## **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

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

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

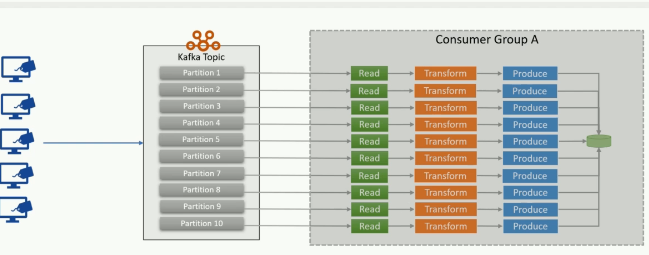
If Every consumer have same group id all consumers will join same group

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,



## What is consumer group 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, old consumer would have been reading from offset number -22 ,1st segment, 10th partition 10, then that died consumer 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 ,

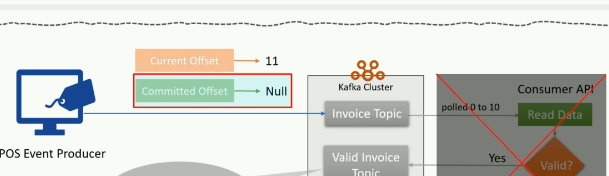
Like while we are working in a team , when we are leaving the company we should commit all our code then only other can take and continue from there right

### Enable auto commit offset

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

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 partition 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
2. This committed offset is used to avoid duplicate reading
3. 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
4. fetch.max.wait.ms It is the maximum amount of time the consumer has to wait for the broker to send back records.
5. session.timeout.ms,
6. partition.assignment.strategy: It is the strategy used by PartitionAssignor to split the partition among consumers in a 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

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.