Each kafka server is a Broker and each Broker is given a number.  
Each Broker may store multiple Topics with multiple partitions of the topic. All or few based on the number of Brokers available.  
Broker: 0 means Broker number 0

**Setup**Create empty folder inside kafka directory  
C:/kafka\_2.13-2.5.0/data/zookeeper  
C:/kafka\_2.13-2.5.0/data/kafka

**Starting zookeeper**  
Edit config\zookeeper.properties. All forward slashes  
set dataDir=C:/kafka\_2.13-2.5.0/data/zookeeper  
From kafka folder start zookeeper  
C:\kafka\_2.13-2.5.0>  
zookeeper-server-start config\zookeeper.properties  
It will bind to port 2181

**Starting kafka**  
Edit config/server.properties. All forward slashes  
set log.dirs=C:/kafka\_2.13-2.5.0/data/kafka  
From kafka folder start kafka but from a new command prompt as the earlier one has zookeeper running.  
C:\kafka\_2.13-2.5.0>  
kafka-server-start config\server.properties  
We will now find files created under these two  
C:/kafka\_2.13-2.5.0/data/zookeeper  
C:/kafka\_2.13-2.5.0/data/kafka

**Create topic**kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --create --partitions 3 --replication-factor 1  
Now see first\_topic folders created under C:\kafka\_2.13-2.5.0\data\kafka  
**List topics**  
kafka-topics --zookeeper 127.0.0.1:2181 --list  
**Detailed info about a topic**  
kafka-topics --zookeeper 127.0.0.1:2181 --topic first\_topic --describe  
**Creating and deleting topic**kafka-topics --zookeeper 127.0.0.1:2181 --topic second\_topic --create --partitions 6 --replication-factor 1  
kafka-topics --zookeeper 127.0.0.1:2181 --list  
kafka-topics --zookeeper 127.0.0.1:2181 --topic second\_topic --delete  
The topic will be marked for deletion  
kafka-topics --zookeeper 127.0.0.1:2181 --list  
**Producing data**c:\kafka\_2.13-2.5.0>kafka-console-producer --broker-list 127.0.0.1:9092 --topic first\_topic

>hello Tez

>learning Kafka

>lets see

>Terminate batch job (Y/N)? y  
**Producing data with acks=all Acknowledgement after replication**c:\kafka\_2.13-2.5.0>kafka-console-producer --broker-list 127.0.0.1:9092 --topic first\_topic --producer-property acks=all  
**Producing to a topic that does not exist**kafka-console-producer --broker-list 127.0.0.1:9092 --topic new\_topic  
You get a warning and the topic will be created and a leader would be elected. If tried to produce again, we are good. It is recommended to created topic first and not produce to non-existing topic directly. The default number of partitions in this case is 1. This is of no use. So change server.properties num.partitions= 1 property to **num.partitions=3  
Consuming from topic**kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first\_topic  
kafka-console-consumer by default will only read messages produced after it is loaded. This CLI utility would not read messaged produced earlier to its launch. So, start producing now to see consumer working.

**Consuming from topic from the beginning**

kafka-console-consumer --bootstrap-server localhost:9092 --topic first\_topic --from-beginning  
Now all the messages added earlier to consumer launch into the topic will also be consumed.  
**Multiple Consumers as a group**Run two consumers - following in TWO terminals  
And one producer  
kafka-console-consumer --bootstrap-server localhost:9092 --topic first\_topic **--group my-first-application**

We can see the messages produced being consumed by two consumers in a random fashion based on which partition of the topic the messages lands in. The consumers in the group make sure they understand who is reading from which partition.  
When using consumer group, last message read is remembered by storing offset. So when connected later, it will read the messages added later into the topic even before consumer with same consumer group is relaunched.

**Listing consumer groups**kafka-consumer-groups --bootstrap-server localhost:9092 --list  
**Describing a consumer group to see offset details**kafka-consumer-groups --bootstrap-server localhost:9092 --describe --group my-first-application  
Details about where the offset is in **each partition,** if all is read by the consumer, consumer info. etc. is detailed here.   
CURRENT-OFFSET - Offset till where the messages are read in the partition.  
LOG-END-OFFSET - Offset indicating the last messages published.  
If CURRENT-OFFSET is less than LOG-END-OFFSET, it means that the consumer-group my-first-application is yet to consume some more messages. This will be evident in LAG.  
CONSUMER-ID and HOST tell which consumer from the consumer-group and from which machine is consuming the messages from the partition.  
**Resetting offsets**consumer group can read again data from certain period again. Here are possible options about duration possible --to-datetime, --by-period**, --to-earliest**, --to-latest, **--shift-by**, --from-file, --to-current  
kafka-consumer-groups --bootstrap-server localhost:9092 --group my-first-application --reset-offsets --execute --topic first\_topic **--to-earliest**  
  
