**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Kafka\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**What are different Messaging system available in the market?**

**Ans-**

The following table describes some of the popular high throughput messaging systems −

|  |  |
| --- | --- |
| **Distributed messaging system** | **Description** |
| Apache Kafka | Kafka was developed at LinkedIn corporation and later it became a sub-project of Apache. Apache Kafka is based on brokerenabled, persistent, distributed publish-subscribe model. Kafka is fast, scalable, and highly efficient. |
| RabbitMQ | RabbitMQ is an open source distributed robust messaging application. It is easy to use and runs on all platforms. |
| JMS(Java Message Service) | JMS is an open source API that supports creating, reading, and sending messages from one application to another. It provides guaranteed message delivery and follows publish-subscribe model. |
| ActiveMQ | ActiveMQ messaging system is an open source API of JMS. |
| ZeroMQ | ZeroMQ is broker-less peer-peer message processing. It provides push-pull, router-dealer message patterns. |
| Kestrel | Kestrel is a fast, reliable, and simple distributed message queue. |

**-------------------------------------------X----------------------------------------------**

**What is Kafka Broker and Cluster?**

**Ans-**

* A **Kafka broker** allows consumers to fetch messages by topic, partition and offset.
* **Kafka brokers** can create a **Kafka cluster** by sharing information between each other directly or indirectly using Zookeeper.
* A **Kafka cluster** has exactly one **broker** that acts as the Controller.

-------------------------------------X-----------------------------------------

**How many brokers can be created in a Kafka cluster?**

Ans- A Kafka cluster can have, **10, 100, or 1,000 brokers** in a cluster if needed.

-------------------------------X---------------------------------------

**Is Kafka a message queue?**

Ans-

* We can use **Kafka as a Message Queue or a Messaging System**.
* But as a distributed streaming platform Kafka has several other usages for stream processing or storing data.
* We can use Apache Kafka as: Storage System: a fault-tolerant, durable and replicated storage system
* **Messaging System**: a highly scalable, fault-tolerant and distributed Publish/Subscribe messaging system.
* **Storage System**: a fault-tolerant, durable and replicated storage system.
* **Streaming Platform**: on-the-fly and real-time processing of data as it arrives.

**---------------------------------X------------------------------------------**

**Why do we use connector and stream api in Kafka?**

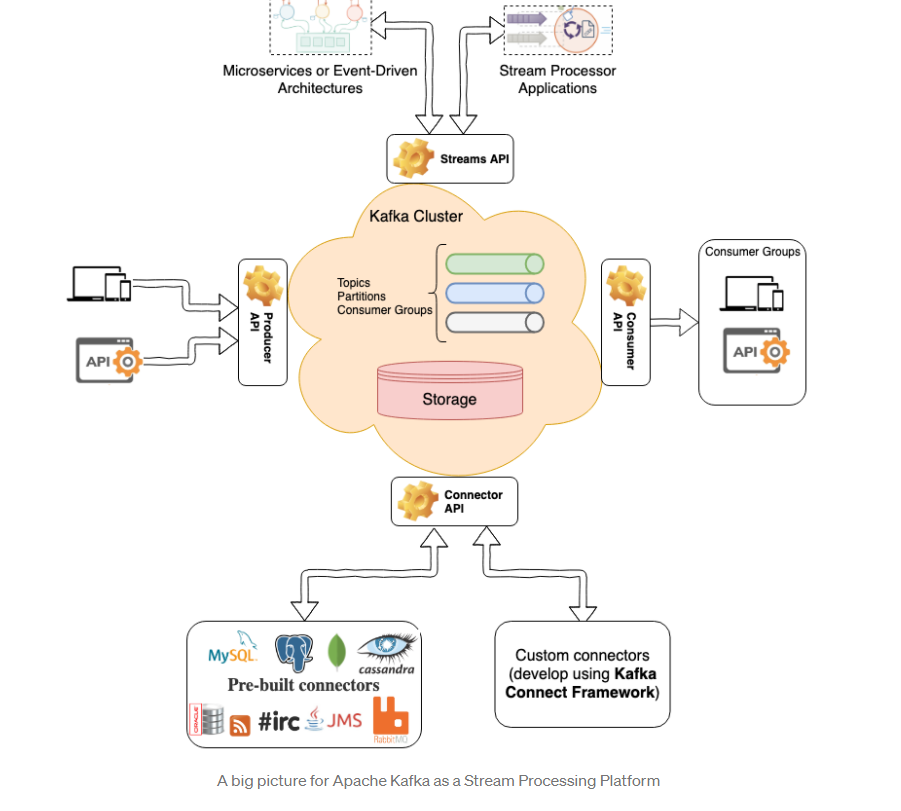
**Ans-**

* **Connector API**: Applications can easily integrate with other systems by this API and send or receive messages (or records) on top of the Kafka platform. Developers can create reusable connectors in a standardizes way. A connector can be a producer or consumer or both, For example, a connector for a relational database can publish changes of a table in a Kafka topic or store messages from a Kafka topic to a table. There are several [ready-to-use Kafka Connectors](https://www.confluent.io/hub/), which can use it easily.
* **Streams API**: By using this API, you can easily develop an application using stream processing concepts, such as transforming streams using high-level operations, data parallelism, distributed coordination and …(instead of using Producer & Consumer API)

