# Kafka

Apache Kafka is a distributed software system that allows developers to process and store large amounts of real-time data

For this tutorial, Landoop kafka distribution and Windows version of the Kafka is used.

Landoop docker container installation steps exist in install\_and\_config folder.

Open landoop docker bash in CMD and mount local dev folder to docker -

*Directories are generally mounted for consuming the files present in host system by docker.*

*If the below command does not work then, docker copy command can be executed to move files from host system to docker*

|  |
| --- |
| *docker run --rm -it -v %cd%:/main --net=host landoop/fast-data-dev bash* |

Open docker bash in CMD without mounting any directory - use below command

|  |
| --- |
| *docker run --rm -it --net=host landoop/fast-data-dev bash* |

## Test Kafka executing below commands

### Create new topic

|  |
| --- |
| *kafka-topics --create --topic test-topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 2* |

### List kafka topic

|  |
| --- |
| *kafka-topics --bootstrap-server localhost:9092 --list* |

### Console kafka producer - Starts console producer

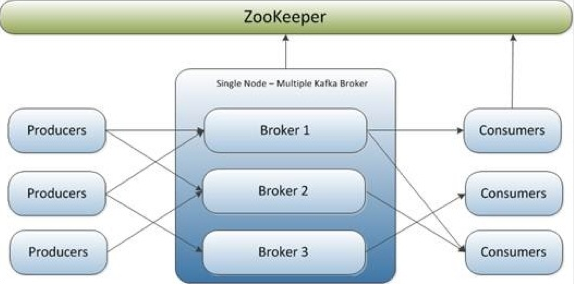
|  |
| --- |
| *kafka-console-producer --bootstrap-server localhost:9092 --topic test-topic* |

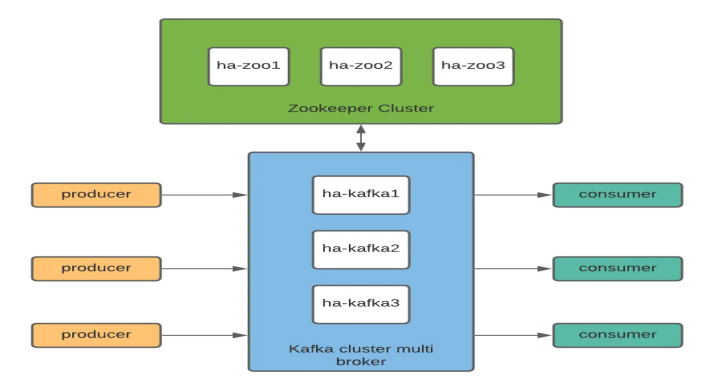
### Console kafka Consumer - Starts console consumer

|  |
| --- |
| *kafka-console-producer --bootstrap-server localhost:9092 --topic test-topic* *kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic --from-beginning* |

# Kafka Architecture

* In production systems, there will be multiple brokers and zookeepers.
* While configuring the topics we can specify only 1 broker server and configuration property ***replication-factor*** integer value will decide that how many times the topics will be replicated across multiple other brokers.
* If any Broker fails then other brokers can manage the messages to and from the topics
* Zookeeper will manage the health of all the brokers and manage the communication across them
* Multiple Zookeeper servers will be there in case primary zookeeper server fails then other redundant server will take its place





# Topics, partitions, and offsets in Kafka

* Kafka **Topic** is the queue where Producer pushes the messages and Consumer consumes it.
* Each Topic can be divided into multiple **partitions**. Partition Id for a kafka topic start from 0.
* Each partition can have an incremental message counter called **Offset**. Each partition Offset starts from 0 and it will incrementally increase. Offset denotes the unique position of a message within a partition.
* Offset guarantees the chronological order of the messages in a partition, but it does not guarantee the order across different partitions i.e. In a same partition, Message with Offset 0 came before the Message with Offset 1 however, Message with Offset 1 in Partition 1 may occur before Offset 0 of Partition 2.

A screen shot of a screen

Description automatically generated

* Messages stored in the topics as immutable objects.
* Messages in a topic are stored in partitions. Each message is composed of a key, a value, and additional metadata (such as headers and timestamps).
* The key-value pair is the main component of each message, where the key is optional and is often used to determine the partition to which the message is assigned.
* Based on the key, Kafka stores messages in specific partitions, ensuring that messages with the same key are consistently placed in the same partition. If the key is absent, Kafka distributes the messages across partitions using a round-robin or another default strategy.
* Consumers are not required to consume messages in chronological order, especially when consuming from multiple partitions, as each partition can be processed independently and concurrently.
* Consumers can consume the messages from a specific Partition and can also be configured with the Offset (offset indicates the specific position from which a consumer starts or continues consuming messages in a partition)
* While configuring Offset in the consumers, Partition needs to be configured else error will occur.
* While creating Topic, partitions is the property used to specify the number of partitions.
* While consuming data from a Topic by a Consumer, ***partition*** and ***offset*** is the property used to specify the partition number from where message needs to be read and offset number for the position

(from that position, messages will be consumed by the consumer) .

***Bash Scripts:***

|  |
| --- |
| * **Create new topic with 3 partitions**   *kafka-topics --create --topic test-topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 3*   * **Console kafka Consumer - Configure consumer to get the messages from a specific partition**   *kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic --partition 1*   * **Console kafka Consumer - Configure consumer to get the messages from a specific Offset**   *kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic --partition 1 --offset 1*   * **Console kafka Consumer - configure consumer to get the messages from a specific Offset without configuring partition. Below code will throw error, while configuring offset, partition needs to be configured**   *kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic --offset 1* |

# Kafka Cluster

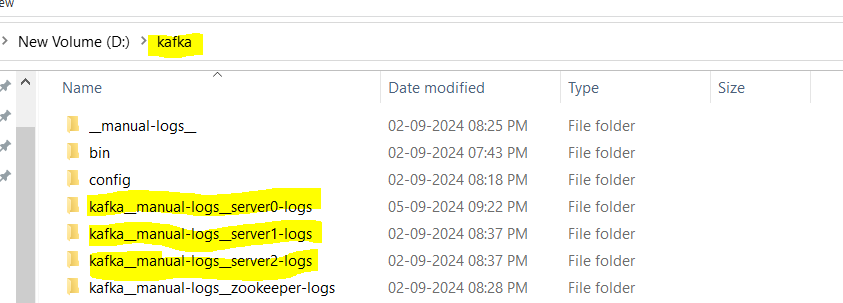
* In production environment multiple brokers are used for fault tolerance.
* For this exercise
  + Kafka is downloaded on Windows system and multiple copies of server.properties files will be used to spin up (mimic) cluster with multiple brokers.
  + Each copy of server.properties file will have different listener, different broker Id, different server log folder path and Zookeeper host will remain same in all the files, so that Zookeeper can track all the brokers.
  + In consumer.properties and producer.properties file, update bootstrap-server property to include all the broker server.
* Installation steps exist in install\_and\_config folder.

***Scripts:***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server0.properties*  *kafka-server-start D:\kafka\config\server1.properties*  *kafka-server-start D:\kafka\config\server2.properties*   * **Create a topic with 5 partitions**   *kafka-topics --create --topic test-cluster-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 1 --partitions 5*   * **Create a producer with key separator. Test Message - 1: Jerin, 2:ABCD etc...**   *kafka-console-producer --topic multi-broker-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --property key.parse=true --property key.seperator=: --property "parse.key=true" --property "key.separator=:"*   * **Create kafka console consumer to populate key and its values**   *kafka-console-consumer --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --topic multi-broker-topic --from-beginning --property print.key=true --property "key.separator=:"* |

* Whenever a topic is created with multiple partitions then we can see folders for each partition in log folder.

***Below Screenshot shows 3 log folders for 3 Kafka brokers***

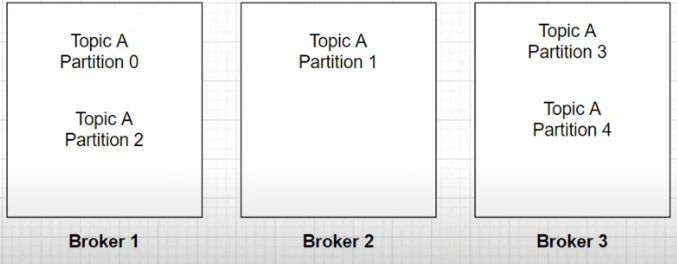


***After getting into 1 of the log folders, we can see multiple sub-folders for each partition (applicable in standalone as well as multi broker Kafka cluster).***

A screenshot of a computer

Description automatically generated

* In Kafka Cluster with multiple brokers (replication-factor = 1), whenever a topic is created with multiple partitions, those partitions will reside in different brokers so that if any broker crashes, then partitions in other brokers should still be receiving and sending messages.



Refer above screenshot, if Broker 1 crashes, then Partition 1, 3 and 4 will still be receiving and sending messages.

However, in standalone cluster i.e. cluster with 1 broker, if the broker crashes, the entire topic will be lost since all the partitions reside in that single broker.

# Topic with Replication in Multiple Broker Kafka Cluster

* Replication is a process of having multiple copies of the data for the sole purpose of availability in case one of the brokers goes down and is unavailable to server the request.

Topic’s Partition from one broker will be replicated to another broker i.e. In the below snippet, Topic 1 Partition 0 from Broker is replicated to Broker 2.

A diagram of a diagram

Description automatically generated

So, if 1 of the broker crashes then Partitions of that broker can be accessible from the other brokers where those partitions are replicated.