**Shift offset back and forth by number of messages positive to move forward and negative to move backward.**  
kafka-consumer-groups --bootstrap-server localhost:9092 --group my-first-application --reset-offsets --execute --topic first\_topic **--shift-by** 2  
This will lag by 2 messages now for **each partition in the topic**.  
**Download kafkatool.com to read messaged and see offsets. Just an alternate to CLI.  
  
replication.factor, Acks and min.insync.replicas (MIR)**

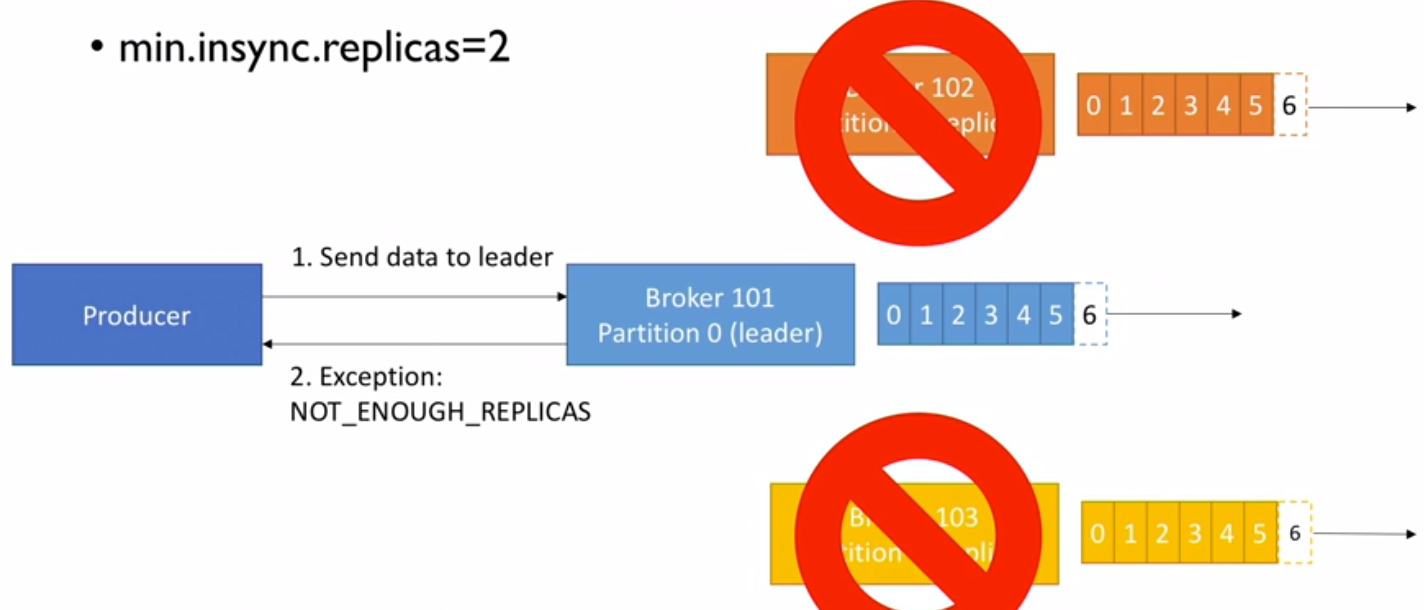
acks=0 – Just produce and forget. Higher chances of losing data. Usable for logging or sample collection data.

acks=1 – Produce and wait for acknowledgement from Leader Broker. Single point of failure if Leader goes down and if replicas do not receive the message by that time.