**----------------------------------------------X-------------------------------------------------------**

**Explain about Kafka Architecture?**

**Ans-**

****

**----------------------------------X-----------------------------------------------------**

**What is Kafka Broker?**

**Ans-**

* A Broker is **a Kafka server that runs in a Kafka Cluster**.
* Kafka Brokers form a cluster.
* The Kafka Cluster consists of many Kafka Brokers on many servers.
* Broker sometimes refer to more of a logical system or as Kafka as a whole.

------------------------------------X-------------------------------------------------------

**Why does Kafka use polling?**

Ans-

* It **creates any threads necessary**, connects to servers, joins the group, etc.
* Consumer is not thread safe - you can't call its methods from different threads at the same time or else you'll get an exception.
* You have to call poll once in a while to ensure it is alive and connected to Kafka

------------------------------------X----------------------------------------

**What is Polling in Kafka?**

Ans-

The Kafka consumer uses the poll method **to get N number of records**. Consumers in the same group divide up and share partitions as we demonstrated by running three consumers in the same group and one producer. Each consumer groups gets a copy of the same

--------------------------------------------X--------------------------------

**How are messages stored in topic partitions in Kafka?**

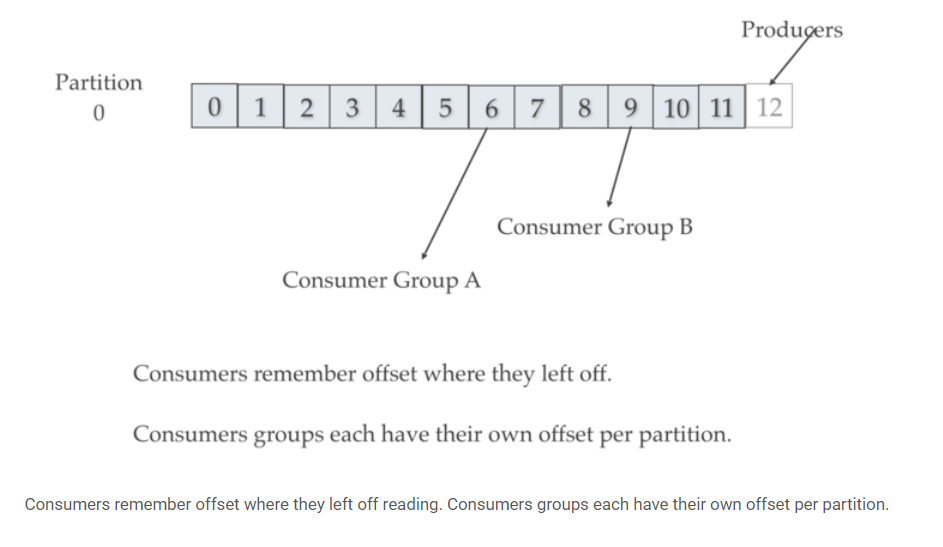
Messages (records) are stored as **serialized bytes**; the consumers are responsible for de-serializing the message. ... Therefore, all messages on the same partition are pulled by the same task. Topic partitions contain an ordered set of messages and each message in the partition has a unique offset.

---------------------------------------X-----------------------------------------

**What is consumer groups and how it works in Kafka?**

Ans-

* You group consumers into a consumer group by use case or function of the group.
* One consumer group might be responsible for delivering records to high-speed, in-memory microservices while another consumer group is streaming those same records to Hadoop.
* Consumer groups have names to identify them from other consumer groups.
* A consumer group has a unique id.
* Each consumer group is a subscriber to one or more Kafka topics.
* Each consumer group maintains its offset per topic partition
* If you need multiple subscribers, then you have multiple consumer groups.
* A record gets delivered to only one consumer in a consumer group.
* Each consumer in a consumer group processes records and only one consumer in that group will get the same record
* Consumers in a consumer group load balance record processing.



--------------------------------------------------X---------------------------------

**How Kafka consumer shares load?**

Ans-

* Kafka consumer consumption divides partitions over consumer instances within a consumer group.
* Each consumer in the consumer group is an exclusive consumer of a “fair share” of partitions.
* This is how Kafka does load balancing of consumers in a consumer group.
* Consumer membership within a consumer group is handled by the Kafka protocol dynamically.
* If new consumers join a consumer group, it gets a share of partitions.
* If a consumer dies, its partitions are split among the remaining live consumers in the consumer group.
* This is how Kafka does fail over of consumers in a consumer group.