* In Kafka, replication happens at partition level or partition granularity.
* While creating Topic, replication-factor is the property used to specify the number of replicas for a partition.
* If replication factor is **r** and total partitions are **p** then there will be **p\*r** partitions (replica partitions) . For e.g., Let’s say replication factor is 2 and Partitions are 4 then there will be 2\*4=8 partitions, which includes ***replicated partitions + primary partitions***
* For all these replica partitions, there will be a Leader and In Sync replica(s). As the name suggests, **Leader** will be responsible for all the read and writes for that partition and **In Sync replica** will have data replicated from the primary partition.

e.g.

In above snippet, Topic 1 Partition 0 in Broker 1 is the Leader which will be responsible for all the read and writes.

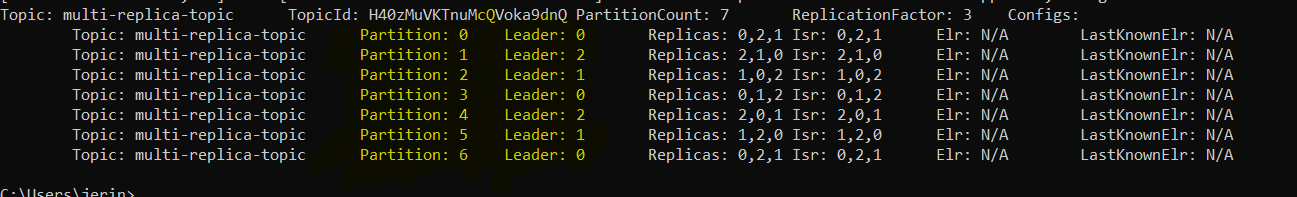
Topic 1 Partition 0 in Broker 2 is the In Sync replica which will have all the data replicated from the primary partition.

* **Leader and In-Sync Replicas (ISR)**:
  + In Kafka, each partition has one **Leader** replica, which is responsible for **handling all reads and writes** for that partition.
  + Other replicas, if any, are referred to as **In-Sync Replicas (ISR)**. These replicas keep a copy of the partition data and **replicate data from the Leader**.
  + The **Leader** ensures that all messages are replicated to the ISRs.
  + For a replica to be considered an **In-Sync Replica**, it must stay in sync with the Leader by consistently replicating the data.
  + **Key Points:** 
    - **Leader** handles all the **client requests** (both reads and writes) for a partition.
    - **In-Sync Replicas (ISRs)** are replicas that have the most up-to-date data from the Leader and are ready to take over leadership if the current Leader fails.
    - The **ISR set** may not include all replicas. Only those replicas that are synchronized with the Leader will be part of the ISR.

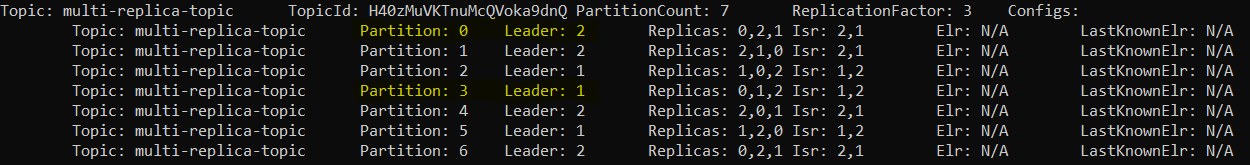
***Scripts:***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server0.properties*  *kafka-server-start D:\kafka\config\server1.properties*  *kafka-server-start D:\kafka\config\server2.properties*   * **Create a topic with 7 partitions and 3 as replication-factor**   *kafka-topics --create --topic multi-replica-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 3 --partitions 7*   * **Describe topic to check which partition is Leader**   *kafka-topics --topic multi-replica-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --describe*   * **Create Kafka console Producer – Add Messages to the Topic**   *kafka-console-producer --topic multi-replica-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --property key.parse=true --property "key.separator=:"*   * **Create Kafka console Consumer**   *kafka-console-consumer --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --topic multi-replica-topic --from-beginning --property print.key=true --property "key.separator=:"* |

* When above scripts are executed, a topic with 21 partitions will be created i.e. 3 replication-factor \* 7 partitions = 21 replica partitions.
* There will be 7 leaders and remaining will be In sync replicas.
* Total brokers in our exercise are 3 i.e. broker 0,1 and 2.
* To check, which replica partition is considered as Leader, use ***--describe*** property.

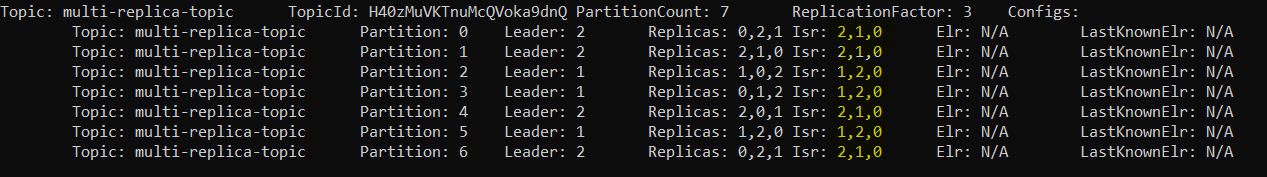


* + As per above snip, For Partition 0 Leader is Broker 0 i.e. for the given topic Partition present in Broker 0 is the Leder and other replica partitions will be In Sync replicas.
  + Replicas:0,2,1 in the above results indicate, in which broker replica partitions exist.
  + Isr denotes In Sync Replica. Isr:0,2,1 indicate, Partition 0 in broker 0,2,1 are in sync
* If one of the brokers goes down, then Kafka will promote one of the In Sync replica from other brokers as Leader.
  + For e.g. If Broker 0 crashes, then as can be seen in below screenshot:
    - Partition 0’s Leader is changed from broker 0 to Partition 0 replica present in broker 2
    - Partition 3’s Leader is changed from broker 0 to Partition 3 replica present in broker 1.



* + As can be seen in Isr section, Broker 0 is now removed from Isr.
* If the broker which earlier crashed comes back alive, then change in Leader depends on how many brokers are available.
  + Let’s say 2 out of 3 broker goes down, then Leader will be the Partitions present in the broker which is alive. Now once both the crashed brokers are up and running then Leader will be shuffled across all the available brokers. In this case, Kafka shuffle the Leader because Kafka tries to do load balancing
  + Let’s say only 1 out of 3 broker goes down, then Leader will be the Partitions present in the brokers which are alive. Now once the crashed brokers are up and running then Leader will not be shuffled.

For e.g., If Broker 0 is back online which was previously crashed, then as can be seen below there is no change in Leader.



* + As can be seen in Isr section, Broker 0 is now added back to Isr

# In-depth Intuition on Kafka Rack Awareness

* Rack Awareness feature in Kafka spreads replicas of the same partition across different racks to minimize data loss in the event of rack failure.

A screenshot of a computer

Description automatically generated

***\*DN: Data Node***

* Intuition of Rack Awareness –
  + If let’s say, 3 Kafka brokers exist in the same Rack and due to some issue like, network connectivity, natural calamities whole Rack is unavailable then topics in all the 3 brokers will reject any read and write request from the client/ user.
* Kafka internally manages the replication of partitions in separate broker and separate rack, once the rack details are provided in the server properties.
* Rack Id should be mentioned in the ***server.properties file, broker.rack = <integer>***
* Test rack awareness in Kafka by creating a topic with 1 partition and 2 replication.

***Scripts: Start Kafka***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server0.properties*  *kafka-server-start D:\kafka\config\server1.properties*  *kafka-server-start D:\kafka\config\server2.properties* |

* + Create a topic without rack details in server.properties file –

***assumption****: brokers will be part of same rack*

***Scripts: Create topic without adding broker.rack property in server.properties files***

|  |
| --- |
| * **Create a topic with 1 partitions and 2 as replication-factor**   *kafka-topics --create --topic test-topic-no-rack --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 2 --partitions 1*   * **Describe topic to check which partition is Leader**   *kafka-topics --topic test-topic-no-rack --bootstrap-server localhost:9092,localhost:9093,localhost:9094 –describe* |

After executing the describe script, as can be seen that 1 partition is created with Leader as broker 0 and In-Sync replica in broker 2. Both of these brokers exist in the same rack.



Now if the entire rack crashes, then topic won’t be accessible for any read or write.

* + Create topic with rack details in server.properties file

Update server.properties file by adding broker.rack property in all the broker’s server.properties files. Following are the rack details.

* + - Broker 0 : broker.rack = 0
    - Broker 1 : broker.rack = 1
    - Broker 2 : broker.rack = 0

A screenshot of a computer

Description automatically generated

***Scripts: Create topic after adding broker.rack property in server.properties files***

|  |
| --- |
| * **Create a topic with 1 partitions and 2 as replication-factor**   *kafka-topics --create --topic rack-test-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 2 --partitions 1*   * **Describe topic to check which partition is Leader**   *kafka-topics --topic rack-test-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 –describe* |

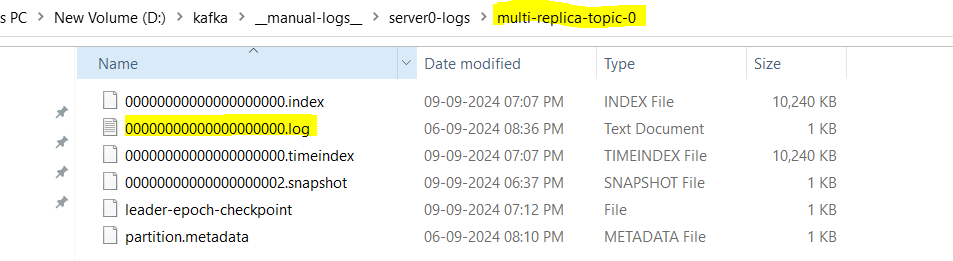
After executing the describe script, as can be seen that 1 partition is created with **Leader** as **broker 2** which is in **rack 1** and **In-Sync replica** is in **broker 1** which is in **rack 0**.



