit the offsets

1. Verify the lag

p.put(ConsumerConfig.*ENABLE\_AUTO\_COMMIT\_CONFIG*,false);  
p.put(ConsumerConfig.*AUTO\_COMMIT\_INTERVAL\_MS\_CONFIG*,5\*1000);

Due to above properties offsets will be committed every 5 seconds, but if u want to verify whether offsets have been committed properly or not, then check the lag on the topic

If u describe the topic it will show you about that consumer group like if we ask manager he will team progress as like how many stories we were assigned with and how many stories we have completed

Refer the lagging concept in above for full clarity

1. Loosing the offsets-broker setting

For kafka >2.0 if consumer hasn’t read any new data in 7 days offsets can be lost

This can be controlled by broker setting offset.retention.minutes

We should set this atleast to a month

1. **Consumer polling - Heartbeat**

Each consumer in a consumer group must send the heart beat to the broker for each and every 3 seconds

Heartbeat.interval.ms=3 usually set to 1/3rd of session.timeout.ms

If no heart beats are send the consumer is considered as dead

This setting is to detect if consumer is being down

**Max.poll.interval.ms=5\_\*60\_0000 (5 minutes) for each and every 5 minutes poll should happen**

**Means** there can be maximum of 5 minutes gap between 2 consumer polls , means consumer should poll for every 5 minutes

Generally consumer should poll for every and every 2-5 seconds, else how do they know how many records are there in the broker.., but u can configure 5 minutes,if that guy is not polling even once for 5 minutes its dead

If I don’t poll, for 5 mins , consumer is considered as dead, means in this case will the consumer be removed from the consumer group????

1. Max.poll.records (default 500)

Means in a single poll maximum only 500 records at a time, let’s say I am doing bulk insertion, if I can insert only 500 records at a time, then in a single poll take only 500 records ,if u take 1000 records if u cant insert all at a time no use right

1. session.timeout.ms

It is the maximum amount of time that the consumer can stay connected without sending a heartbeat to the broker

Consumer must send an heartbeat within that time, else that consumer will be considered as died and broker will initiate a rebalance

Default is 45 seconds (45,000 ms)

The timeout used to detect client failures when using Kafka’s group management facility. The client sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this client from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by group.min.session.timeout.ms and group.max.session.timeout.ms.

1. Fetch.min.bytes=1,fetch.max.wait.ms=500

Both should be used in conjunction, same like producer batching ,linger.ms+batch.size

Like buses from kavali-Nellore -can max wait upto 30min,even if single customer came within 30mins also bus will start after 30 mins

30 mins is the max waiting time,if by 20 mins itself bus got filled, then also driver will start in 20 mins, ehich ever is earlier

If bus fills he will start else max wait time is 30 mins, lly here also min is 1MB, if broker doesn’t have data he cant wait for hours

To accumulate for that 1MB, he can wait upto “fetch.max.wait.ms” only

“Fetch.min.bytes” is the minimum amount of data server should return as a fetch request or poll request(FYI between every poll there is a

Break/interval of ex:-500ms wait time,in which the consumer wont poll at all)

Official definition

============

**The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than 1 will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.**

U can/may poll once for every 2 minutes, but this is to configure in single poll minimum how much data should be returned by the broker byte or 1MB,

but lets say the size of each message is very low lesser than a KB, but our data flow is only 10 messages per hour, means per hour max data can be 10KB,, if u set minimum as 1MB when u will get ur message u might get ur message after 10 hours if and only if 100 messages are received so

along with setting minimum bytes as 1 MB u should set max wait milliseconds also

so set as below

**fetch.min,bytes=1024 –** we should get min 1MB from the broker

fet.max.wait.ms=5000 means this is the maximum amount of time that kafka broker can block to respond for your fetch request, even If u set to 1MB/1024 KB as minimum bytes, broker cant keep u waiting for so many hours,it should give u the records within this max.wait.ms

If there is no sufficient data to immediately satisfy the requirement of **fetch.min.bytes broker** can max wait until **fetch.max.wait.ms=5\_000**

for 5 seconds after that even if broker have 1 KB also it should respond to the consumer,broker can keep u waiting

1. fetch.max.wait.ms

The maximum amount of time the server will block before answering the fetch request if there isn’t sufficient data to immediately satisfy the requirement given by fetch.min.bytes.

Ex:- gururaj asking profiles from RMG,if they send 1 per day its waste,if we cfg 10 min , if it took 1 month then it is time waste, gururaj can wait upto maximum a week,in that week even if 1 profile came also we will process

Ex:- Its like consume minimum100- max 500 records at once for 1 minute wait time

This is introducing a potential delay withinto be more efficient in consume requests

|  |  |
| --- | --- |
| fetch.min.bytes=1024  fetch.max.bytes=55MB | Max.poll.records=500 |
| Fetch.max.wait.ms=2000 (consumer can wait max of 2 seconds for each and every poll) |  |
| **max.poll.interval.ms=5\_60\*1000** |  |
|  |  |
|  |  |
|  |  |

1. max.poll.interval.ms

interval means- same like school intervals where we have gap between 2 classes,

this is the time gap between 2 polls,

this is the maximum delay between 2 polls where consumer will be idle before doing consumer.poll()

ex:- if u give “max.poll.interval.ms=5\*1000”means consumer can sit idle max upto 5 seconds without polling,

if he didn’t poll in this 5 seconds, that’s it consumer is considered as dead,

default is 5 minutes- means maximum for 5 minutes consumer can sit idle and enjoy without polling

same like us-if we don’t send any absence intimation for 5 days, they can remove us from our company

if poll() is not called before expiration of this timeout then the consumer is considered as failed and the group will be rebalanced in order to reassign the partitions to another member. For consumers using a non-null group.instance.id which reach this timeout, partitions will not be immediately reassigned. Instead, the consumer will stop sending heartbeats and partitions will be reassigned after expiration of session.timeout.ms. This mirrors the behavior of a static consumer which has shutdown.

1. Fetch.max.bytes=55MB

Maximum data that can be returned for each and every fetch or poll request, if u have more memory try increasing it so that u can process more records each and every single time

max.partition.fetch.bytes (1MB)

it is the max amount of data per partition server will return if u have 100 partitions,u need more RAM

1. Unique value for each message

what's a generic unique id that I can use for messages I receive from a consumer?

Topic+partition+offset –this could be the unique id for each and every message+timestamp

Because after some days data can be deleted in the topic based on the retention policy

Say If that old data in partition -1 got deleted, if new data also comes to the same

partition-1/folder-1, offset 20 then it will be a duplicate that’s why to avoid that scenario I am saying to store along with the new timestamp