-------------------------------X----------------------------------------

**How Kafka Manages Consumer fail overs?**

Ans-

* Consumers notify the Kafka broker when they have successfully processed a record, which advances the offset.
* If a consumer fails before sending commit offset to Kafka broker, then a different consumer can continue from the last committed offset.
* If a consumer fails after processing the record but before sending the commit to the broker, then some Kafka records could be reprocessed.
* In this scenario, Kafka implements the at least once behavior, and you should make sure the messages (record deliveries ) are idempotent.

---------------------------------------X-------------------------------------

**What is Offset in Kafka and How Kafka does offset management?**

**Ans-**

Kafka stores offset data in a topic called "\_\_consumer\_offset". These topics use log compaction, which means they only save the most recent value per key.

When a consumer has processed data, it should commit offsets. If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in "\_\_consumer\_offset" or as discussed another consumer in the consumer group can take over.

---------------------------X-----------------------------------

**How is Kafka topics partitioned?**

Ans-

* Kafka breaks topic logs up into partitions.
* A record is stored on a partition usually by record key if the key is present and round-robin if the key is missing (default behavior).
* The record key, by default, determines which partition a producer sends the record.
* Kafka uses partitions to scale a topic across many servers for producer writes.
* Kafka also uses partitions to facilitate parallel consumers
* Consumers consume records in parallel up to the number of partitions.
* The order guaranteed per partition. If partitioning by key then all records for the key will be on the same partition which is useful if you ever have to replay the log.
* Kafka can replicate partitions to multiple brokers for failover.

-----------------------------X---------------------------------------------

**What is Offset in Kafka?**

Ans-

* Kafka maintains record order only in a single partition.
* A partition is an ordered, immutable record sequence.
* Kafka continually appended to partitions using the partition as a structured commit log.
* Records in partitions are assigned sequential id number called the offset.
* The offset identifies each record location within the partition.
* Topic partitions allow Kafka log to scale beyond a size that will fit on a single server. Topic partitions must fit on servers that host it, but topics can span many partitions hosted on many servers.
* Also, topic partitions are a unit of parallelism - a partition can only be worked on by one consumer in a consumer group at a time.
* Consumers can run in their own process or their own thread. If a consumer stops, Kafka spreads partitions across the remaining consumer in the same consumer group.

**-------------------------------------X-----------------------------**

**How replication happens in Kafka?**

**Ans-**

* Kafka can replicate partitions across a configurable number of Kafka servers which is used for fault tolerance.
* Each partition has a leader server and zero or more follower servers. Leaders handle all read and write requests for a partition.
* Followers replicate leaders and take over if the leader dies.
* Kafka uses also uses partitions for parallel consumer handling within a group
* Kafka distributes topic log partitions over servers in the Kafka cluster.
* Each server handles its share of data and requests by sharing partition leadership.

**Replication: Kafka Partition Leaders, Followers and ISRs.**

Kafka chooses one broker’s partition’s replicas as leader using ZooKeeper.  
The broker that has the partition leader handles all reads and writes of records for the partition. Kafka replicates writes to the leader partition to followers (node/partition pair). A follower that is in-sync is called an ISR (in-sync replica). If a partition leader fails, Kafka chooses a new ISR as the new leader.

**----------------------------------------------X---------------------------------------------**

**How to run Kafka nodes on different server?**

**Ans-**

You need to start the Kafka brokers with this configuration file. This is assuming that you have already started ZooKeeper and have a single Kafka node that is running:

1. **> bin/kafka-server-start.sh config/server-1.properties &**
2. **...**
3. **> bin/kafka-server-start.sh config/server-2.properties &**

**# Licensed to the Apache Software Foundation (ASF) under one or more**

**# contributor license agreements. See the NOTICE file distributed with**

**# this work for additional information regarding copyright ownership.**

**# The ASF licenses this file to You under the Apache License, Version 2.0**

**# (the "License"); you may not use this file except in compliance with**

**# the License. You may obtain a copy of the License at**

**#**

**# http://www.apache.org/licenses/LICENSE-2.0**

**#**

**# Unless required by applicable law or agreed to in writing, software**

**# distributed under the License is distributed on an "AS IS" BASIS,**

**# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.**

**# See the License for the specific language governing permissions and**

**# limitations under the License.**

**# see kafka.server.KafkaConfig for additional details and defaults**

**############################# Server Basics #############################**

**# The id of the broker. This must be set to a unique integer for each broker.**

**broker.id=1**

**################### Socket Server Settings #############################**

**# The address the socket server listens on. It will get the value returned from**

**# java.net.InetAddress.getCanonicalHostName() if not configured.**

**# FORMAT:**

**# listeners = listener\_name://host\_name:port**

**# EXAMPLE:**

**# listeners = PLAINTEXT://your.host.name:9092**

**#listeners=PLAINTEXT://:9092**

**port=9093**

**# Hostname and port the broker will advertise to producers and consumers. If not set,**

**# it uses the value for "listeners" if configured. Otherwise, it will use the value**

**# returned from java.net.InetAddress.getCanonicalHostName().**

**#advertised.listeners=PLAINTEXT://your.host.name:9092**

**# Maps listener names to security protocols, the default is for them to be the same. See the config documentation for more details**