Now if the entire rack 1 crashes, then topic will still be accessible for read or write through partition present in broker 1 of rack 0.

# Kafka Log Segments

* Kafka writes the messages in the partition of a topic and each partition has log file where messages reside.



In the above snippet, following details can be deduced:

* + Topic – multi-replica-topic
  + Partition Name – multi-replica-topic-0
  + Partition Number – 0
  + Log File – 00000000000000000000.log

Log folder structure is ***/<Server-Log-Folder>/<Topic-Partition-Folder>/<XXXXXXXXX.log>***

* Kafka writes the messages in these log file in append manner.
* Instead of having one big log file with all the messages, Kafka maintains multiple smaller log files.
* If a log file exceeds certain size, then Kafka create a new log file
* These smaller log files inside a particular partition is called segment
* Following snippet shows how Topics, Partitions and Log Segments are structured.

A diagram of a diagram

Description automatically generated

* log.segment.bytes property in server.properties file is used to define the maximum size of a log segment file, beyond this size new segment file will be created.

***Snippet from server.properties file***



* Log Segment in Action:

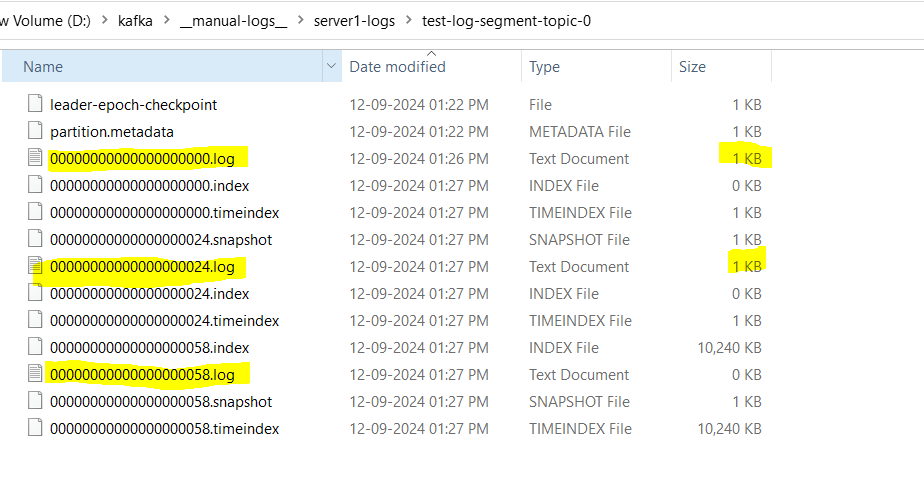
After setting log.segment.bytes property in server.properties files to 1000 bytes and then start the kafka server.

Multiple log segments will be created when publishing messages to the topic exceeds 1000 bytes

***Scripts***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server0.properties*  *kafka-server-start D:\kafka\config\server1.properties*  *kafka-server-start D:\kafka\config\server2.properties*   * **Create a topic with 1 partitions and 2 as replication-factor**   *kafka-topics --create --topic test-log-segment-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 2 --partitions 1*   * **Create kafka console Producer – Add Messages to the Topic**   *kafka-console-producer --topic test-log-segment-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --property key.parse=true --property "key.separator=:"* |

As can be seen below, after the messages exceeds 1000 bytes i.e. 1 KB a new log file is created.



# Kafka Producer Key & Message Acknowledgements

## Producer Keys

* In Kafka, Messages can be sent in Key Value pair through Producer
* Kafka uses Message Keys to decide the Topic Partition where the message will be stored
* Kafka uses Hash algorithm to convert the keys and map the Hash values to the partitions

For Example, Let’s say we have 2 partitions and If Hash algorithm is n%2 i.e. if any key is divisible by 2 then message associated to that key can be stored in Partition 0 else It can be stored in Partition 1

* If keys are null or not provided then, Kafka uses round robin method to store the messages across multiple partitions
* ***kafka-console-producer*** use following properties to send key value messages
  + ***parse.key*** – by default its false
  + ***key.seperator*** *–* delimiter for key and message
* ***kafka-console-consumer*** use following properties to read key and its value
  + ***print.key*** – by default its false
  + ***key.seperator*** *–* delimiter for key and message

***Scripts:***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server0.properties*  *kafka-server-start D:\kafka\config\server1.properties*  *kafka-server-start D:\kafka\config\server2.properties*   * **Create a topic**   *kafka-topics --create --topic test-cluster-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --replication-factor 1 --partitions 5*   * **Create a producer with key separator**   *kafka-console-producer --topic multi-broker-topic --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --property key.parse=true --property key.seperator=: --property "parse.key=true" --property "key.separator=:"*   * **Create kafka console consumer to populate key and its values**   *kafka-console-consumer --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --topic multi-broker-topic --from-beginning --property print.key=true --property "key.separator=:"* |

## Message Acknowledgements

* Message Acknowledgement is a feature to understand by the producer when a message is considered to be delivered or not
* ***acks*** parameter controls how many partition replicas must receive the message before the producer can consider the write successful
* This parameter has an impact on how likely messages are to be lost
* If acks = 0, it means that-
  + The producer will not wait for a reply from the broker before assuming message was sent successfully
  + If the broker goes offline or due to some exception, broker does not receive the message then producer will not know, and message will be lost.

A blue sign with white text

Description automatically generated

* If acks = 1, it means that-
  + Producer will receive a success message from the broker the moment leader receives the message
  + Producer won’t wait whether In Sync replicas received the message or not, the moment Leader partition receive the message, broker will send the acknowledgement to the Producer and then producer will send the next message
  + If the message is not written to the leader, then the producer will receive an error response and can try resending the message again
  + This helps to avoid potential loss of data

A blue box with white text

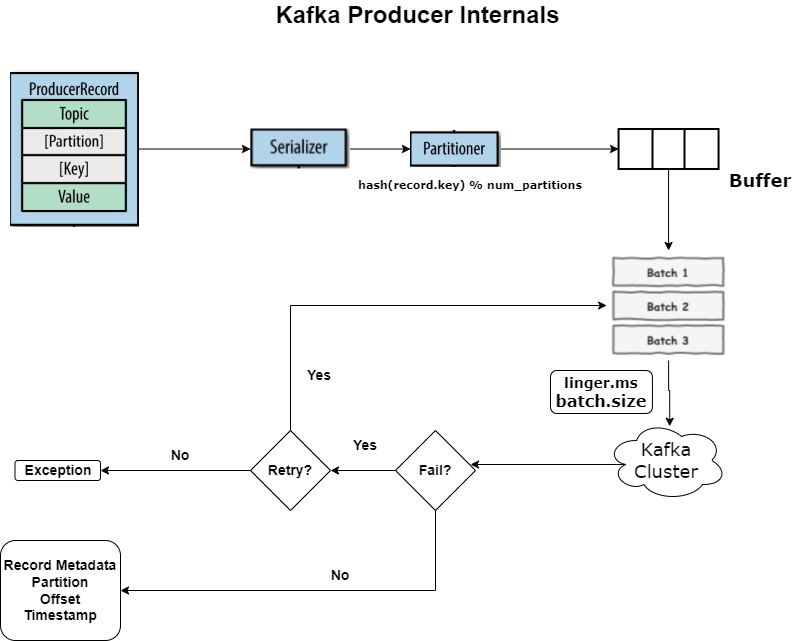
Description automatically generated

* If acks = all, it means that-
  + Producer will receive a success message from the broker only when all the In Sync replicas receive the message.
  + This is a slow process as Producer must wait for the success messages from all the In Sync Replica partitions

A diagram of a company

Description automatically generated

# Kafka Producer Internals



* **PartitionRecord**:
  + Each Kafka message is called a record. A record contains a key (optional) and a value, which represent the message's metadata and payload.
  + PartitionRecord contains –
    - Topic
    - Partition (optional)
    - Key (optional)
    - Value
* **Serializer**:
  + Before sending data over the network, the key and value are serialized into byte arrays. Kafka uses serializers like StringSerializer etc., depending on the data type of the key and value.
* **Partitioner**:
  + Kafka uses a partitioner to determine which partition a message should be written to. The partitioner applies a hashing algorithm (default: hash of the message key) to map a key to a specific partition, ensuring consistent message placement for keys across partitions.
* **Buffer**:
  + The producer maintains a buffer where messages are accumulated before being sent to Kafka. The buffer improves throughput and allows for message compression, improving network efficiency.
  + In-memory queue kind of data structure.
* **Batch**:
  + Messages are grouped into batches for efficient writing to Kafka.
  + There will be 3 batches created by Producer
  + ***linger.ms***: Defines how long the producer waits before sending a batch of messages to Kafka Broker, allowing time to accumulate more messages to improve throughput.
  + ***batch.size***: Defines the maximum size of a batch. Once the batch reaches this size, it is sent to Kafka regardless of **linger.ms**.
* **Retry**:
  + If a batch fails to be written to Kafka (due to network issues, partition unavailability, etc.), Kafka's retry mechanism attempts to resend the batch of messages. The retry count can be configured.
* **Exception**:
  + If all retries fail, the producer raises an exception, indicating that the message could not be delivered.
* **Metadata Assignment**:
  + Once a message is successfully stored in Kafka, metadata such as the partition, offset, and timestamp is assigned to the record. This helps consumers know where the message is stored and allows them to read the message in the correct order.