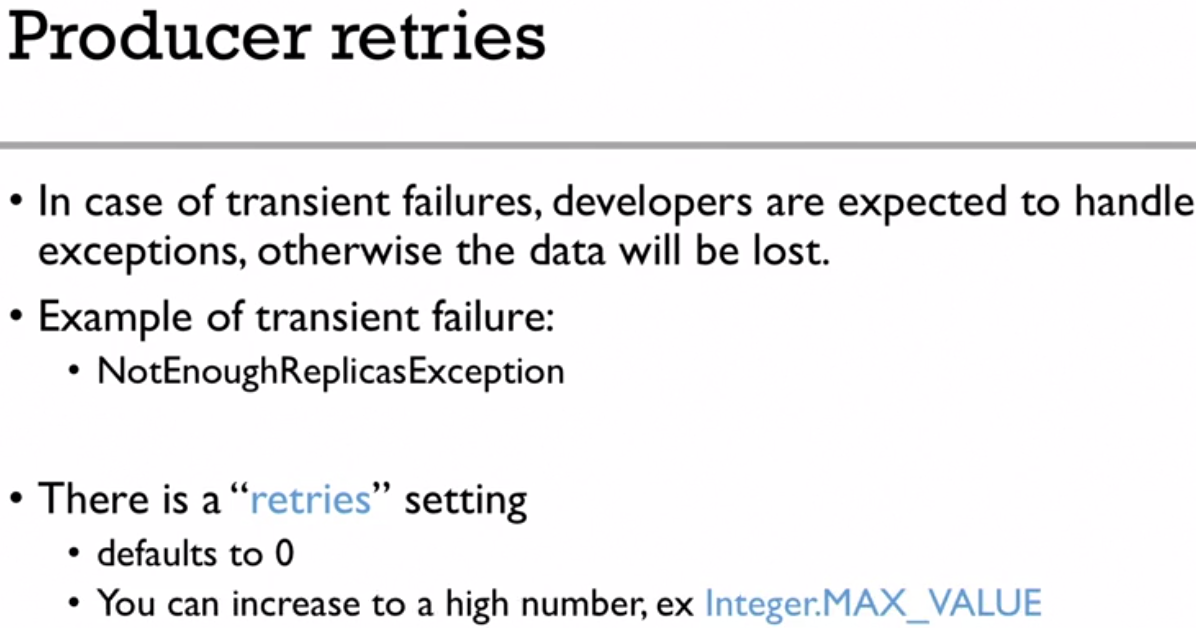
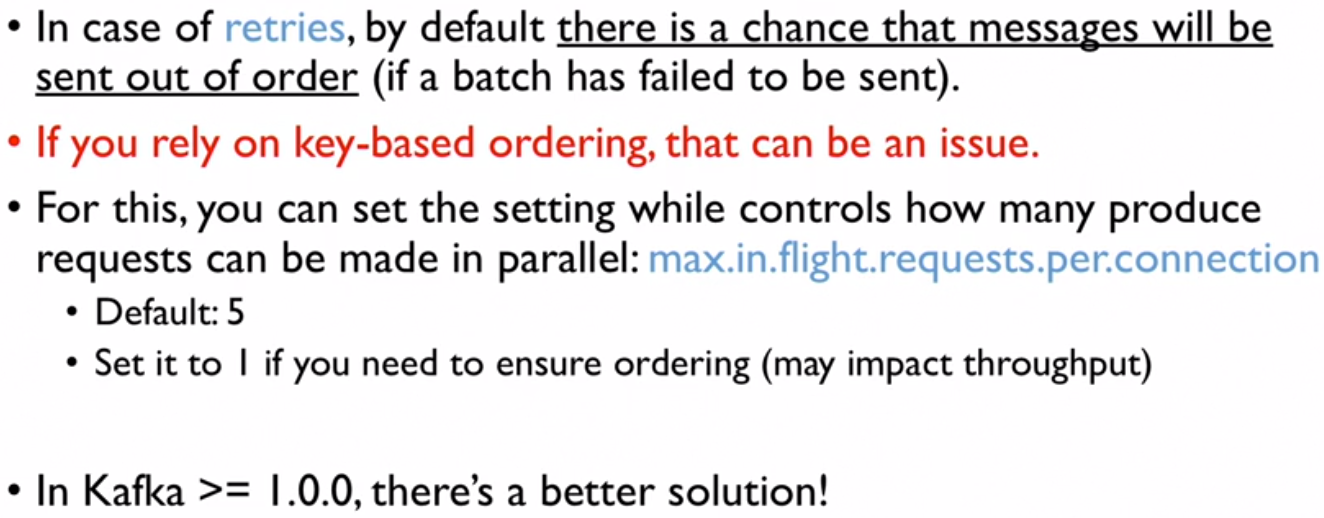
acks=all - Produce and wait for acknowledgement from Leader Broker after all replicas also receive a copy. A bit of latency added because the leader has to wait for all replica brokers to acknowledge before it acknowledges to Producer. But safety.

Too many replicas? Required to get replicated in all but do not have to wait until all acknowledge? Just a few replicas acknowledge is enough to move on?

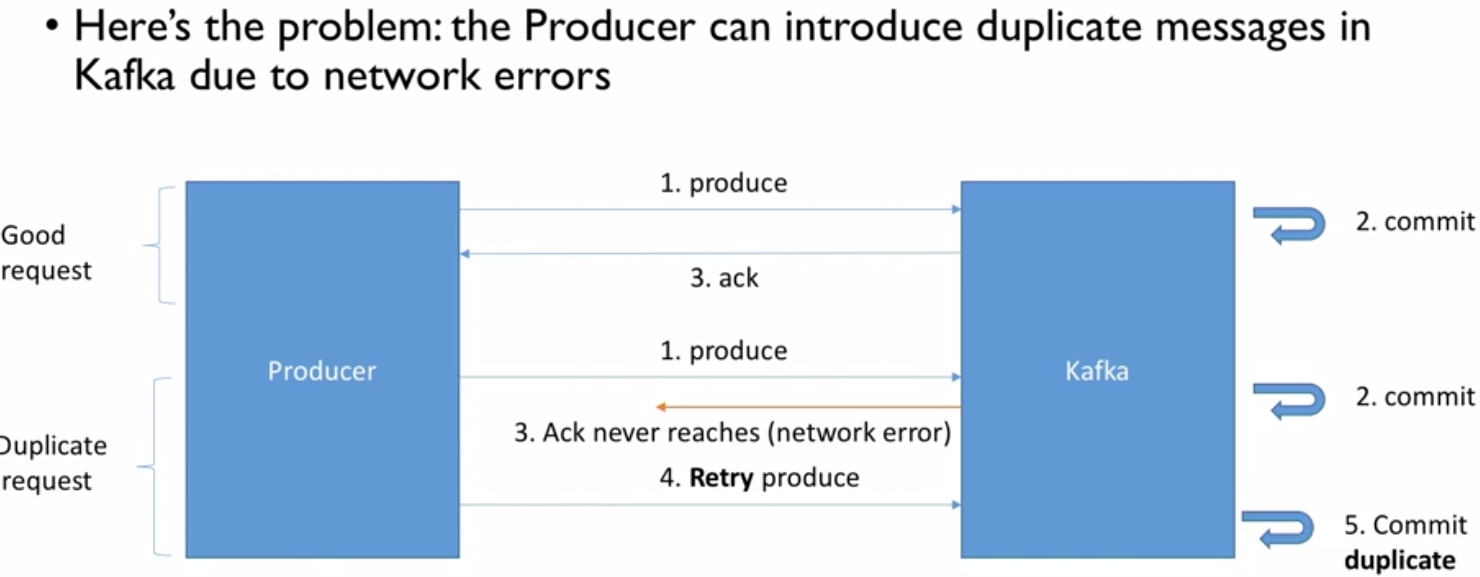
min.insync.replicas can be set at broker or topic level (override)  
min.insync.replicas (MIR) = 2 means that atleast 2 brokers including replicas and leader must respond/acknowledge that they have data.  

So having big number of MIR is also risky. It would keep throwing exceptions if that many number of replicas are not up. So a reasonable and practical number is to be chosen.

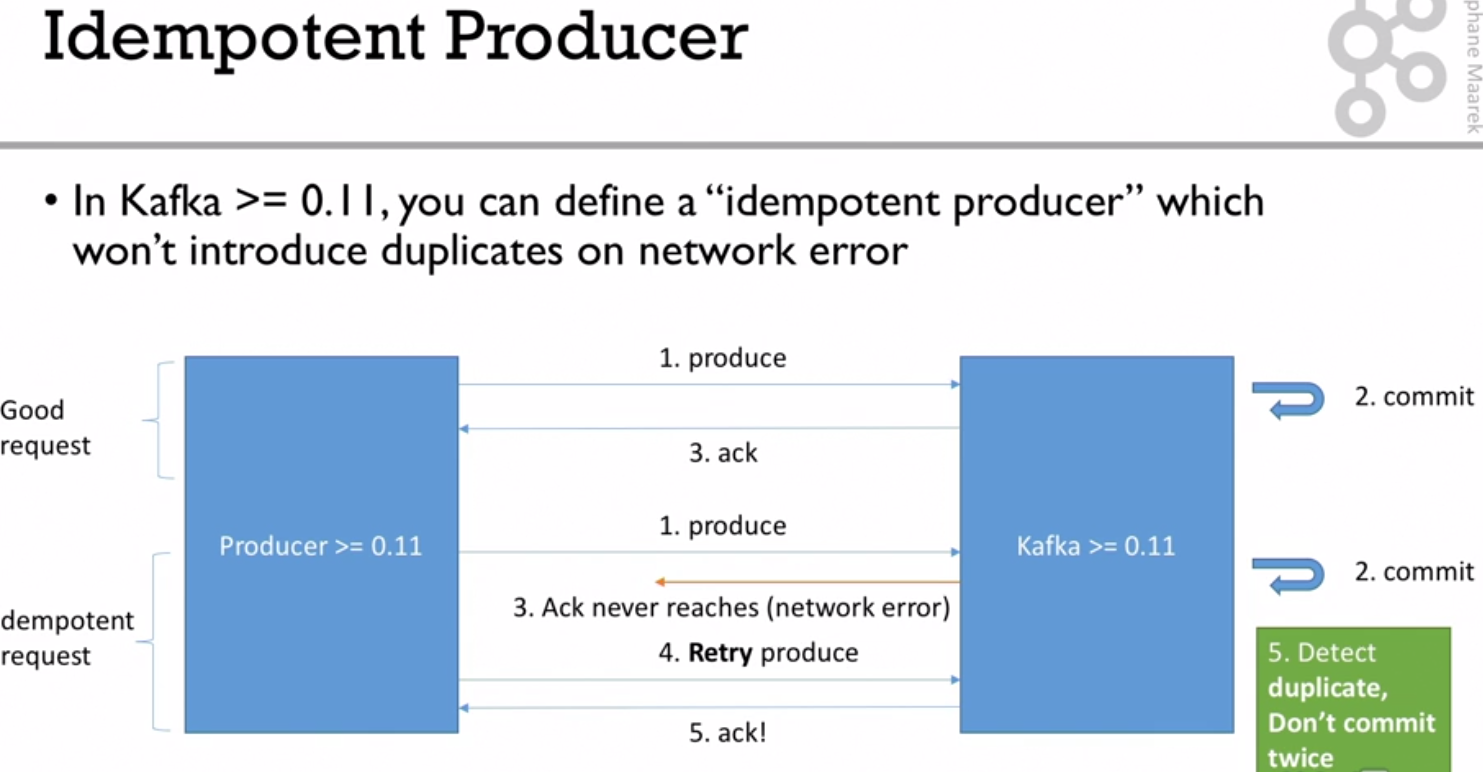
  


**max.in.flight.requests.per.cpnnection** set to 5 would ensure ordering even if various messages are retried in parallel. This is in later versions. Only in older versions, we have the problem of ordering.

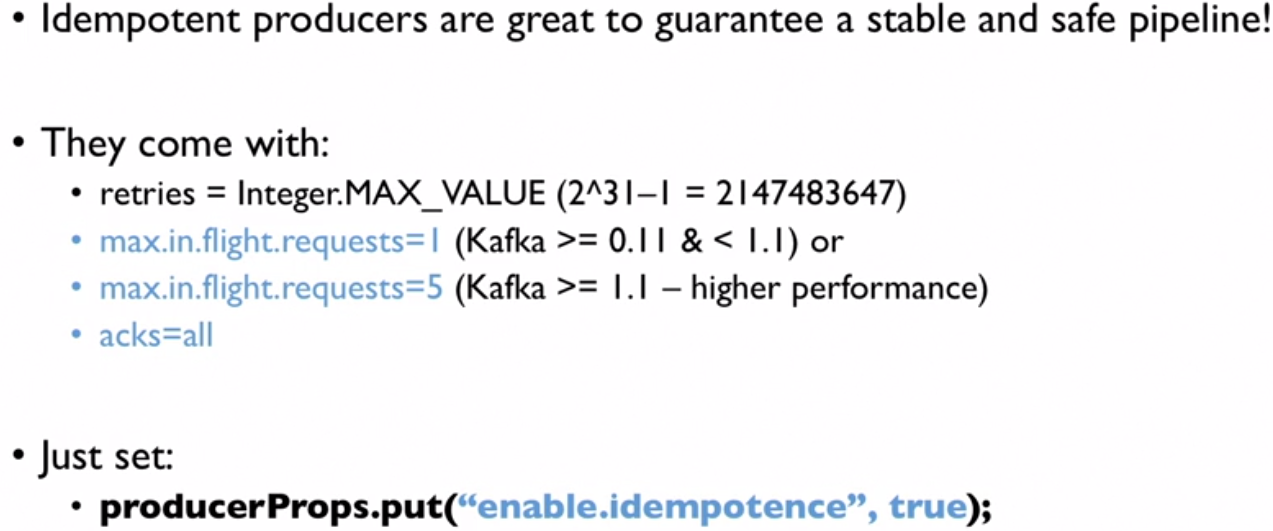


There was a problem when ack was being returned to Producer and Kafka was not aware of that 😊

Producer as it did not receive ack first time, will automatically try again sending the same message. Producer assumes that Kafka broker received its message only once because it received ack only once. But Kafka received it twice and committed twice.



When Producer produces messages, it also sends a produce request id along with the message. So when a retry is done, it send the same produce request id. This will help Kafka Broker realize that it has received that request id earlier already and it committed. So, this time, it would just understand the message as duplicate and simply returns ack without committing a new message again.



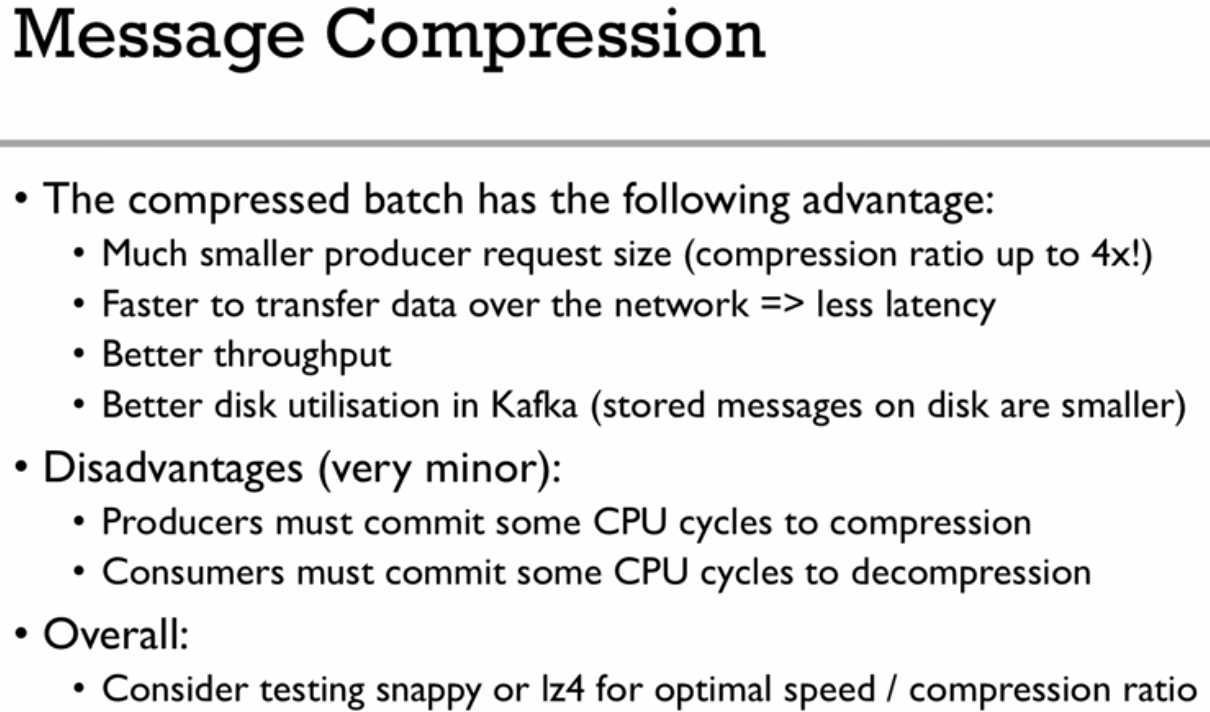
With Kafka >= 1.1, even parallel requests/messages on retries also would be sent in order unlike earlier versions.

A big number for retries would ensure that the sending is retried indefinitely. This can also create a lose of noisy traffic in case the minimum number of brokers are not available.

**max.in.flight.requests.per.cpnnection** set to 5 would ensure ordering even if various messages are retried in parallel. This is in later versions >=1.1. Only in older versions, we have the problem of ordering.

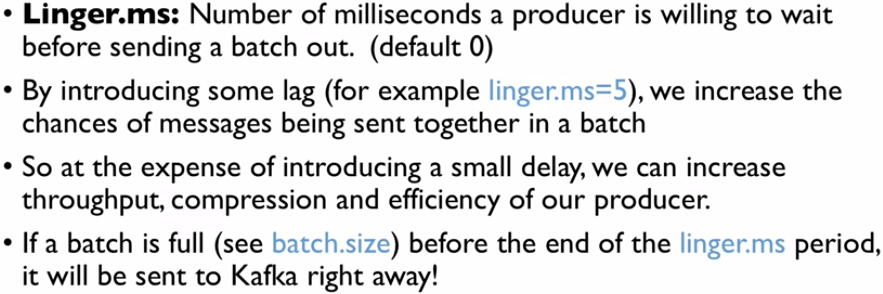


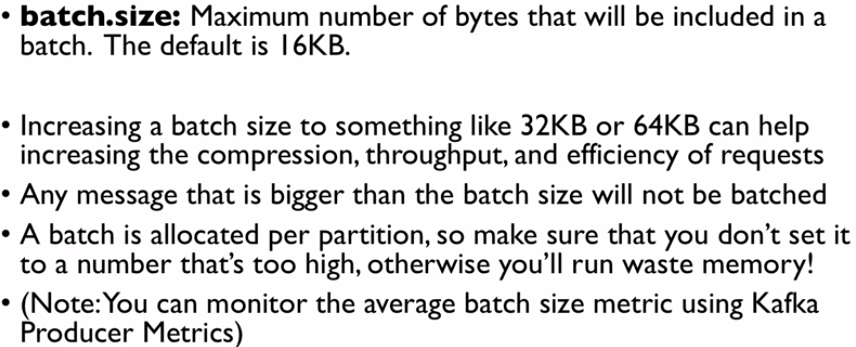
*//create a safe producer*properties.setProperty(ProducerConfig.***ENABLE\_IDEMPOTENCE\_CONFIG***, **"true"**);  
properties.setProperty(ProducerConfig.***ACKS\_CONFIG***, **"all"**);  
properties.setProperty(ProducerConfig.***RETRIES\_CONFIG***, Integer.*toString*(Integer.***MAX\_VALUE***));  
properties.setProperty(ProducerConfig.***MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION***, **"5"**);

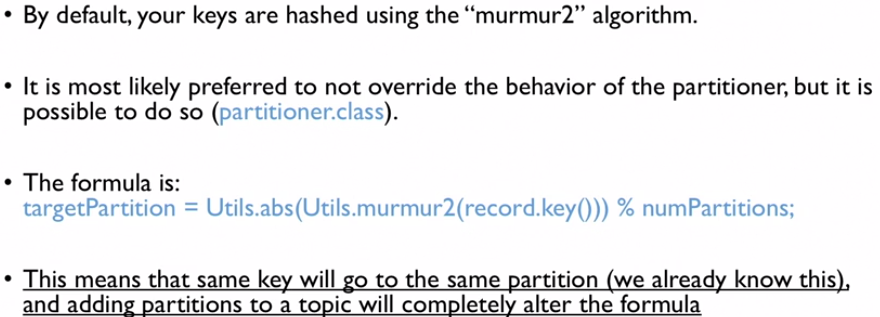
**Message Compression  
compression.type** none, gzip, lz4, snappy ****

**Gzip does very high compression but is not fast. Test all compression algorithms for your data and choose the best.**

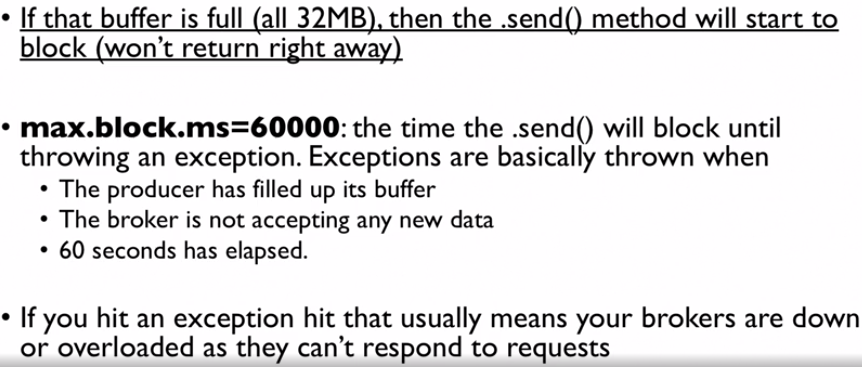
**linger.ms & batch.size**

****





**max.block.ms & buffer.memory**



**Setup elastic search on machine or use ready to use online Bonsai elastic search**

Register Bonsai.io

API documentation to understand elastic search <https://www.elastic.co/guide/en/elasticsearch/reference/current/docs.html>

**PUT /twitter** creates index ‘twitter’ to which documents can be posted

GET /\_cat/indices?v To confirm the the index is created

PUT /twitter/tweets/1 twitter index, tweets type, id 1  
{

"course":"Kafka for Beginners",

"student":"Tez K",

"module":"Elasticsearch"

}

GET /twitter/tweets/1

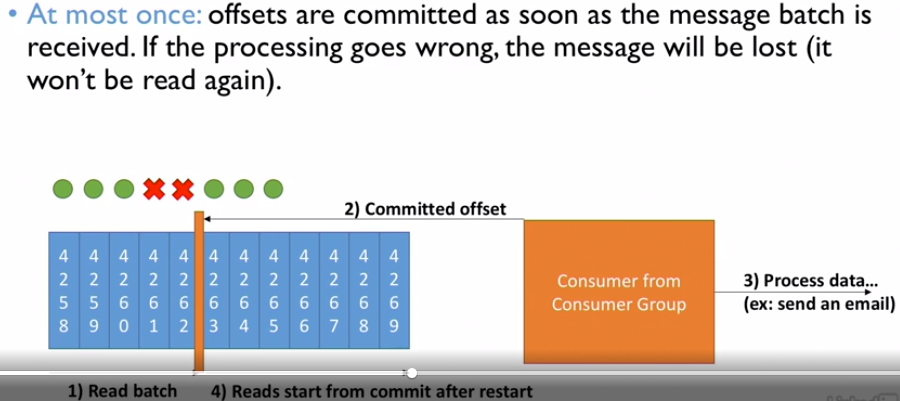
DELETE /twitter

<https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/_maven_repository.html>

dependencies {

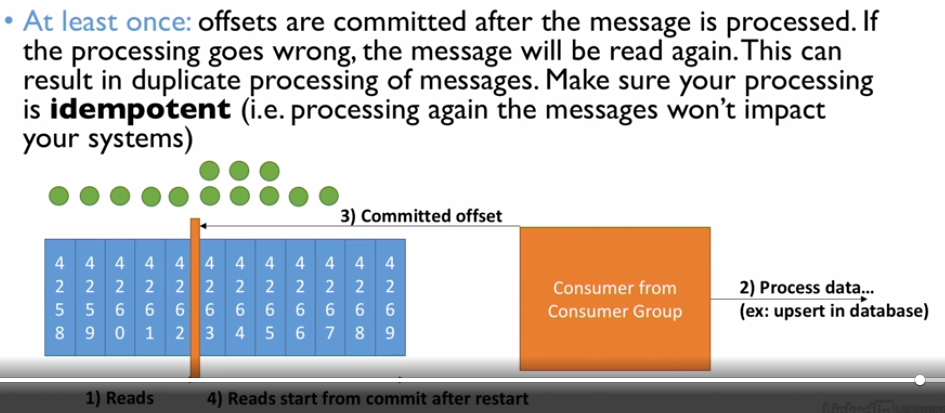
compile 'org.elasticsearch.client:elasticsearch-rest-client-sniffer:7.7.1'

}

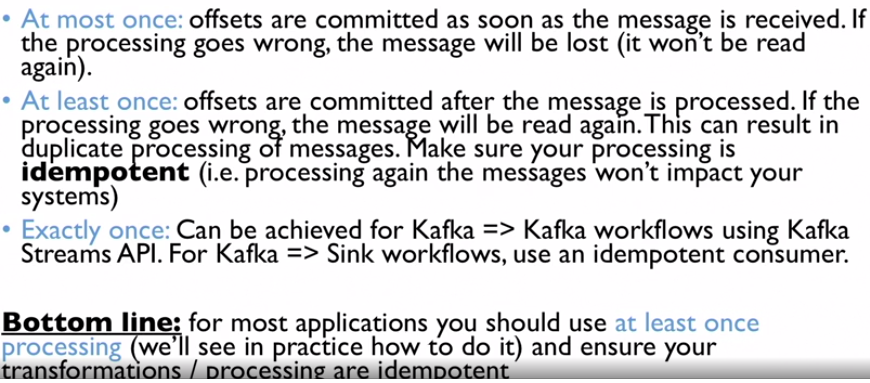
**Delivery Semantics**

Because it is **at most once**…. Offset is committed first before processing messages.

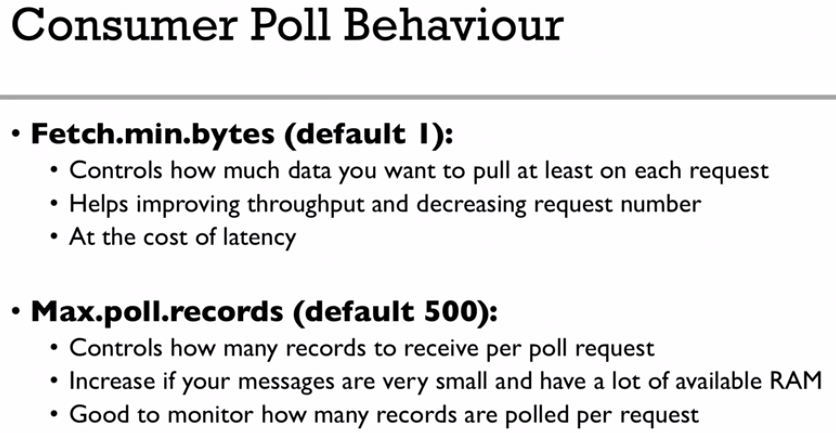
For a batch of 5 records, consumer commits 4262 as offset and starts processing the messages.  
After processing 4260, the consumer goes down. After coming up it sees offset 4262. So it starts from 4263. But 4261, 4262 are not processed ☹ This is fine for large data scenarios where a few messages missing is fine.

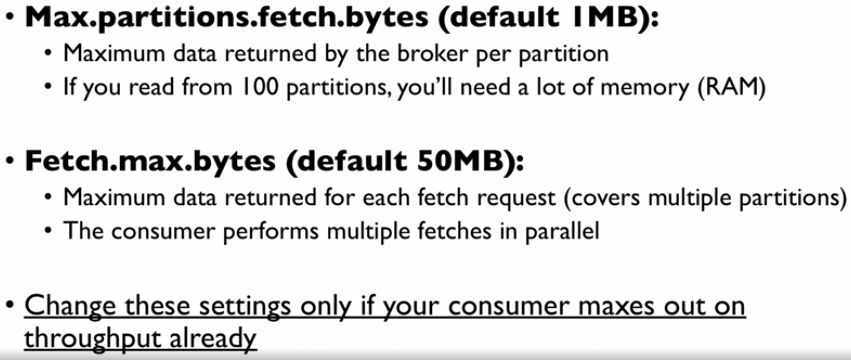


Because it is **at least once (Default)…** offset is committed only after processing messages. Consumer processes till 4262 and then commits offset. It processes 4263, 4264, 4265 and then consumer goes down. When consumer starts again, it finds offset as 4262 and hence start consuming from 4263 again. So, 4263, 4264, 4265 will be reprocessed again followed by 4266, 4267…. And the commits offset as 4267. When reprocessing, we have to make sure that it is idempotent and does not effect our system.



Consuming application can expect a unique id for the message to be sent and take appropriate action. In the program developed, a twitter message id extracted by kafka consumer and given to elasticsearch helped elasticsearch to determine POST versus PUT.

****

****

**Offset Commit Strategies**