**#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL\_PLAINTEXT:SASL\_PLAINTEXT,SASL\_SSL:SASL\_SSL**

**# The number of threads that the server uses for receiving requests from the network and sending responses to the network**

**num.network.threads=3**

**# The number of threads that the server uses for processing requests, which may include disk I/O**

**num.io.threads=8**

**# The send buffer (SO\_SNDBUF) used by the socket server**

**socket.send.buffer.bytes=102400**

**# The receive buffer (SO\_RCVBUF) used by the socket server**

**socket.receive.buffer.bytes=102400**

**# The maximum size of a request that the socket server will accept (protection against OOM)**

**socket.request.max.bytes=104857600**

**############################# Log Basics #############################**

**# A comma separated list of directories under which to store log files**

**log.dirs=c:/kafka/server1**

**# The default number of log partitions per topic. More partitions allow greater**

**# parallelism for consumption, but this will also result in more files across**

**# the brokers.**

**num.partitions=1**

**# The number of threads per data directory to be used for log recovery at startup and flushing at shutdown.**

**# This value is recommended to be increased for installations with data dirs located in RAID array.**

**num.recovery.threads.per.data.dir=1**

**############## Internal Topic Settings #############################**

**# The replication factor for the group metadata internal topics "\_\_consumer\_offsets" and "\_\_transaction\_state"**

**# For anything other than development testing, a value greater than 1 is recommended to ensure availability such as 3.**

**offsets.topic.replication.factor=1**

**transaction.state.log.replication.factor=1**

**transaction.state.log.min.isr=1**

**############################# Log Flush Policy #############################**

**# Messages are immediately written to the filesystem but by default we only fsync() to sync**

**# the OS cache lazily. The following configurations control the flush of data to disk.**

**# There are a few important trade-offs here:**

**# 1. Durability: Unflushed data may be lost if you are not using replication.**

**# 2. Latency: Very large flush intervals may lead to latency spikes when the flush does occur as there will be a lot of data to flush.**

**# 3. Throughput: The flush is generally the most expensive operation, and a small flush interval may lead to excessive seeks.**

**# The settings below allow one to configure the flush policy to flush data after a period of time or**

**# every N messages (or both). This can be done globally and overridden on a per-topic basis.**

**# The number of messages to accept before forcing a flush of data to disk**

**#log.flush.interval.messages=10000**

**# The maximum amount of time a message can sit in a log before we force a flush**

**#log.flush.interval.ms=1000**

**#################### Log Retention Policy #############################**

**# The following configurations control the disposal of log segments. The policy can**

**# be set to delete segments after a period of time, or after a given size has accumulated.**

**# A segment will be deleted whenever \*either\* of these criteria are met. Deletion always happens**

**# from the end of the log.**

**# The minimum age of a log file to be eligible for deletion due to age**

**log.retention.hours=168**

**# A size-based retention policy for logs. Segments are pruned from the log unless the remaining**

**# segments drop below log.retention.bytes. Functions independently of log.retention.hours.**

**#log.retention.bytes=1073741824**

**# The maximum size of a log segment file. When this size is reached a new log segment will be created.**

**log.segment.bytes=1073741824**

**# The interval at which log segments are checked to see if they can be deleted according**

**# to the retention policies**

**log.retention.check.interval.ms=300000**

**############################# Zookeeper #############################**

**# Zookeeper connection string (see zookeeper docs for details).**

**# This is a comma separated host:port pairs, each corresponding to a zk**

**# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".**

**# You can also append an optional chroot string to the urls to specify the**

**# root directory for all kafka znodes.**

**zookeeper.connect=localhost:2181**

**# Timeout in ms for connecting to zookeeper**

**zookeeper.connection.timeout.ms=18000**

**########## Group Coordinator Settings #############################**

**# The following configuration specifies the time, in milliseconds, that the GroupCoordinator will delay the initial consumer rebalance.**

**# The rebalance will be further delayed by the value of group.initial.rebalance.delay.ms as new members join the group, up to a maximum of max.poll.interval.ms.**

**# The default value for this is 3 seconds.**

**# We override this to 0 here as it makes for a better out-of-the-box experience for development and testing.**

**# However, in production environments the default value of 3 seconds is more suitable as this will help to avoid unnecessary, and potentially expensive, rebalances during application startup.**

**group.initial.rebalance.delay.ms=0**

**------------------------------------X--------------------------------------**

### How it works…-------------------------------------------------------------

The server.properties files contain the configuration of your brokers. They all should point to the same ZooKeeper cluster. The broker.id property in each of the files is unique and defines the name of the node in the cluster. The port number and log.dir are changed so we can get them running on the same machine; else all the nodes will try to bind at the same port and will overwrite the data. If you want to run them on different machines, you need not change them.