# Kafka Producer Properties

## Buffer and Batch

A screen shot of a computer

Description automatically generated

Whenever a message is sent, it will not be directly sent to Kafka cluster, It will be accumulated to a buffer in multiple batches then these batches will be sent to I/O thread and then to Kafka cluster.

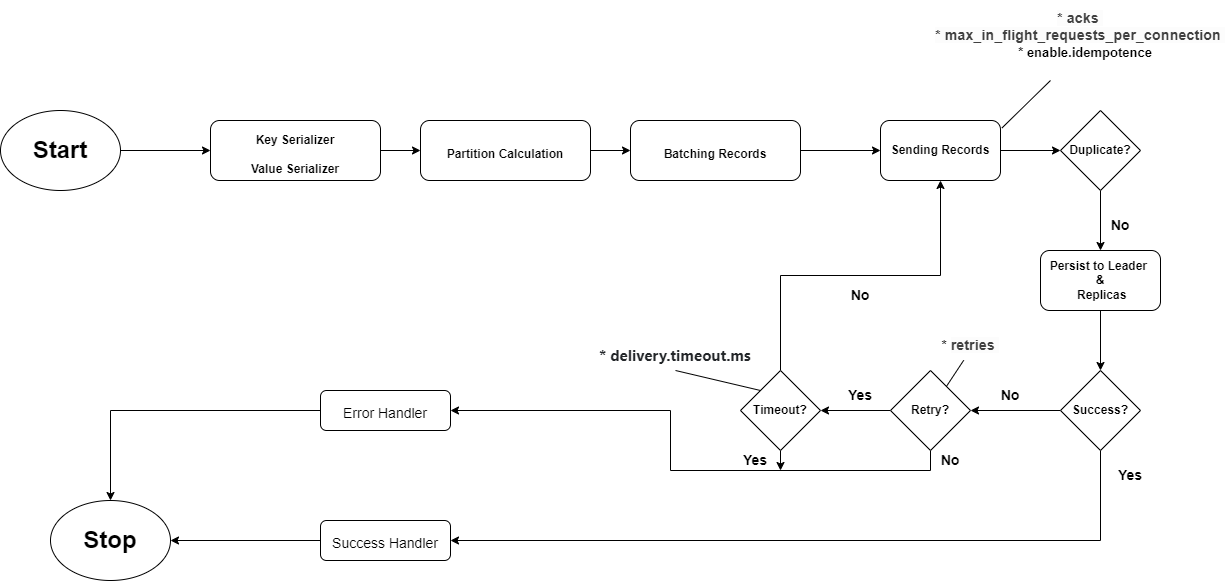
Following properties decide the operation on the Batch -

* ***batch.size***: Defines the maximum size of a batch. Once the batch reaches this size, it is sent to Kafka regardless of **linger.ms**.
* ***linger.ms***: Defines how long the producer waits before sending a batch of messages to Kafka Broker, allowing time to accumulate more messages to improve throughput.
* ***buffer.memory***: Defines memory size of the buffer, it is used to accumulate the messages for better I/O and compression. Default buffer.memory is 32 MB
* ***max.block.ms***:If I/O thread is busy and batches are full (can happen if producer sending messages faster than they can be transmitted to the broker) then new messages from the producer cannot be accumulated then sending new messages need to be blocked. ***max.block.ms*** defines how long messages are blocked from sending.
* If **linger.ms = 0**, then messages will be sent immediately to Kafka cluster, it will not be accumulated in buffer memory or in batch. However, if I/O thread is busy then message cannot be sent immediately, and it must wait.
* ***producer.flush()*** :This is the python Kafka producer API method which is used to flush the messages stored in buffer to Kafka cluster.

Let’s say, if this producer is not flushed and producer is closed then messages in the buffer will be lost and won’t be sent to Kafka cluster.

* ***producer.close()*** : This is the python Kafka producer method which will be used close the producer connection to Kafka cluster.

## Retry Mechanism



Kafka producer sends the message and wait for its acknowledgement (depends on ***ack*** property) whether message is written to the partition or not.

If Kafka producer does not receive successful acknowledgement, then it could be due to following reasons:

1. Messages are failed to be written to partition.
   * ***retry*** property is used to manage this scenario
2. Messages are successfully written but due to some issue successful acknowledgements are not transmitted back to producer.
   * ***enable.idempotence*** property ensure duplicate messages are not processed again.

**Another intricacy, Considering below scenario –**

If Kafka producer is sending 3 batches of messages to a same partition, let’s call these batches as B1, B2 and B3.

* B1 is successfully written to the partition
* B2 is failed and before retrying B3 is sent and written successfully to the partition
* After retrying, B2 is successfully written to the partition

Now the order of the messages is deteriorated, original order of the batches were B1, B2 and B3. After writing it to the partition, order has changed to B1, B3 and B3.

To maintain the order of the messages in the partition, property “***max\_in\_flight\_requests\_per\_connection***” can be used. This property states that how many unacknowledged messages can be sent at a time. If it is set to 1 then Kafka Producer will wait for the acknowledgement before sending next message. So in this scenario,

* B1 is successfully written to the partition and after the acknowledgement B2 will be sent.
* B2 is failed and now retry happens and let’s say after retrying, B2 is successfully written to the partition and acknowledgement is received.
* After receiving the acknowledgement for B2, B3 will be sent.

This helps to maintain the order of the messages.

Details can be seen below for the property - “***enable.idempotence***” and “**max\_in\_flight\_requests\_per\_connection**” -

* ***enable.idempotence:***
  + **Purpose**: Ensures that **exactly-once delivery** semantics are maintained by the Kafka producer. This setting ensures that even if retries happen, duplicate messages are not sent.
  + **Default**: false (can be enabled as true).

**When it's enabled (true):**

* + Kafka producer will avoid sending duplicate messages during retries. Even if a producer experiences failures, it will not produce the same message twice.
  + To achieve idempotency, the producer assigns a **sequence number** to each message per partition and ensures the broker deduplicates them, so only unique messages are written.
  + **acks=all**: This setting will automatically require acknowledgments from **all replicas** before a message is considered successfully sent.
  + **retries**: The number of retries will be automatically set to a high value (Integer.MAX\_VALUE) to ensure failed sends are retried indefinitely.
* ***max\_in\_flight\_requests\_per\_connection:***
  + **Purpose**: Controls the maximum number of unacknowledged messages (or requests) that can be sent **per connection** at any given time before receiving an acknowledgment from Kafka.
  + **Default**: 5.

**How it works:**

* + When a producer sends a message, Kafka may not immediately acknowledge it. If the producer is waiting for the acknowledgment before sending the next message, this could reduce throughput.
  + Setting max\_in\_flight\_requests\_per\_connection allows the producer to send multiple messages (up to the specified limit) without waiting for the previous message’s acknowledgment. This improves **throughput** by increasing the number of requests in flight (outstanding requests).
  + Setting this value to a higher number of increases throughput but could lead to **message reordering** if retries are enabled, because messages sent later may succeed while earlier ones fail and retry. If ordering of messages is critical, this setting must be carefully tuned.

**Recommended Setting for Idempotence:**

* + When enable.idempotence=true, Kafka requires **max\_in\_flight\_requests\_per\_connection=5** or **lower** to avoid duplicate or out-of-order message production.
  + To maintain **ordering guarantees**, if enable.idempotence is enabled, it is usually best to set max\_in\_flight\_requests\_per\_connection=1 or 5 to balance throughput with reliability.

# Primary methods of sending messages

There are 3 primary methods of sending messages:

1. **Fire and Forget**
   * In this method, the producer sends the message without waiting for an acknowledgment.
   * Even though Kafka is highly available, there might be scenarios of losing the message
   * In this method, logging the lost messages are not possible
   * This method is faster as it does not have to wait for the acknowledgement.
   * Python Code at:
     + */main/pyhton/*2-kafka-send-message-fire-forget.py
2. **Synchronous Send**
   * In this method, Producer will wait for an acknowledgment before sending the next message
   * When producer sends a message, Kafka python method ***send()*** will be used and it will return a ***future object***. If ***get()*** method is used on future object, it will wait to see if send() is successful or not.
   * Since this method needs to wait for the acknowledgement, makes it slow (reduce throughput)
   * Logging of the lost messages can be enabled
   * Python Code at:
     + */main/pyhton/*3-kafka-send-message-synchronous.py
3. **Asynchronous Send**
   * In this method, the message is sent asynchronously, and a callback is used to handle success or failure.
   * All the messages will be fired parallelly and then in the background, success or failure callback functions will be called to get the successful or failure messages
   * Using the call-back function, logging of the lost messages can be enabled
   * Python Code at:
     + */main/pyhton/*4-kafka-send-message-asynchronous.py
     + By default, Kafka python uses Asynchronous Send without callbacks.
     + For callbacks use, ***add\_callback*** method for success and ***add\_errback*** for error
     + These methods take function as input
     + The functions need standard parameter, i.e. 1st parameter should be the standard parameter and then user can define other parameters

# Kafka Topic Partitioning Strategy when Key is NULL

* By default, Kafka follows round robin method for message assignment to the partitions.
* Round Robin method is costly in nature
* To tackle this problem, Sticky partitioning can be used.
* Sticky Partitioning is a producer-side feature that ensures messages from a batch are sent to a specific partition, rather than randomly distributing them across available partitions.
* It is mainly used to optimize batching of messages and to reduce the overhead caused by frequent switching between partitions.