### There's more…

To run Kafka nodes on different servers, you also need to change the ZooKeeper connection string's details in the config file:

**ZooKeeper.connect=localhost:2181**

This is good if you are running Kafka off the same server as ZooKeeper; but in real life, you would be running them off different servers. So, you might want to change them to the correct ZooKeeper connection strings as follows:

**ZooKeeper.connect=localhost:2181, 192.168.0.2:2181, 192.168.0.3:2181**

This means that you are running the ZooKeeper cluster at the localhost nodes, 192.168.0.2 and 192.168.0.3, at the port number 2181.

**-----------------------------------------X-----------------------------------**

**C:\kafka> .\bin\windows\kafka-console-consumer.bat --topic MovieTopic --bootstrap-server localhost:9092 --property print.key=true --property key.separator="-" --partition 1 --offset 6**

**Note- --group and –partition cannot be used together**

**----------------------------------X-----------------------------------------------**

**what is current off set and log end offset in kafka?**

**Ans-**

* CURRENT-OFFSET means the current max offset of the consumed messages of the partition for this consumer instance
* Whereas LOG-END-OFFSET is **the offset of the latest message in the partition**.
* **current-offset is the last committed offset of the consumer instance**, l
* **l**og-end-offset is the highest offset of the partition (hence, summing this column gives you the total number of messages for the topic)

**C:\kafka>.\bin\windows\kafka-consumer-groups.bat --bootstrap-server localhost:9092 -describe -group MovieAdminGroup**

**Consumer group 'MovieAdminGroup' has no active members.**

**GROUP TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST CLIENT-ID**

**MovieAdminGroup MovieTopic 1 15 18 3 - - -**

**MovieAdminGroup MovieTopic 2 18 19 1 - - -**

**MovieAdminGroup MovieTopic 0 17 21 4 - - -**

**-----------------------------------X--------------------------------------**

**-----------------------------------X----------------------------------------**

**can we produce the messages in kafka with specific partition?**

**Ans-**

* The **key is not the partition number** but Kafka uses the key to specify the target partition.
* The default strategy is to choose a partition based on a hash of the key or use round-robin algorithm if the key is null. If you need a custom algorithm to map the messages to partitions you need to implement org

------------------X------------------------------------------------

What is isr in kafka?

Ans-

The ISR is simply **all the replicas of a partition that are "in-sync" with the leader**. The definition of "in-sync" depends on the topic configuration, but by default, it means that a replica is or has been fully caught up with the leader in the last 10 seconds.

**---------------------------------------X--------------------------------------**

**How partitions and brokered are created?**

**Ans-**

Assuming 3 brokers are created

# Create the "movietopis" topic

$ bin/kafka-topics.bat --zookeeper localhost:2181 \

--create --topic movietopic --partitions 3 --replication-factor 2

* Kafka will create 3 logical partitions for the topic.
* Kafka will create a total of two replicas (copies) per partition.
* For each partition it will pick two brokers that will host those replicas.
* For each partition Kafka will elect a “leader” broker.

--------------------------------Kafka Stream Demo---------------------------------

**------------------------------WorldCountDemo.java----------------------------------------**

**package** com.cg.kafka;

**import** org.apache.kafka.common.serialization.Serdes;

**import** org.apache.kafka.common.utils.Bytes;

**import** org.apache.kafka.streams.KafkaStreams;

**import** org.apache.kafka.streams.StreamsBuilder;

**import** org.apache.kafka.streams.StreamsConfig;

**import** org.apache.kafka.streams.kstream.KStream;

**import** org.apache.kafka.streams.kstream.KTable;

**import** org.apache.kafka.streams.kstream.Materialized;

**import** org.apache.kafka.streams.kstream.Produced;

**import** org.apache.kafka.streams.state.KeyValueStore;

**import** java.util.Arrays;

**import** java.util.Properties;

**public** **class** WordCountApplication

{

**public** **static** **void** main(**final** String[] args) **throws** Exception {

Properties props = **new** Properties();

props.put(StreamsConfig.***APPLICATION\_ID\_CONFIG***, "wordcount-application");

props.put(StreamsConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "kafka-broker1:9092");

props.put(StreamsConfig.***DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG***, Serdes.*String*().getClass());

props.put(StreamsConfig.***DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG***, Serdes.*String*().getClass());

StreamsBuilder builder = **new** StreamsBuilder();

KStream<String, String> textLines = builder.stream("TextLinesTopic");

KTable<String, Long> wordCounts = textLines

.flatMapValues(textLine -> Arrays.*asList*(textLine.toLowerCase().split("\\W+")))

.groupBy((key, word) -> word)

.count(Materialized.<String, Long, KeyValueStore<Bytes, **byte**[]>>*as*("counts-store"));

wordCounts.toStream().to("WordsWithCountsTopic", Produced.*with*(Serdes.*String*(), Serdes.*Long*()));

KafkaStreams streams = **new** KafkaStreams(builder.build(), props);

streams.start();