**How Sticky Partitioning Works:**

* When a Kafka producer sends messages to a topic, it needs to decide which partition the messages should be sent to.
* By default, Kafka producers distribute messages across partitions in a round-robin fashion if no key is provided, or it will hash the key to determine the partition.
* Sticky partitioning changes this behaviour by sending multiple consecutive messages to the same partition (even without a key) for a configurable amount of time (i.e. linger.ms property) or until a batch is full (i.e. batch.size property).
* After a batch of messages has been sent (or timed out), the producer will select a new partition and continue sending messages to that partition.

**Benefits of Sticky Partitioning:**

1. Batching Efficiency: Sticky partitioning allows the producer to batch multiple messages more efficiently since they are sent to the same partition. This reduces the number of smaller, inefficient batches.
2. Reduced Latency: By avoiding the frequent switching between partitions, sticky partitioning helps reduce the latency associated with switching and improves throughput.
3. Improved Resource Utilization: By sending larger batches of messages to a single partition, the producer uses network resources more efficiently, leading to better overall performance.

**Example Use Case:**

In a scenario where the producer is sending a high volume of messages without explicit keys, sticky partitioning ensures that the producer can group several messages together into a single batch, reducing the number of round-trip requests to the broker.

**Configuring Sticky Partitioning:**

Kafka's linger.ms (linger time) and batch.size (batch size) configurations can be used to control how long the producer sticks to a partition before switching:

* linger.ms: Time to wait before sending a batch, even if it is not full.
* batch.size: Maximum size of the batch to accumulate before sending it to the broker.

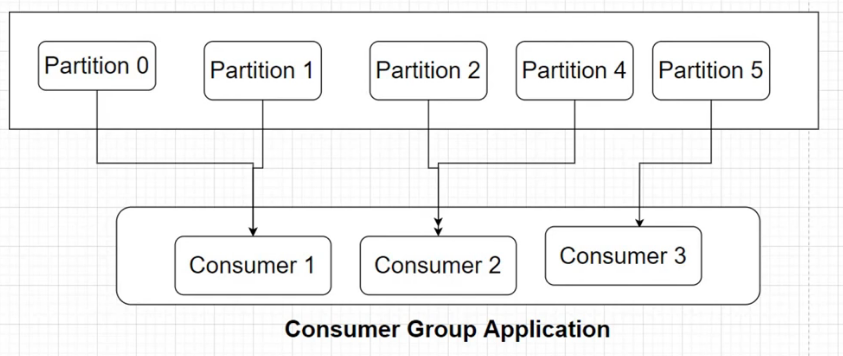
# Consumer & Consumer Group

* Consumer is an application to consume the messages from the topic.

A diagram of a diagram

Description automatically generated

* Kafka Consumer must request for the message from Topic, it’s a ***pull method*** and not the push method i.e. Broker won’t push the message automatically to Consumer.
* Data is always read from a low to high offset.
* When an app must consume messages from a topic, application needs to create a consumer object, subscribe to a topic and start receiving the messages, validate it and write the messages to data store
* If messages are published to a topic at a very high speed and 1 consumer can not process the messages at the same speed then there will be too much lag in message processing, to overcome this, ***consumer group*** is created with multiple consumers to scale up the consumption and these consumers will receive messages from a different subset of the partitions
* Consumer Group is a feature for Load Balancing while consuming the high speed messages



* If there is only 1 consumer in the consumer group, then messages from all the partitions will be consumed by that consumer

A diagram of a diagram

Description automatically generated

* If there are only 2 consumers in the consumer group, then messages from all the partitions will be consumed by those 2 consumers

A diagram of a diagram of a consumer

Description automatically generated

* If total consumers in the consumer group is equal to the total number of partitions, then messages from each partition will be consumed by each consumer

A diagram of a diagram

Description automatically generated

* If total consumers in the consumer group is greater than the total number of partitions, then messages from each partition will be consumed by each consumer and additional consumers in the group will remain idle. (There’s no point of adding more consumers than partitions in a topic)

A diagram of a diagram

Description automatically generated

* Kafka supports multiple consumer groups on the same topic.

A diagram of a diagram

Description automatically generated

***Scripts:***

|  |
| --- |
| * **Start Zookeeper using below code - change path of zookeeper.properties**   *zookeeper-server-start D:\kafka\config\zookeeper.properties*   * **Start kafka clsuter - multiple brokers**   *kafka-server-start D:\kafka\config\server.properties*   * **Create a topic with 3 partitions and 1 as replication-factor**   *kafka-topics --create --topic consumer-group-topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 3*   * ***<Create Python code to push messages to specific partition***   ***Refer Python Script: /main/python/ 5-kafka-send-message-to-specific-partition.py >***   * **Create 3 Consumers in the same Consumer group**   *kafka-console-consumer --bootstrap-server localhost:9092 --topic consumer-group-topic --from-beginning --group consumer-group-abc*  *kafka-console-consumer --bootstrap-server localhost:9092 --topic consumer-group-topic --from-beginning --group consumer-group-abc*  *kafka-console-consumer --bootstrap-server localhost:9092 --topic consumer-group-topic --from-beginning --group consumer-group-abc* |

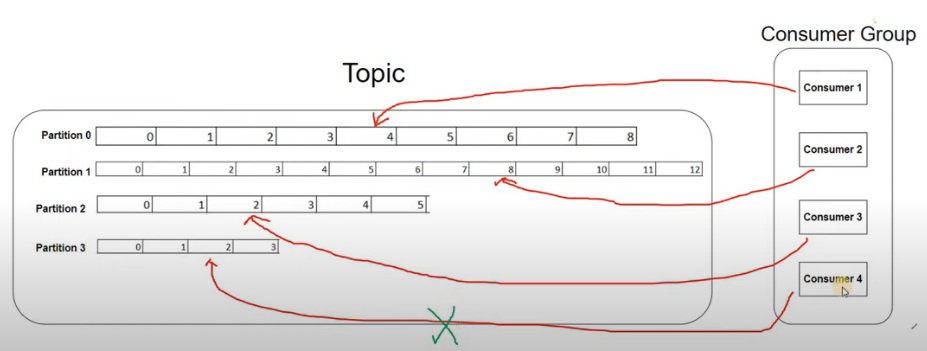
When ever a new consumer is added to the Consumer Group, Kafka will automatically rebalance i.e. let’s say, there are 4 partitions in a topic and 2 Consumers in a Consumer Group, so each consumer will consumer messages from 2 partitions each.

Now If 2 new consumers are added in the consumer group then Kafka will perform consumer rebalance and partitions from previous consumers will be allocated to the new consumers, so that each consumer can have 1 partition each.

# Consumer Offset Commit

Let’s consider a topic with 4 partitions and 4 consumers in the same consumer group, each partition will be mapped to a consumer from the same consumer group. If one of the consumers from consumer group fails, then another consumer from the same group will take its place.

In below snippet, consumer 4 fails which was connected to partition 4.



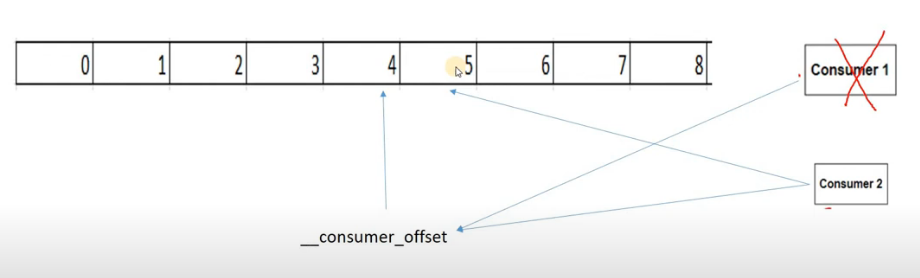
Now, Consumer 3 will start processing messages from partition 4.

A screenshot of a graph

Description automatically generated

Now question arises, from which offset, new consumer will start processing the message from the partition. To answer this question, ***offset commit*** comes into picture.

* *Offset Commit refers to the process where a Kafka consumer saves the last offset it has successfully processed.*
* *This is important for fault tolerance and consumer restart scenarios because Kafka uses these committed offsets to resume consuming messages from where it left off after a failure or restart*
* *After a consumer successfully processes a message, it can* ***commit the offset*** *to Kafka topic.*
* *Kafka stores these offsets in a special internal topic* ***(\_\_consumer\_offsets)****.*
* *If the consumer crashes or restarts, Kafka uses the last committed offset to know where to resume reading.*
* ***“enable\_auto\_commit = True”*** *property in Kafka Python API will enable offset commit to topic* ***“\_\_consumer\_offsets”***
* ***“auto\_commit\_interval\_ms”*** *property in Kafka Python API will allow to set time in milliseconds to send offset information to the topic* ***“\_\_consumer\_offsets”***
* Python Code at:
  + */main/pyhton/*7-kafka-consumer-offset-commit.py



In the above snippet, Consumer 1 processed messages till offset 4 and that offset information is sent to Kafka topic “***\_\_consumer\_offsets***”. After consumer 1 fails, Consumer 2 will communicate with “***\_\_consumer\_offsets***” topic to identify the offset information and then will pick up the messages beyond that offset.