}

}

**----------------------------End-----------------------Kafka-----------------------------------------------**

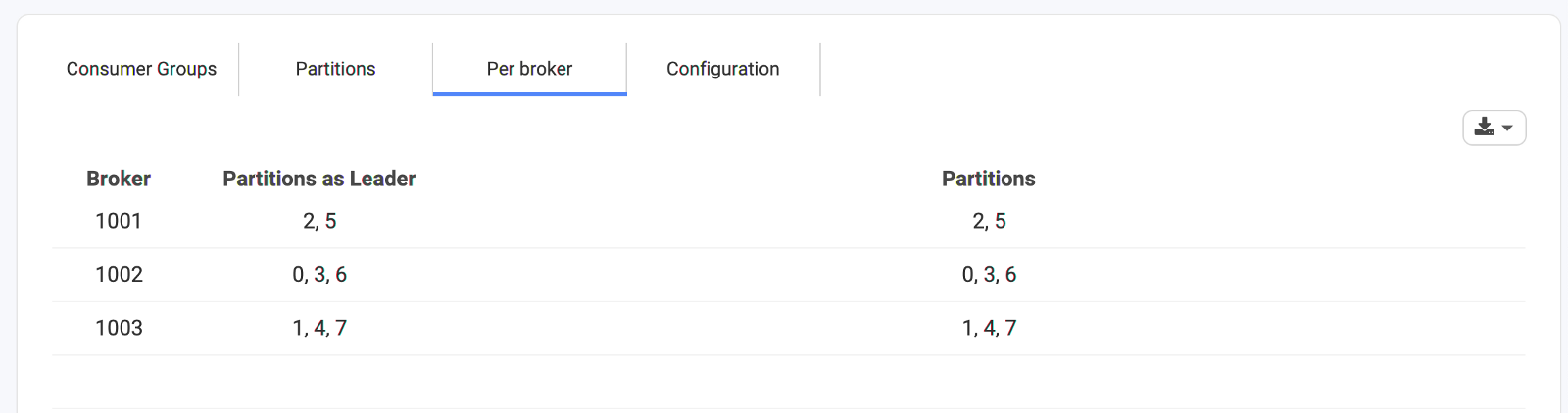
**Isr**

Here, we can see in Conduktor UI, that we have a cluster with:

* 3 brokers
* 1 topic with 8 partitions

We have created our topic with a replication factor of 1x, this means these 8 partitions are divided among these 3 brokers, so we can say the load of this topic is divided among these 3 brokers.

Conduktor shows you how the repartition is made among the brokers:

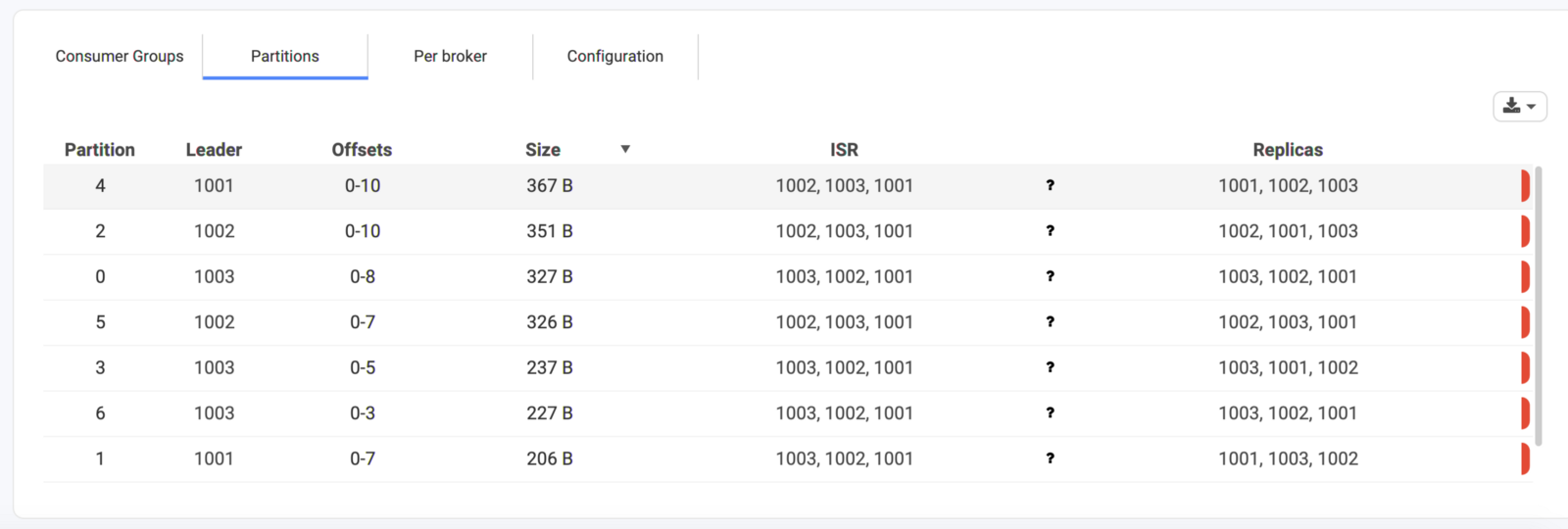


The first broker 1001 has 2 partitions, and the remaining brokers 1002 and 1003 have 3 partitions each.

What is Replication and ISR In kafka?

Ans-

* Replication is the process of having multiple copies of the data available across different servers for purpose of availability in case one of the brokers goes down.
* In Kafka, replication happens at the partition level i.e. copies of the partition are maintained at multiple broker instances.
* When we say a topic has a replication factor of 3, this means we will be having three copies of each of its partitions.
* Kafka considers that a record is committed when all replicas in the In-Sync Replica set (ISR) have confirmed that they have taking the record into account.
* While creating a Kafka topic, we can define the number of copies we want to have for the data. We define this using the replication-factor config setting.
* Let’s say we have created another topic with a 3x replication factor, here is how it is dispatched among brokers:



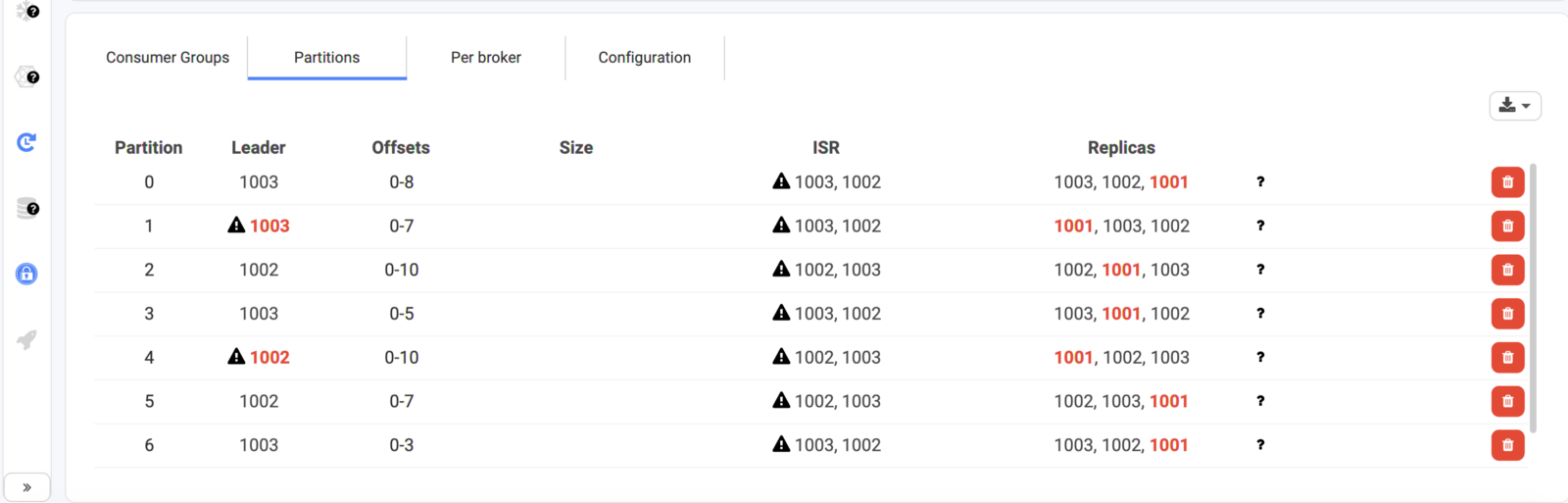
* under the Replicas column, each partition has 3 replicas
* under the ISR column, we have 3 replicas which mean all the replicas are In Sync with the partition leader, i.e. those followers that have the same data as the leader.
* It’s not mandatory to have ISR equal to the number of replicas.
* By default, if a replica is or has been fully caught up with the leader in the last 10 seconds, it is said to be “in-sync”. The setting for this time period has a server default of 10 sec that can be overridden on a per topic basis.