***Script:***

|  |
| --- |
| * **Check messages in topic** “***\_\_consumer\_offsets***”   *kafka-console-consumer --bootstrap-server localhost:9092 --topic \_\_consumer\_offsets* |

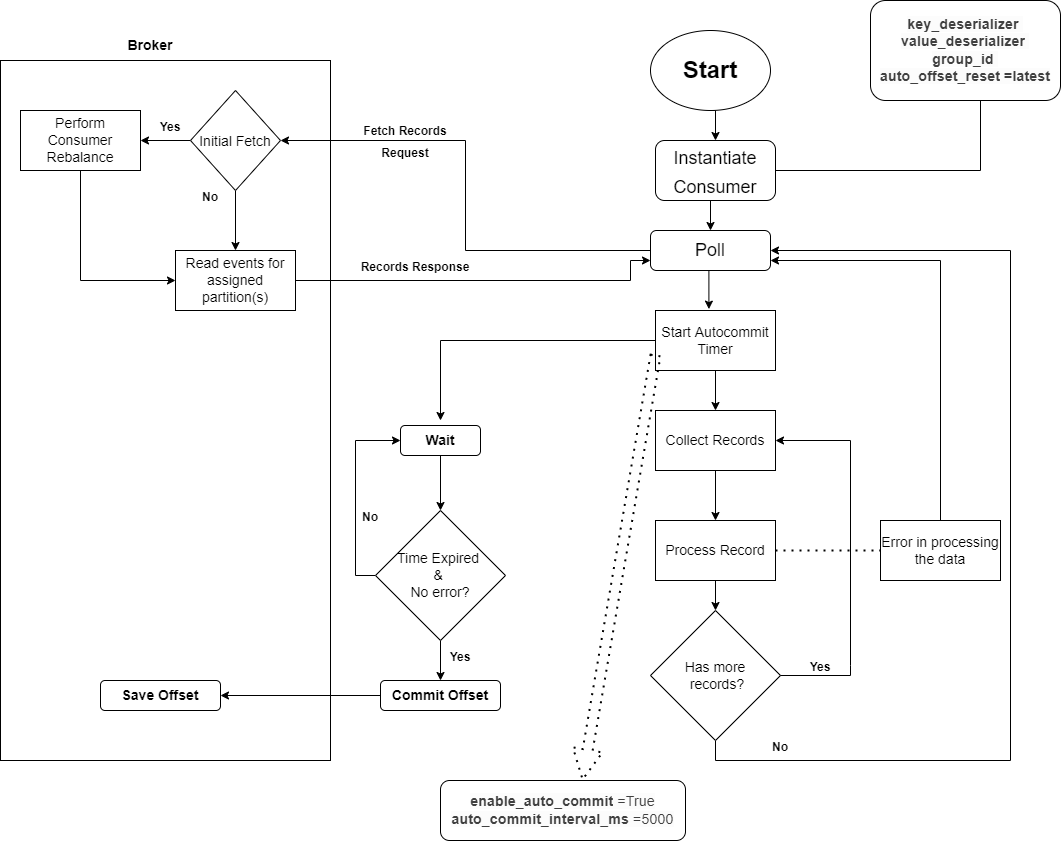
Let’s say if there’s no commit information available in the topic “***\_\_consumer\_offsets***” for the consumer group then –

* Does consumer start consuming messages from the beginning of the partition i.e. offset 0?
* (OR) will it start consuming latest messages i.e. latest offset value?

To answer this question, i***nitial offset*** comes into picture.

* The decision on whether to consume from the beginning of a topic partition or to consume new messages when there is not initial offset for the consumer group is controlled by the ***auto.offset.reset*** configuration parameter on the kafka consumer.
* ***auto.offset.reset*** can take one of the following 2 values –
  + ***earliest –*** Reset offset to the earliest offset i.e. consume from the beginning of the topic partition.
  + ***latest –*** Reset offset to the latest offset i.e. consume from end of the topic partition (**default**).
* Once the consumer group has an offset written to the topic “***\_\_consumer\_offsets***” then ***auto.offset.reset*** configuration no longer applies.
* Python Code at:
  + */main/pyhton/*6-kafka-consumer-initial-offset.py

# Kafka Consumer Internals



**Kafka Consumer Internals Process:**

1. **Instantiate Consumer**:
   * A Kafka consumer is created by specifying key configurations:
     + **value\_deserializer**: Defines how to deserialize the value of the messages from Kafka topics.
     + **key\_deserializer**: Defines how to deserialize the key of the messages.
     + **group\_id**: Specifies the consumer group the consumer belongs to.
     + **auto\_reset\_offset**: Defines the behavior when there is no initial offset or if the current offset doesn't exist (e.g., latest or earliest).
2. **Polling Mechanism**:
   * The consumer starts the polling mechanism to retrieve records from Kafka brokers.
3. **Broker Connection and Initial Fetch**:
   * When polling, the consumer connects with the Kafka broker.
   * If the consumer is requesting records for the first time (initial fetch), a consumer rebalance occurs to assign partitions.
4. **Partition Assignment**:
   * After rebalancing, the consumer reads messages from the partitions assigned to it by the broker.
5. **Auto Commit Timer Initialization**:
   * Two properties are set to manage the automatic committing of offsets:
     + **enable\_auto\_commit**: Enables or disables auto committing of offsets. **Default value is True**.
     + **auto\_commit\_interval\_ms**: Defines the interval (in milliseconds) for committing offsets. **Default value is 5000 ms.**
   * If **enable\_auto\_commit** is true, a parallel thread is started to handle auto committing.
6. **Auto Commit Process**:
   * A parallel thread waits for the **auto\_commit\_interval\_ms** to expire.
   * Once the interval is reached, the consumer commits the latest processed offset to the internal Kafka topic **\_\_consumer\_offsets**.
7. **Collecting Records**:
   * During the polling process, records (batch of records) from the attached partitions are collected for processing.
8. **Processing Records**:
   * The collected records are processed one by one by the consumer.
9. **Checking for More Messages**:
   * After processing, the consumer checks if more messages are available for processing in a batch.
   * If messages are available, it continues processing; otherwise, it returns to the polling mechanism (step 2) in an infinite loop.
10. **Error Occurrence During Processing**:
    * If an error occurs during the processing of records (step 8), **the entire process flow** is interrupted.
    * The **auto commit thread is also interrupted**, meaning the current offset is not committed to the **\_\_consumer\_offsets** topic.
11. **Error-Triggered Re-polling**:
    * When an error interrupts the processing flow, the consumer restarts polling (step 2).
    * This leads to reprocessing of the messages that were already processed but not committed, since the offset was not updated.
12. **Example of Reprocessing**:
    * If the consumer fetches 10 messages and the auto commit interval is 5 seconds, but an error occurs after 3 seconds (with only 4 messages processed), the remaining messages (and potentially the already processed messages) will be reprocessed during the next polling cycle (re-polling done by Error Workflow).
    * The offset is not updated because the auto commit process was interrupted, so the consumer uses the last committed offset, leading to possible reprocessing of messages that were already handled but not committed.
    * This issue is resolved using “***Manual Offset Commits & Exactly Once Processing***” method.
13. **Example of Lost Messages**:
    * If the consumer fetches 10 messages and the auto commit interval is 5 seconds, but an error occurs after 6 seconds (with only 8 messages processed), the remaining messages will be lost during the next polling cycle (polling done by Error Workflow) as auto offset commit for those 10 messages are already done.
    * This is called **premature** commits.
    * This issue is resolved using “***Manual Offset Commits & At Least Once Processing***” method.

# Consumer - Manual Offset Commits & At Least Once Processing

Workflow for Manual Offset Commits and At Least Once Processing can be seen below.

A diagram of a manual flowchart

Description automatically generated

**Manual Offset Commit and At Least Once Processing Process:**

1. **Disable Auto Commit**:
   * By setting ***enable\_auto\_commit = false***, the automatic offset commit process is completely disabled, ensuring the consumer has full control over when offsets are committed.
2. **Manual Offset Commit After Processing all the messages**:
   * In this mode, the consumer commits offsets manually after all the records in a batch are successfully processed.
   * This prevents committing offsets **prematurely**, giving flexibility to handle errors before marking the records as "processed."
3. **Error Occurrence During Processing**:
   * During record processing, if any error occurs while processing message from the batch of messages, the flow will be interrupted and since offset commit happens after the processing of entire batch of messages therefore offset of the successfully processed message from the batch will not be committed as well.
   * This leads to a re-polling of the entire batch of messages (including the ones that were partially processed) when the consumer resumes, ensuring no message is lost.
4. **Ensuring At Least Once Processing**:
   * This method guarantees that each message is processed **at least once**, as the offset is only committed after successfully processing the batch.
   * If even a single message in the batch fails during processing, entire flow will be interrupted therefore the offset will not be committed, allowing the entire batch to be reprocessed in the next poll.
   * This ensures that no message is skipped or lost, as the offset only moves forward when all records are correctly processed.
   * This process performs duplicate processing of the messages.

Manual offset committing provides greater control over message processing and ensures that message loss is minimized by reprocessing messages in case of failure, enabling an "**at least once**" processing guarantee.

# Consumer - Manual Offset Commits & Exactly Once Processing

Workflow for Manual Offset Commits and Exactly Once Processing can be seen below.

This process prevents the reprocessing of the same message after an error occurs, which would typically happen in the "At Least Once" processing method.

A diagram of a manual process

Description automatically generated

**Manual Offset Commit and Exactly Once Processing Process:**

1. **Disable Auto Commit:**
   * By setting enable\_auto\_commit = false, the automatic offset commit process is disabled, giving the consumer full control over when to commit offsets.
2. **Manual Offset Commit After Processing Each Message:**
   * In this mode, the consumer manually commits offsets after successfully processing each individual record within the batch.
3. **Error Occurrence During Processing:**
   * Since offsets are committed in ***\_\_consumer\_offsets*** *topic* after processing each message, the consumer avoids reprocessing messages that were processed successfully during the error re-polling flow.
   * In case of an error, only the message that failed will be reprocessed in the next poll as after the error occurs the following flow along with offset commit of the failed messages will be interrupted.
4. **Ensuring Exactly Once Processing:**
   * This method ensures that each message is processed exactly once, as the offset is only committed after the successful processing of each message.
   * If an error occurs, only messages from the failed offset onward will be reprocessed in the subsequent poll.
   * This guarantees that no message is processed more than once.

This approach guarantees **exactly-once** processing, ensuring that messages are neither lost nor reprocessed unnecessarily, maintaining data consistency across the system. This approach is widely used.

# Consumer - Manual Offset Commits & At-Most Once Processing

Workflow for Manual Offset Commits and At-Most Once Processing can be seen below.

A diagram of a process

Description automatically generated

**Manual Offset Commit and At-Most Once Processing Process:**

1. **Disable Auto Commit:**
   * By setting enable\_auto\_commit = false, the automatic offset commit process is disabled, giving the consumer full control over when to commit offsets.
2. **Manual Offset Commit After Record Collection:**
   * In this mode, the consumer manually commits offsets **after collecting records from the partition** but **before processing each individual message** within the batch. This means the offset is committed before any actual message processing begins.
3. **Error Occurrence During Processing:**
   * Since the offset is committed **before message processing**, if an error occurs during processing, then that message with the committed offset will be skipped during re-polling.
   * This happens because the offset has already been committed in ***\_\_consumer\_offsets*** *topic*, and Kafka will deliver messages beyond that committed offset.
4. **Ensuring At-Most Once Processing:**
   * This approach guarantees at-most once processing, meaning that each message is processed no more than once.
   * If an error occurs, the consumer will skip reprocessing the failed message, leading to potential message loss.
   * This ensures that each message is processed at most once, but any message that has a committed offset and encounters an error during processing will be lost and will not be retried.

This method is ideal in scenarios where message loss is acceptable, but duplicate processing must be avoided, ensuring minimal risk of reprocessing the same message.

# Manual Offset Commit Python– Exactly Once Processing Approach

**Python Code at:**

* */main/pyhton/*8-kafka-consumer-exactly-once-processing.py

## Exactly Once Processing

To enable manual offset commit with exactly once processing using Kafka Python consumer API, the following methods are used:

* **enable\_auto\_commit** :
  + Set auto commit property to false i.e. **enable\_auto\_commit =False**
* **commit**:
  + The commit method is used to manually commit the offset. It requires a dictionary with the **TopicPartition** and **OffsetAndMetadata** objects.
* **TopicPartition**:
  + This class is used to specify the **topic** and **partition** from which the consumer is reading.
  + It takes two arguments: the topic name and the partition number.
* **OffsetAndMetadata**:
  + This class is used to define the offset and metadata for manual offset commits.
  + It takes two arguments:
    - offset + 1: This tells Kafka the next message position to start from after the commit.
    - metadata: Metadata associated with the offset, often used as a timestamp.

## Consumer Rebalancing

Consumer rebalancing happens when the number of consumers in a consumer group changes, either by adding or removing a consumer. Rebalancing ensures that the load of message consumption is evenly distributed across all available consumers in the group. Here's how it works:

* **Initial Setup (1 Consumer):**
  + Start with one consumer in a consumer group that subscribes to a topic.
  + All partitions of the topic are assigned to this single consumer.
  + This consumer will consume messages from all partitions.
* **Adding a Second Consumer:**
  + Now, create another consumer in the same consumer group.
  + Kafka will detect the addition of this new consumer and trigger consumer rebalancing.
* **Rebalancing Process:**
  + Kafka will reassign the topic's partitions between the two consumers in the group.
  + The initial consumer may give up some of its assigned partitions, which are then allocated to the new consumer.
  + The new consumer will start receiving messages from the partitions it has been assigned, while the existing consumer continues consuming from the remaining partitions.

The same method is used for At-Least and At-Most once processing

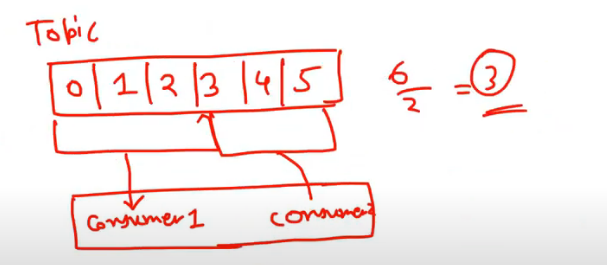
# Kafka Partition Assignment Strategies Across Multiple Consumers within same Consumer Group

## RangePartitionAssignor

The **RangePartitionAssignor** is the built-in partition assignment strategies in Kafka. It determines how partitions of a topic are distributed across consumers within the same consumer group.

**Partition Assignment**:

* + It assigns contiguous partitions (in ranges) to consumers.
  + For each topic, partitions are divided among consumers in a way that each consumer gets a range of partitions.



**How it Works**:

* + The partitions for a given topic are sorted.
  + The consumers are also sorted.
  + Kafka assigns a consecutive range of partitions to each consumer.

**Example:**

Consider a topic with 6 partitions (P0, P1, ..., P5) and a consumer group with 2 consumers (C1, C2).

* **Sorted Partitions**: P0, P1, P2, P3, P4, P5
* **Sorted Consumers**: C1, C2
* **Range Partitioning**:
  + Consumer C1 gets partitions P0, P1, P2
  + Consumer C2 gets partitions P3, P4, P5

## RoundRobinPartitionAssignor

The **RoundRobinPartitionAssignor** is built-in partition assignment strategy in Kafka that focuses on distributing partitions as evenly as possible among consumers within the same consumer group.

**Partition Assignment**:

* + It assigns partitions in a round-robin fashion to consumers.

A diagram of a diagram

Description automatically generated

**How it works:**

* **Consumers and partitions are sorted** in lexicographic order.
* Kafka assigns partitions to consumers one by one in a circular manner, ensuring an even distribution of partitions among all consumers.
* **Non-contiguous partitions** are assigned to each consumer.

**Example:**

Consider a topic with 6 partitions (P0, P1, ..., P5) and a consumer group with 2 consumers (C1, C2).

* **Sorted Partitions**: P0, P1, P2, P3, P4, P5
* **Consumers**: C1, C2
* **Round-Robin Assignment**:
  + Consumer C1 gets partitions P0, P2, P4
  + Consumer C2 gets partitions P1, P3, P5

## Python Code

* **Refer Code:** 
  + 9.1-kafka-producer-partition-consumer-mapping.py
  + 9.2-kafka-consumer-partition-consumer-mapping.py
  + 9.3-kafka-consumer-partition-consumer-mapping.py
  + **Execution Steps:** 
    1. Create Topic before the execution of the python script
    2. Execute 1st python script for Producer
    3. Execute 2nd Python script for Consumer with ***RangePartitionAssignor*** : This will show how all the partitions are assigned to a single Consumer
    4. Execute 3rd python script for Consumer with ***RangePartitionAssignor*** : This will show how rebalancing happens using Range Partition Assignor strategy
    5. Follow same steps for ***RoundRobinPartitionAssignor***
* **partition\_assignment\_strategy**property is used to assign ***“RangePartitionAssignor”*** and ***“RoundRobinPartitionAssignor”***  during consumer configurations.
* **Rebalance Listener:** 
  + It is a mechanism to respond to partition assignment and revocation events, which happen during consumer rebalancing (e.g., when a consumer joins or leaves a consumer group).
  + ***ConsumerRebalanceListener*** class is used implementing the rebalance listener mechanism.
  + When Kafka rebalances partitions, it will trigger the rebalance listener's ***on\_partitions\_assigned*** and ***on\_partitions\_revoked*** methods.

# Consumer Lag Analysis

* Consumer lag is one of the key metrics of Kafka Monitoring.
* Consumer lag happens when Consumer’s message processing speed is less than the speed at which Producer sends the message.
* So, the difference between how fast the producer places records on the broker and how fast consumers read those messages ***is called consumer lag***.
* Example:
  + Producer sends messages to a topic and at a given point in time, let’s say offset is 999
  + Consumer consumes the messages and commit the offset to \_\_consumer\_offsets topic, so at the same given point in time, let’s say consumer commits till offset 994 i.e. consumer processed messages till offset 994.
  + Now, Consumer lag at the same point in time will be 999-994 = 5
* If Consumer lag gradually increases over the period, then that’s concerning.
* If the lag is consistent then, there’s chances that consumer may catch-up eventually.
* **Python Code:**
  + Refer:
    - 10.1-kafka-producer-consumer-lag.py
    - 10.2-kafka-consumer-consumer-lag.py
    - 10.3-kafka-consumer-consumer-lag.py
    - **Execution Steps:** 
      1. Create Topic before the execution of the python script
      2. Execute 1st python script for Producer with sleep of 0.4 – 0.4 seconds denotes the speed of publishing new messages to the topic
      3. Execute 2nd Python script for Consumer with sleep of 0.8 – 0.8 seconds denotes the speed of processing by consumer
      4. Execute following ***kafka-consumer-group*** command to check the lag

***Script:***

|  |
| --- |
| **Check consumer lag in a consumer group**  *kafka-consumer-groups --bootstrap-server localhost:9092 --group demo-consumer-group --describe* |

* + - 1. Execute 3rd python script for Consumer with sleep of 0.8 – 0.8 seconds denotes the speed of processing by consumer
      2. Execute ***kafka-consumer-group*** command (step 4) to check the lag after creating a new consumer in the same consumer group.
* **Python Code Explanation:**

Producer is created which will send a message to a topic with 2 partitions in every 0.4 seconds and consumer is created in a consumer group which will consume that message in 0.8 seconds.

Since message publishing speed is greater than consumer’s consumption speed (consumer group’s speed will also be 0.8 Seconds) therefore there will be consumer lag which will exponentially increase.

As can be seen in the below snippet, Lag is increasing.

A screen shot of a computer code

Description automatically generated

Another consumer is created in the same consumer group with consumption speed of 0.8 seconds, Consumer Rebalancing happens.

Now there are 2 consumers in the same consumer group with consumption speed of 0.8 seconds therefore Consumer Group’s overall consumption speed will be 0.8/2 = 0.4 seconds, which will be same as speed of Producer.

Due to this additional consumer in the consumer group, lag will become constant.

As can be seen in the below snippet, after the addition of new consumer in the consumer group, lag is now consistent.

A screen shot of a computer

Description automatically generated

# Why don’t Kafka allow multiple consumers within Same Group to consume messages from same partition?

1. **Order Guarantee:**
   * Kafka guarantees that messages within a **partition** are consumed **in order**. If multiple consumers in the same group were allowed to consume from the same partition, this guarantee would break because each consumer could process messages at different rates or in different sequences.
   * By assigning each partition to only one consumer within a group, Kafka ensures that the order of messages is maintained across consumers.
2. **Exactly Once Processing:**
   * Kafka’s consumer group model provides exactly-once processing guarantees (depending on configuration). If multiple consumers in the same group could read from the same partition, duplicate processing of messages would be likely.
   * Kafka’s model ensures that each partition is assigned to only one consumer, so there is a clear responsibility for processing, committing offsets, and managing the state of the partition. This avoids inconsistencies in message acknowledgment or offset handling.
3. **Efficient Partition Ownership:**
   * Partitions are designed to be consumed exclusively by a single consumer within a group to avoid resource conflict. If multiple consumers were allowed to consume from the same partition, they would need to coordinate to ensure that they don't process the same message simultaneously, introducing overhead and inefficiency.
   * By having exclusive ownership, Kafka can simplify the logic of message consumption, state management, and offset tracking.
4. **Simplified Offset Management:**
   * Kafka uses consumer offsets to track which messages have been processed by each consumer. If multiple consumers within the same group were allowed to consume from the same partition, managing offsets would be highly complex because different consumers would process different messages at different times.
   * In the current design, since only one consumer processes a partition's messages, Kafka can reliably track offsets and commit them without conflicts.
5. **Scalability via Partition Assignment:**
   * Kafka encourages horizontal scalability by assigning each partition to one consumer in the group. The more partitions you have, the more consumers you can add to a group, allowing you to distribute the workload across multiple consumers efficiently.
   * If multiple consumers were allowed to consume from the same partition, the ability to scale the consumption workload across multiple consumers would be limited because you would lose the clear partition-to-consumer mapping that enables efficient parallelism.
6. **Design Philosophy:**
   * Kafka’s consumer group model is based on the concept that each partition is the unit of parallelism, and each partition should have only one consumer within a group. This design keeps Kafka’s processing model simple, predictable, and efficient, with clear guarantees around message ordering, offset management, and partition ownership.
   * If you need multiple consumers to consume the same data, Kafka allows you to create multiple consumer groups. Each group can consume the same topic independently, enabling fan-out of data.

# Appendix

## Kafka API – Python Code

**Refer Python Code in folder */main/pyhton/****:*

|  |  |
| --- | --- |
| Script | Use Case |
| 0-kafka-producer.py | For producing messages |
| 1-kafka-producer-custom-partitioner.py | Custom Partitioner |
| 2-kafka-send-message-fire-forget.py | Fire and Forget method of sending messages |
| 3-kafka-send-message-synchronous.py | Synchronous method of sending messages |
| 4-kafka-send-message-asynchronous.py | Asynchronous method of sending messages |
| 5-kafka-send-message-to-specific-partition.py | Send messages to specific partition, to test consumer group |
| 6-kafka-consumer-initial-offset.py | Consumer script with initial offset |
| 7-kafka-consumer-offset-commit.py | Consumer script with offset commit |
| 8-kafka-consumer-exactly-once-processing.py | Consumer – Exactly once processing |
| 9.1-kafka-producer-partition-consumer-mapping.py | Producer for the example of partition assignment across consumer |
| 9.2-kafka-consumer-partition-consumer-mapping.py | Consumer 1 for the example of partition assignment across consumer |
| 9.3-kafka-consumer-partition-consumer-mapping.py | Consumer 2 to check rebalancing for the example of partition assignment across consumer |
| 10.1-kafka-producer-consumer-lag.py | Producer for the example of Consumer Lag |
| 10.2-kafka-consumer-consumer-lag.py | Consumer 1 for the example of Consumer Lag |
| 10.3-kafka-consumer-consumer-lag.py | Consumer 2 to check rebalancing and reducing lag - for the example of Consumer Lag |
|  |  |

**Kafka Python References**:

* + Refer link for Kafka Python Producer Documentation

<https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html>

* + Refer link for producer and consumer python example

<https://kafka-python.readthedocs.io/en/master/usage.html#kafkaproducer>

* + Refer link for Kafka Compression- Producer and Topic

<https://www.confluent.io/blog/apache-kafka-message-compression/#what-makes-data-compression-in-a-kafka-producer-work>

<https://dzone.com/articles/compressing-your-big-data-tips-and-tricks>

## Producer API - Python

* **value\_serializer** and **key\_serializer** property can be used to serialize the messages to the byte code. If these are not set, then Kafka will use its default serializer. Value of **value\_serializer** and **key\_serializer** will be a function and should have 1 parameter (*key*)
* **Custom Partitioner** function can be used to controlstoring a message to a partition. **Custom Partitioner** function should have 3 parameters
  + ***key*** – Parameter for partitioning key
  + ***all\_partitions*** - List of all partitions sorted by Partition ID
  + ***available*** - List of available partitions in no particular order
* After configuring Producer, it will be used to send the messages. Producer send method can take following parameters –
  + Topic
  + Partition (optional)
  + Key (optional)
  + Value

These are the same parameters that is part of partitionRecord

* If a producer is configured with custom partitioner and while sending a message if partition parameter is used, then the partition mentioned here will supersede the custom partitioner.
* ***producer.flush()*** :This is the python Kafka producer API method which is used to flush the messages stored in buffer to Kafka cluster.

Let’s say, if this producer is not flushed and it is closed then messages in the buffer will be lost and wont be sent to Kafka cluster.

* ***producer.close()*** : This is the python Kafka producer method which will be used close the producer connection to Kafka cluster.

## Consumer API - Python

* **value\_deserializer and key\_deserializer**:
  + These properties define how the key and value of the Kafka messages should be deserialized after being consumed.
  + For example, JSON deserialization can be specified as value\_deserializer=lambda x: json.loads(x.decode('utf-8')).
* **enable\_auto\_commit**:
  + If set to True, offsets are committed automatically in the background at regular intervals.
  + If set to False, the consumer must commit the offsets manually.
* **auto\_commit\_interval\_ms**:
  + Specifies the interval (in milliseconds) at which the offsets are automatically committed if enable\_auto\_commit is set to True.
  + Example: auto\_commit\_interval\_ms=5000 commits offsets every 5 seconds.
* **auto\_offset\_reset**:
  + Determines the behavior when there is no valid offset for a partition.
  + Options:
    - "earliest": Start reading from the beginning of the partition.
    - "latest": Start reading from the latest offset.
  + Example: auto\_offset\_reset='earliest'.
* **group\_id**:
  + Specifies the consumer group this consumer belongs to.
  + Consumers in the same group share the load of consuming messages from the topic.
* **poll**:
  + The method used by the consumer to fetch records from Kafka.
  + Example: poll(timeout\_ms=1000) fetches records and waits for up to 1 second if no messages are available.
* **commit**:
  + The **commit** method is used to manually commit the offset. It requires a dictionary with the TopicPartition and OffsetAndMetadata objects.
  + **TopicPartition**:
    - This is used to specify the topic and partition from which the consumer is reading.
    - It takes two arguments: the topic name and the partition number
  + **OffsetAndMetadata:**
    - This is used to define the offset and metadata for manual offset commits.
    - It takes two arguments:
      * offset + 1: This tells Kafka the next message position to start from after the commit.
      * metadata: Metadata associated with the offset, often used as a timestamp.
* **partition\_assignment\_strategy**
  + This property is used to assign ***“RangePartitionAssignor”*** and ***“RoundRobinPartitionAssignor”***  during consumer configurations
* **Rebalance Listener:** 
  + It is a mechanism to respond to partition assignment and revocation events, which happen during consumer rebalancing (e.g., when a consumer joins or leaves a consumer group).
  + ***ConsumerRebalanceListener*** class is used implementing the rebalance listener mechanism.
  + When Kafka rebalances partitions, it will trigger the rebalance listener's ***on\_partitions\_assigned*** and ***on\_partitions\_revoked*** methods.