#### What is a Partition Leader in Kafka?

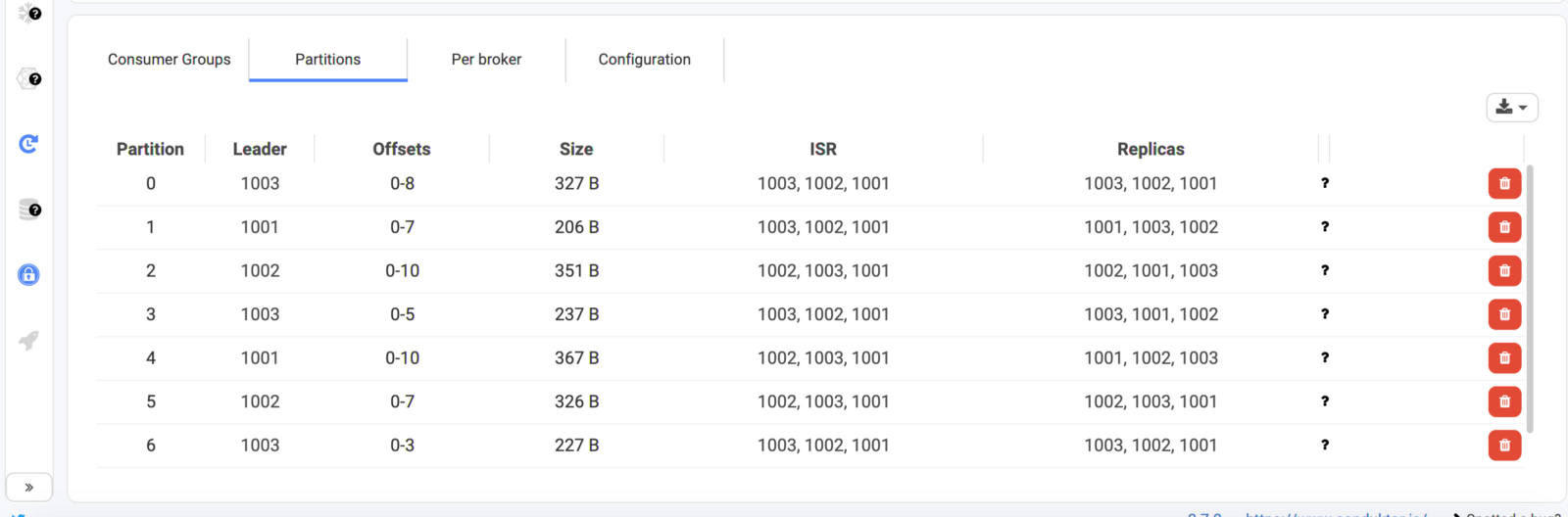
* In Kafka, there is a concept of leader for each partition.
* At any point in time, a partition can have only one broker as the leader. And only that leader can serve the data for the partition.
* Followers will sync the data from the leader.
* In the case of partition 4 and 1, broker 1001 is the leader and broker 1002 and 1003 are in sync with this leader, and replication happens from broker 1001 to broker 1002 and broker 1003.

## **What if a Kafka broker goes down?**

* Now an obvious question arises, what if one of our broker dies? or what if we have to bring one broker down for maintenance? what happens to the partition of that broker?
* In the above image, Broker 1001 is the leader of partition 4 and 1, what if broker 1001 dies? what would happen to these partitions?



* As we lost broker 1001, it is removed from ISR after 10 seconds if not configured, and Broker 1002, and 1003 still have a copy of data, and can still serve the data.
* Leader election will happen for partition 1 and 4. Now partition 1 has broker 1003 and partition 4 has broker 1002 as leaders as they were In Sync with the partition leader earlier.
* If broker 1001 comes back to life, it will try to become a leader again after replicating the data. This is handled by Kafka itself, there is no manual intervention required.



In case of any unforeseen events like hardware failure or for maintenance purposes, if we want to bring the broker down, it is recommended to have at least two brokers in the ISR (the leader + a replica) so no loss of data is guaranteed by Kafka.

**Conduktor**, a Kafka Desktop client, made it easy to manage the replication status, in-sync replicas, broker status of your cluster at one place in an easy to use GUI without running multiple commands.

**----------------------------Sprint-Kafka---------------------------------**

@KafkaListener( topicPartitions = @TopicPartition(topic = "topicName", partitionOffsets = { @PartitionOffset(partition = "0", initialOffset = "0"), @PartitionOffset(partition = "3", initialOffset = "0")}),

containerFactory = "partitionsKafkaListenerContainerFactory") **public** **void** **listenToPartition**(

@Payload String message, @Header(KafkaHeaders.RECEIVED\_PARTITION\_ID) **int** partition) { System.out.println( "Received Message: " + message" + "from partition: " + partition); }

---------------------------------------------------X-----------------

If we don't need to set the offset, we can use the *partitions* property of *@TopicPartition* annotation to set only the partitions without the offset:

@KafkaListener(topicPartitions

= @TopicPartition(topic = "topicName", partitions = { "0", "1" }))

### ****Adding Message Filter for Listeners****

We can configure listeners to consume specific types of messages by adding a custom filter. This can be done by setting a *[RecordFilterStrategy](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/adapter/RecordFilterStrategy.html)* to the KafkaListenerContainerFactory:

@Bean

**public** ConcurrentKafkaListenerContainerFactory<String, String>

**filterKafkaListenerContainerFactory**() {

ConcurrentKafkaListenerContainerFactory<String, String> factory =

**new** ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory());

factory.setRecordFilterStrategy(

record -> record.value().contains("World"));

**return** factory;

}

We can then configure a listener to use this container factory:

@KafkaListener(

topics = "topicName",

containerFactory = "filterKafkaListenerContainerFactory")

**public** **void** **listenWithFilter**(String message) {

System.out.println("Received Message in filtered listener: " + message);

}

In this listener, all the **messages matching the filter will be discarded.**

## **6. Custom Message Converters**

So far, we have only covered sending and receiving Strings as messages. However, we can also send and receive custom Java objects. This requires configuring appropriate serializer in ProducerFactory and deserializer in ConsumerFactory.

Let's look at a simple bean class, which we will send as messages: