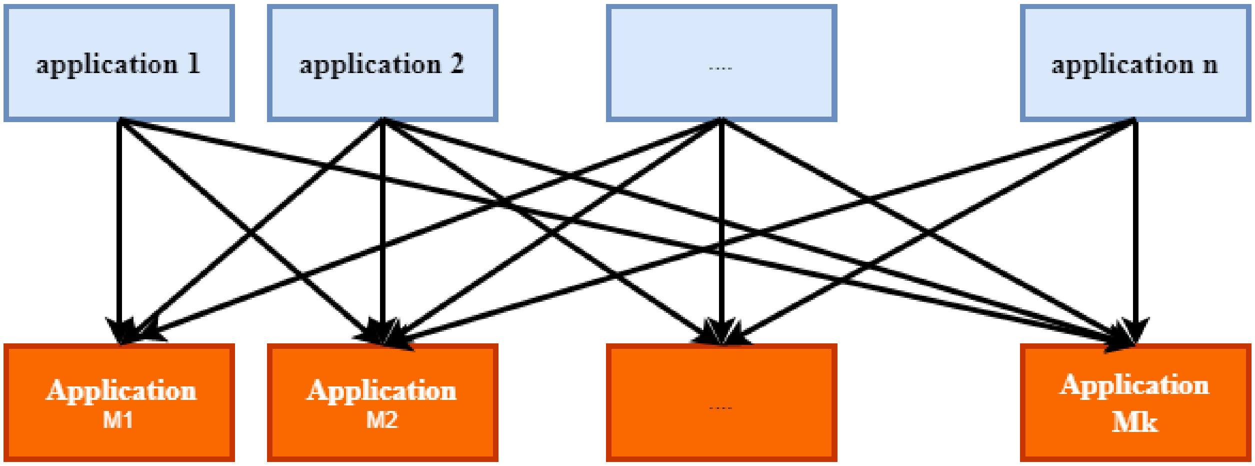
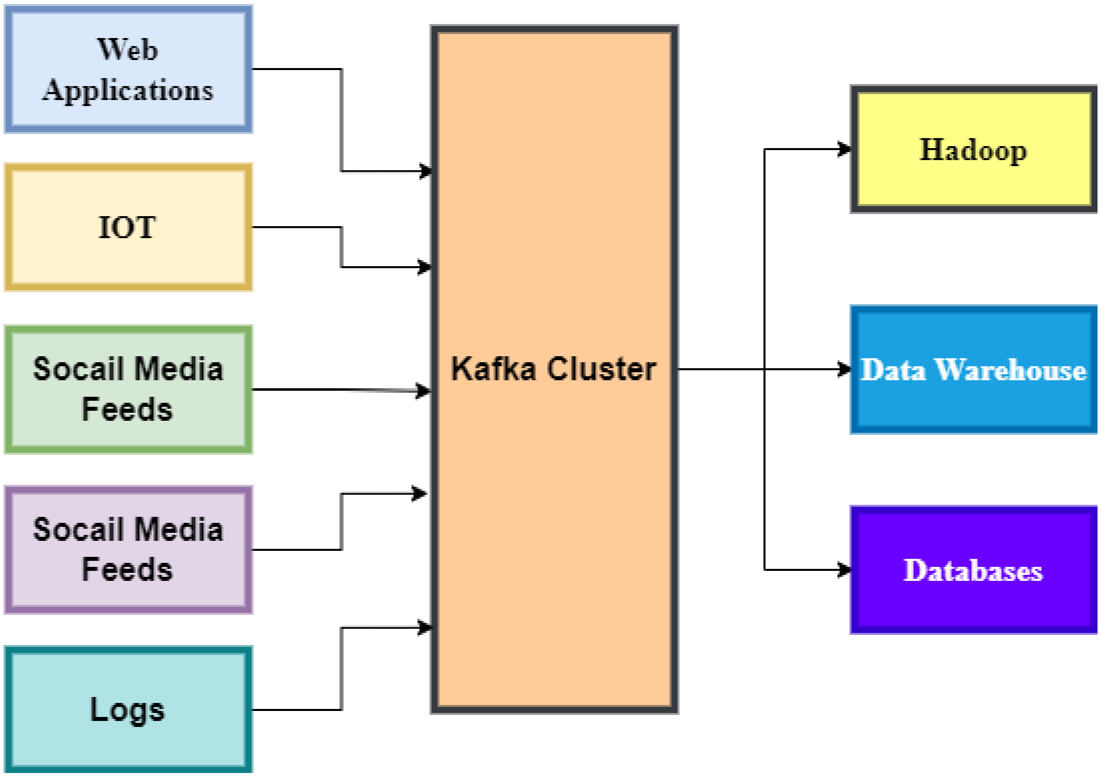
Kafka Framework

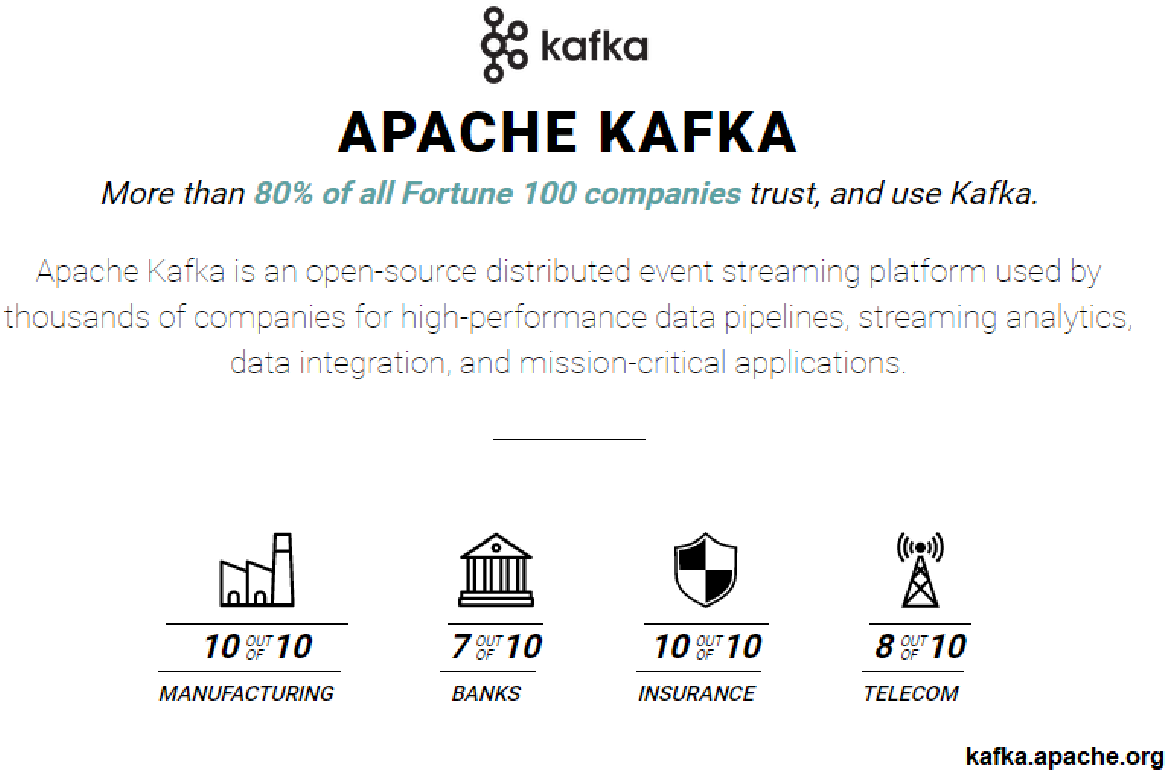
- **Overview**
  - It was originally developed by LinkedIn.
  - It is the most popular distributed streaming framework.
  - It is written in Scala and Java.
  - Kafka is a high-performance, real-time messaging open-source framework.
  - It is a distributed and partitioned messaging system.
  - It is highly fault-tolerant
  - It is horizontally Scalable
  - It can read and send millions of messages per second to several receivers.
  - Stream Processing: It can process a continuous stream of information in real-time.
  - It is a message broker.
  - It can process user activities such as clicks, navigation, and search from different sites.
  - How applications in an enterprise exchange data?
    - Each application needs to connect with multiple applications in the organization: multiple pipelines



- Kafka solution:
  - Kafka de-couples data pipelines



- **Kafka Use Cases:**
  - LinkedIn
  - Netflix: real-time monitoring and event processing
  - Twitter: as part of their Storm real-time data pipelines
  - Spotify: log delivery (from 4h down to 10s), Hadoop
  - Loggly: log collection and processing
  - Uber, Goldman Sachs, PayPal, Cisco, etc.



kafka

APACHE KAFKA

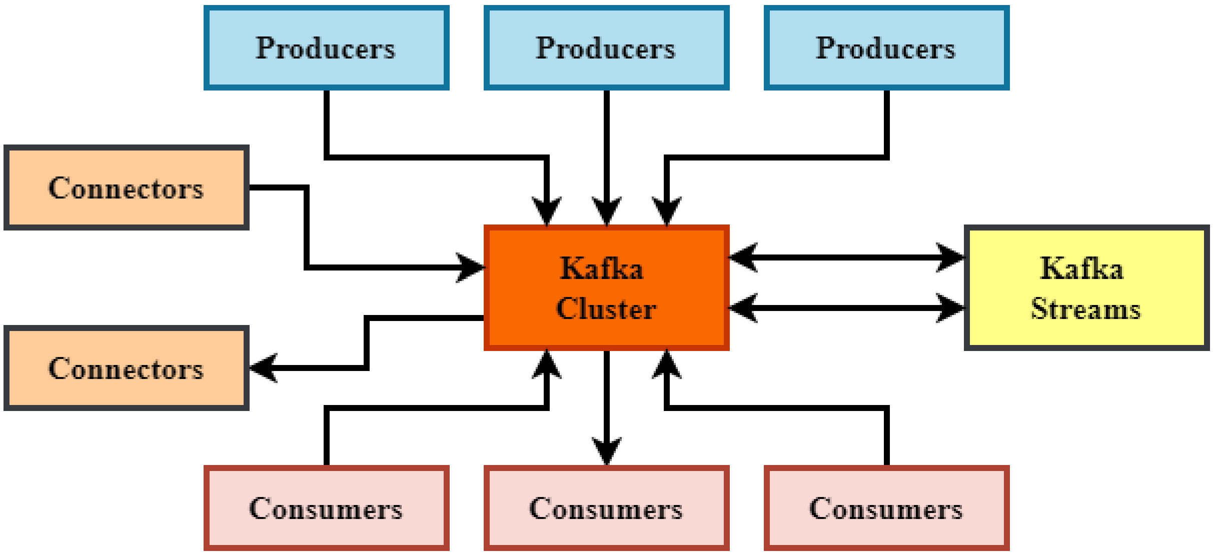
*More than **80% of all Fortune 100 companies** trust, and use Kafka.*

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

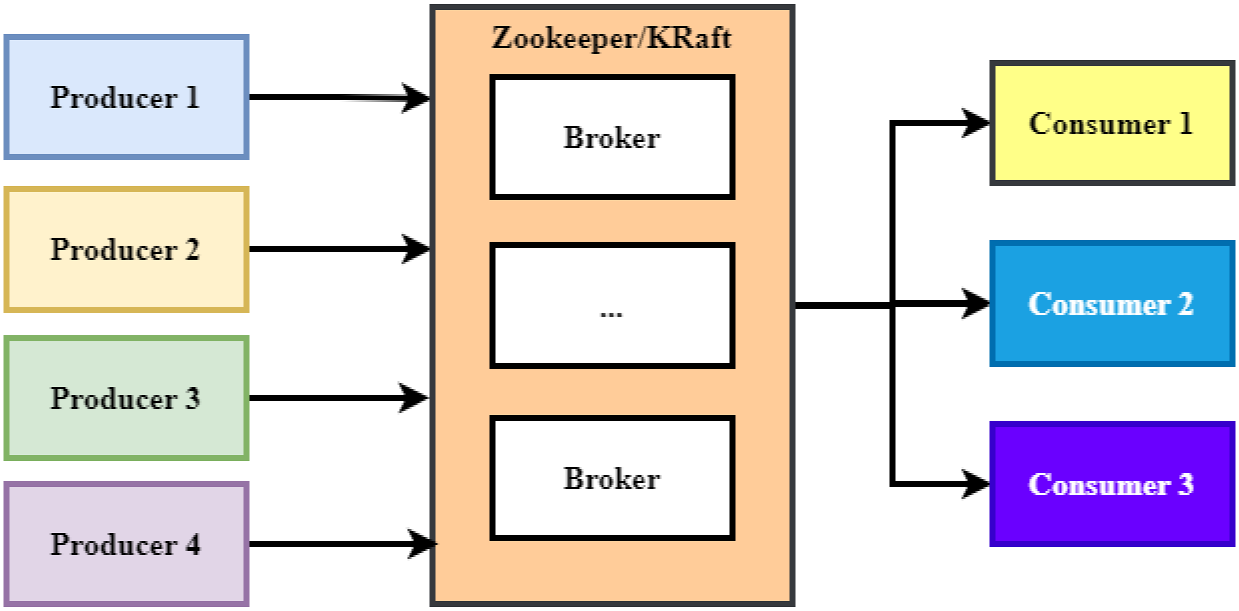
10 OUT OF 10	7 OUT OF 10	10 OUT OF 10	8 OUT OF 10
MANUFACTURING	BANKS	INSURANCE	TELECOM

kafka.apache.org

- **Kafka Architecture**
  - Apache Kafka main components:
    - Producer API and Consumer API
    - Streams API, and
    - Connector API.



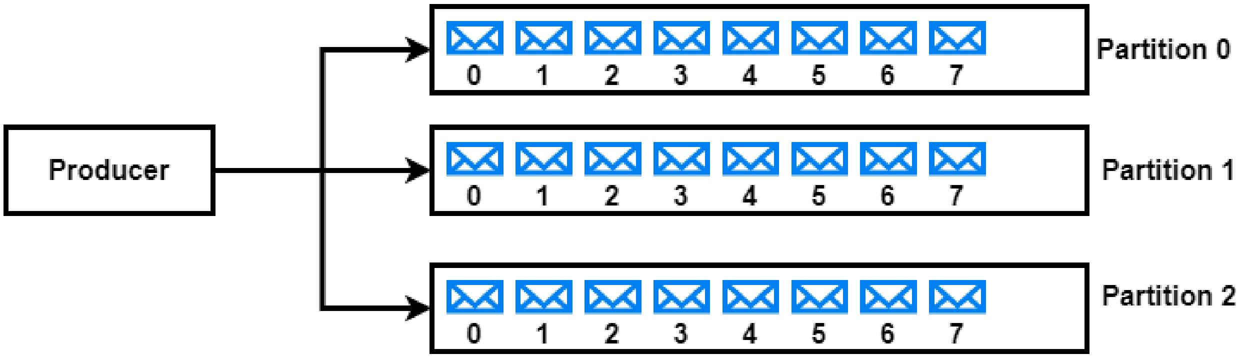
- Producer API:
    - Allows applications to publish to a Kafka topic.
- Consumer API:
    - Allows applications to subscribe to one or more topics.
- Streams API:
    - Allows applications to process an input stream from one or more topics and produce an output stream to one or more output topics.
- Connector API:
    - It allows an application to use Kafka Connectors to move data between Apache Kafka® and other external systems that you want to extract data from or publish data to.
    - For example, a connector can be used to capture every change to a table.
    - Example: MirrorMaker
        - It is a multi-cluster data replication engine based on the Kafka Connect framework.
        - It can be used to migrate data between multiple clusters.
- Main Architecture:



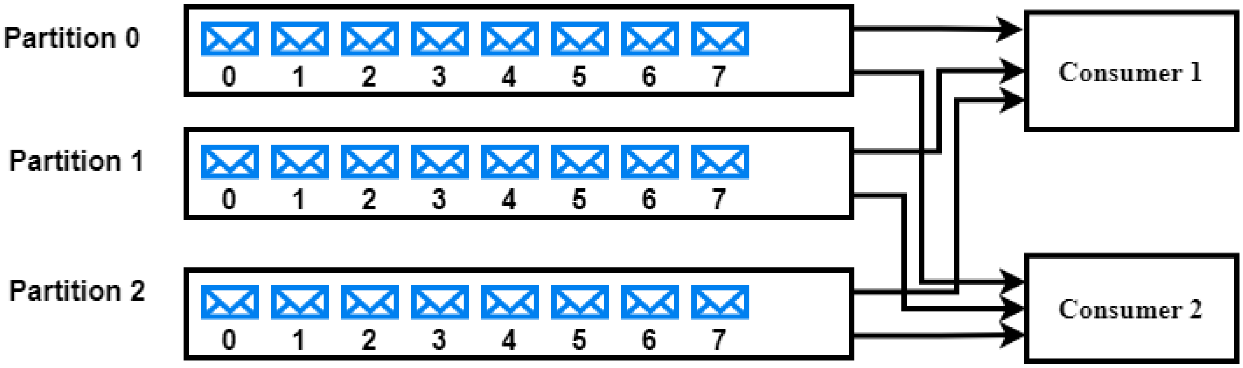
- **Kafka Broker:**
  - A Kafka cluster is a system that consists of several Brokers (servers), Topics, and Partitions for both.
  - Can consists of a single broker.
  - They distribute workloads equally among replicas and Partitions.
  - They are stateless: needs ZooKeeper to maintain cluster status.
  - Each broker can handle TB of messages.
- **Zookeeper**:
  - It is a distributed configuration and synchronization service.
  - It is a coordination interface between Kaka brokers, producers, and consumers:
    - It notifies consumers and producers of the arrival of a broker or failure of existing ones.
    - Routing requests to partition leaders.
    - It stores metadata about Kafka cluster.
    - It tracks the topics, number of partitions assigned to those topics, and leaders/followers location of each partition.
- **KRaft (Kafka Raft):**
  - Apache Kafka Raft (KRaft) metadata management directly in Kafka without the requirement of a third-party tool like Apache ZooKeeper.
  - This greatly simplifies Kafka’s architecture by consolidating responsibility for metadata into Kafka itself, rather than splitting it between two different systems: ZooKeeper and Kafka.
  - KRaft mode improves partition scalability and resiliency.

- **Producers**:
  - They publish messages to a Kafka topic.

    - They need to specify the topic name and one broker to connect to and Kafka takes care of routing the data to the right brokers.
    - They are client applications that publish messages into topics
    - They decide which partition to send each message to: round-robin in case of messages without keys, use hashing, or custom scheme.



- **Consumers**:
    - Consumers read data from brokers.
    - They do not destroy message after reads.
    - It is a client application
    - Maintains ordering within partitions.



- **Note:**

- Decoupling Producers and Consumers: slow producers do not affect fast producers
- Dynamic architecture:
  - Add producers with affecting consumers
  - Failure or removal of consumers does not affect the system

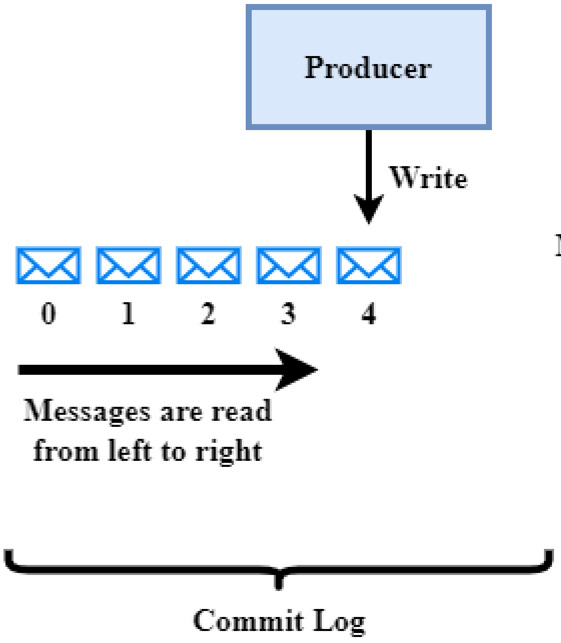
- **Kafka Data Model**
  - Kafka data organization:
    - Messages
    - Topics
    - Partitions
    - Offset
  - **Messages**:
    - Message also called Records are the basic unit of data in Kafka.
    - A message is usually a line of text such as a database row, or a like on a CSV file

EmpId	Lname	Fname		
1001	Mary	Allen	→	Message 1
1002	Jon	Paul	→	Message 2
1003	Kate	Miller	→	Message 3
1004	Ali	Mo	→	Message 3

- Messages are immutable.
- They can only append.
- A message structure:
  - Key:
    - Can be used to direct messages to specific partitions.
    - It can be null if you are included in a message.

- Value (Your message): For Kafka, it is just a sequence of bytes.
- Timestamp
- Metadata such offset, offset, timestamp, compression type, and etc.

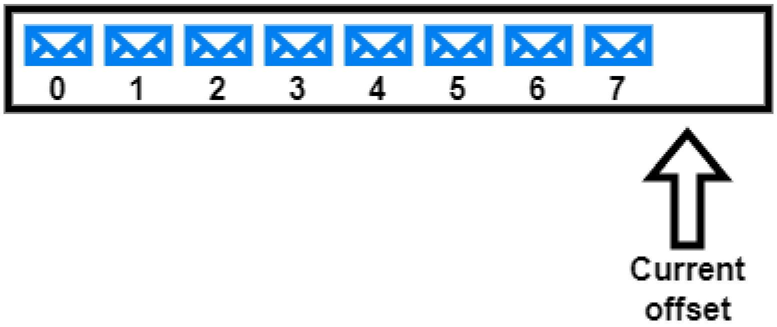
Key	Value
Compression Type (None, gzip,etc.)	
Partition-Offset	
Headers (Optional additional metadata)	
Key	Value
…	…
Timestamp (system or user set)	


Producer
Write
New Message are always inserted at the end
0
1
2
3
4
Messages are read from left to right
Commit Log

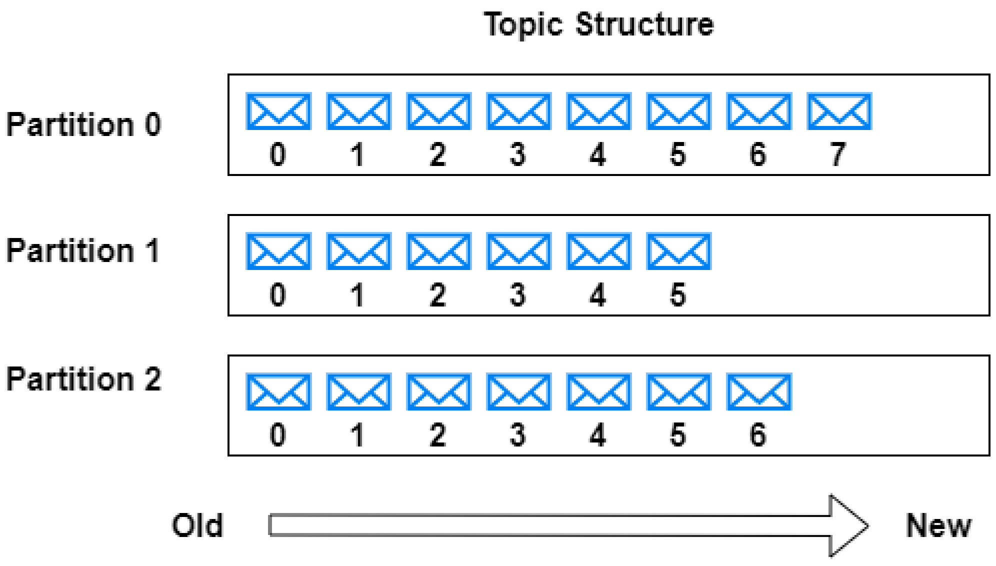
- **Topics:**
  - Messages are organized into logical grouping called topics.
  - A topic is an ordered sequence of events, also called an event log.
  - Producers publish messaged into topics.

- Consumers read messages from topics
- Messages are added at one end of the topics
- Topics are split into partitions, which are replicated.
- When you create a topic, you specify the amount of partitions it has.

- **Partitions:**
  - Topics are split and distributed by partitions for speed and size.
  - Messages in a partition are ordered and each message gets an in ID called offset



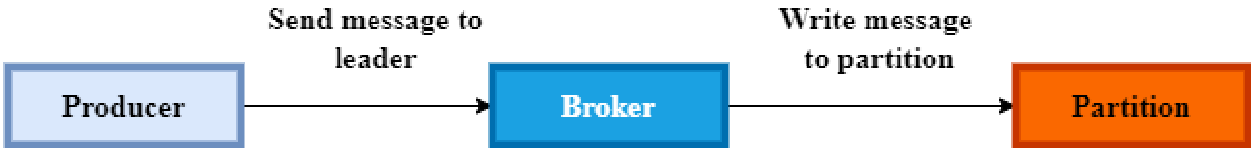
  - Kafka breaks topics up into partitions. A message is stored on a partition usually by message key if the key is present and **round-robin** if the key is missing (default behavior).
  - The record key, by default, determines which partition a producer sends the record.
  - Also, Kafka also uses partitions to facilitate parallel consumers. Consumers read messages in parallel up to the number of partitions.
  - A topic with a single partition can only reside on a single broker.
  - Partitions of a topic can reside on a single broker.
  - Partitions of a topic can be distributed over multi broker cluster



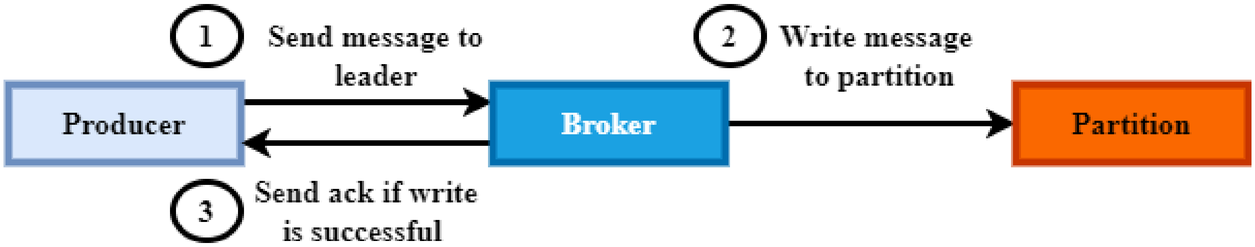
- Every partition is replicated over multiple servers.
- Every partition has one server acting as a leader and the rest as followers.
- The leader handles all read/write requests for the partition. The followers replicate the leader.
- If the leader server fails, one of the followers become a leader.
- **In-Sync Replicas** (ISR):
  - It is the number of replicated partitions that are in sync with its leader
  - The followers have the same messages (or in sync) as the leader.
  - It's not mandatory to have ISR equal to the number of replicas.
- If the leader server fails, one of the ISR become a leader.

- **How are messages processed by Kafka Producers?**
  - A producer needs a confirmation from the broker that the message has been received successfully and stored by the broker.
  - It is set in the broker configuration for the producer.
  - This is achieved by the parameter acks:
    - Acks=0

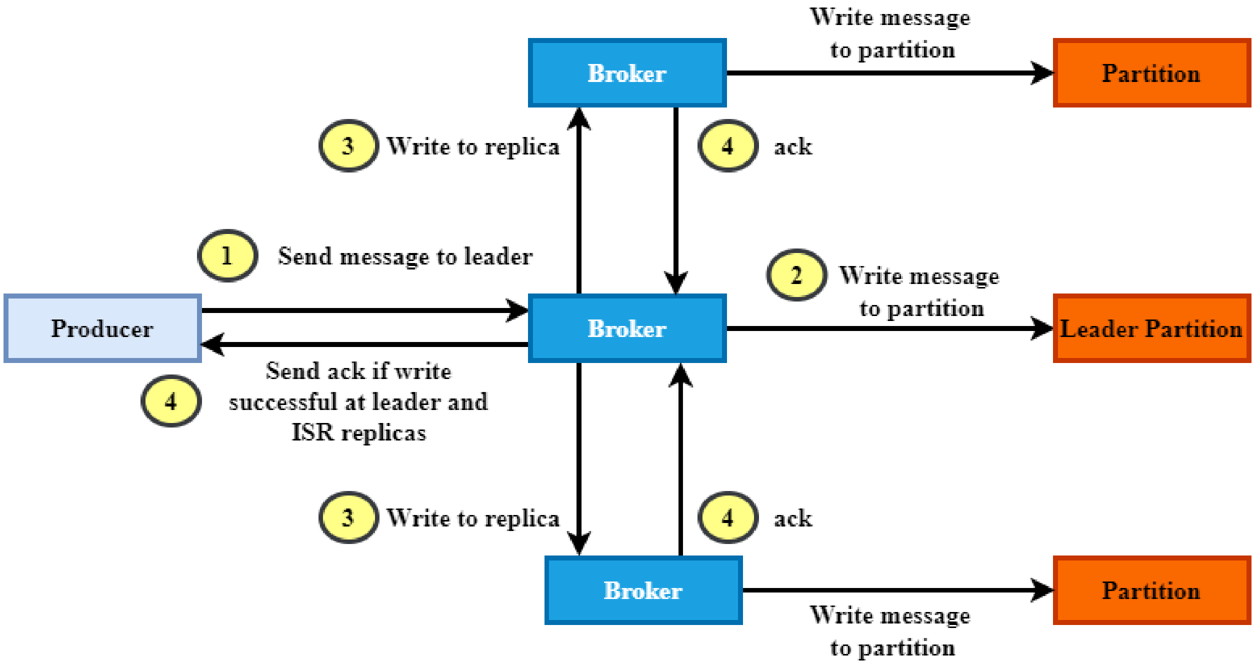
        - Acks=1
        - Acks=all (or acks=-1)
    - Acks=0
        - The producer does not wait for any acknowledgement from the broker.
        - There is no guarantee that the broker has received the message.
        - The producer will never re-send the message in case of failure.
        - This mode has high performance: lower latency and high throughput at the expense of message delivery.



    - Acks=1
        - The producer gets an ack after the leader has received the message.
        - If the producer does not receive the ack, it will retry.
        - Possible data loss: There is no guarantee that the message has been replicated.
        - After writing the message, the leader will respond without awaiting a full acknowledgment from all followers.
        - Performance better than ack =0.



- Acks=all
  - Acks =all is the same as acks=-1.
  - In this case the producer gets an ack when all ISR replicas have received the message.
  - The leader acknowledges the message only when it receives acks from all ISR replicas.
  - No data loss as long as one of the ISR replicas is alive.
  - Performance: higher latency but better safety.



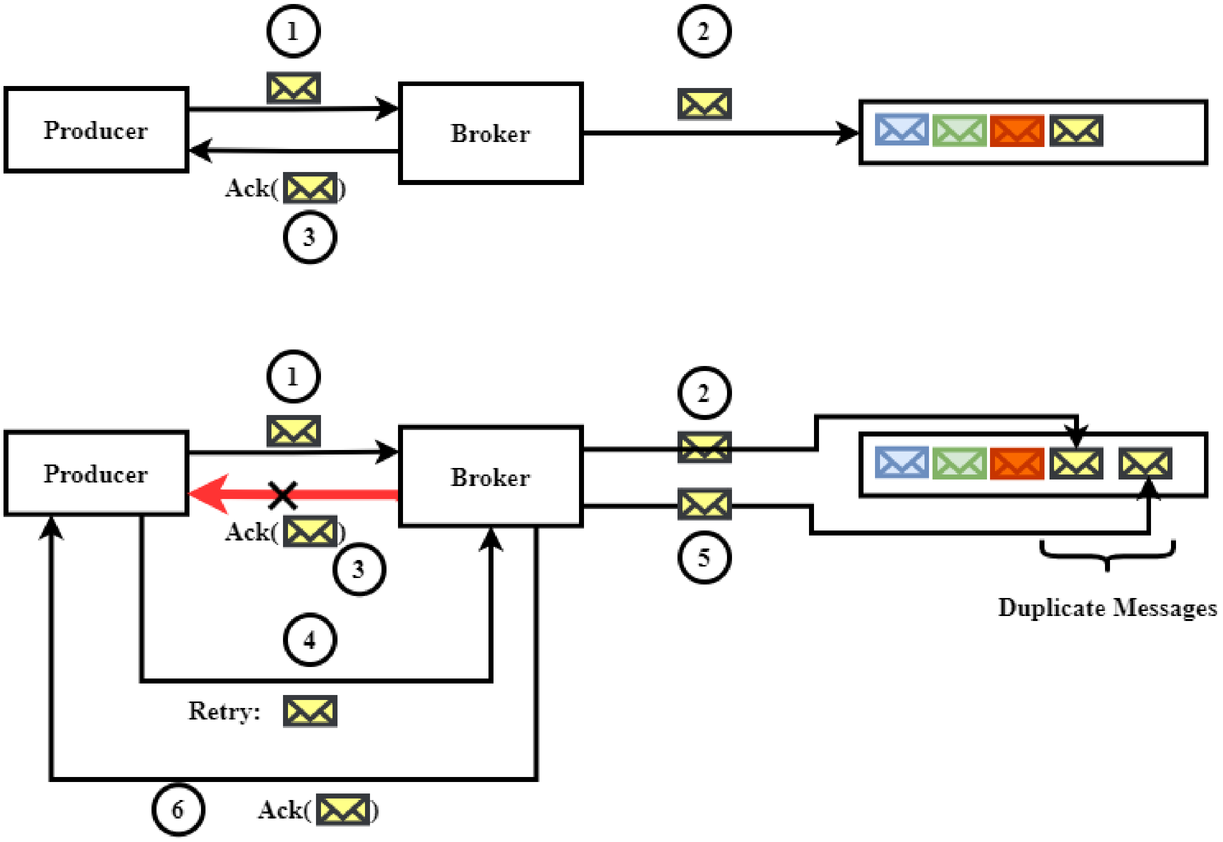
Write message to partition
Broker
Partition
3 Write to replica
4 ack
1 Send message to leader
2 Write message to partition
Producer
Broker
Leader Partition
4 Send ack if write successful at leader and ISR replicas
3 Write to replica
4 ack
Broker
Partition
Write message to partition

- **Min.insync.replicas:**
  - This parameter specifies the minimum number of replicas that must acknowledge a write for a message to be successful.
  - Example:

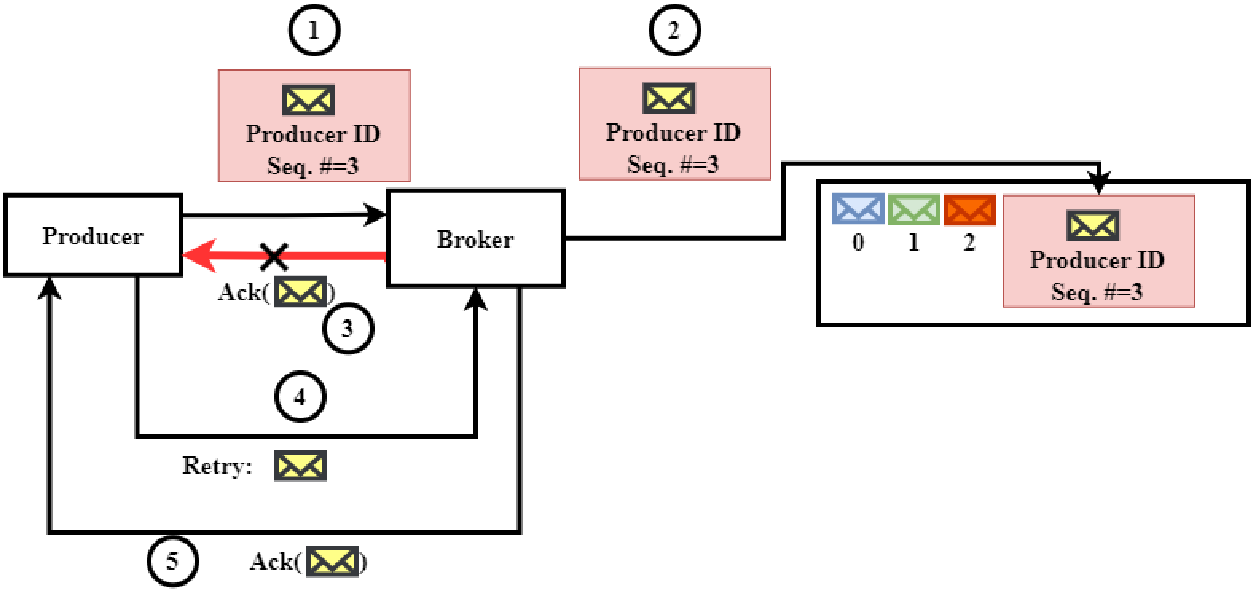
- **retries:**
  - It defines the number of times a producer will attempt to send a message before marking it as failed. The default value is 0.
  - Another related parameter is retry.backoff.ms.
  - It sets the duration between two retry.

        - retry.backoff.ms default’s value is 100 ms.

- **Idempotent Producers:**
    - Producer idempotence is used to prevent publishing a message twice due to an expected retires.
    - Retires may occur due to network issues that prevent the Acks (broker acknowledgment) from reaching the producer.



    - An Idempotent producer is a Kafka producer that writes a message to a topic **EXCATLY** once.



- How to make a producer idempotent?
  - Set the producer parameter:

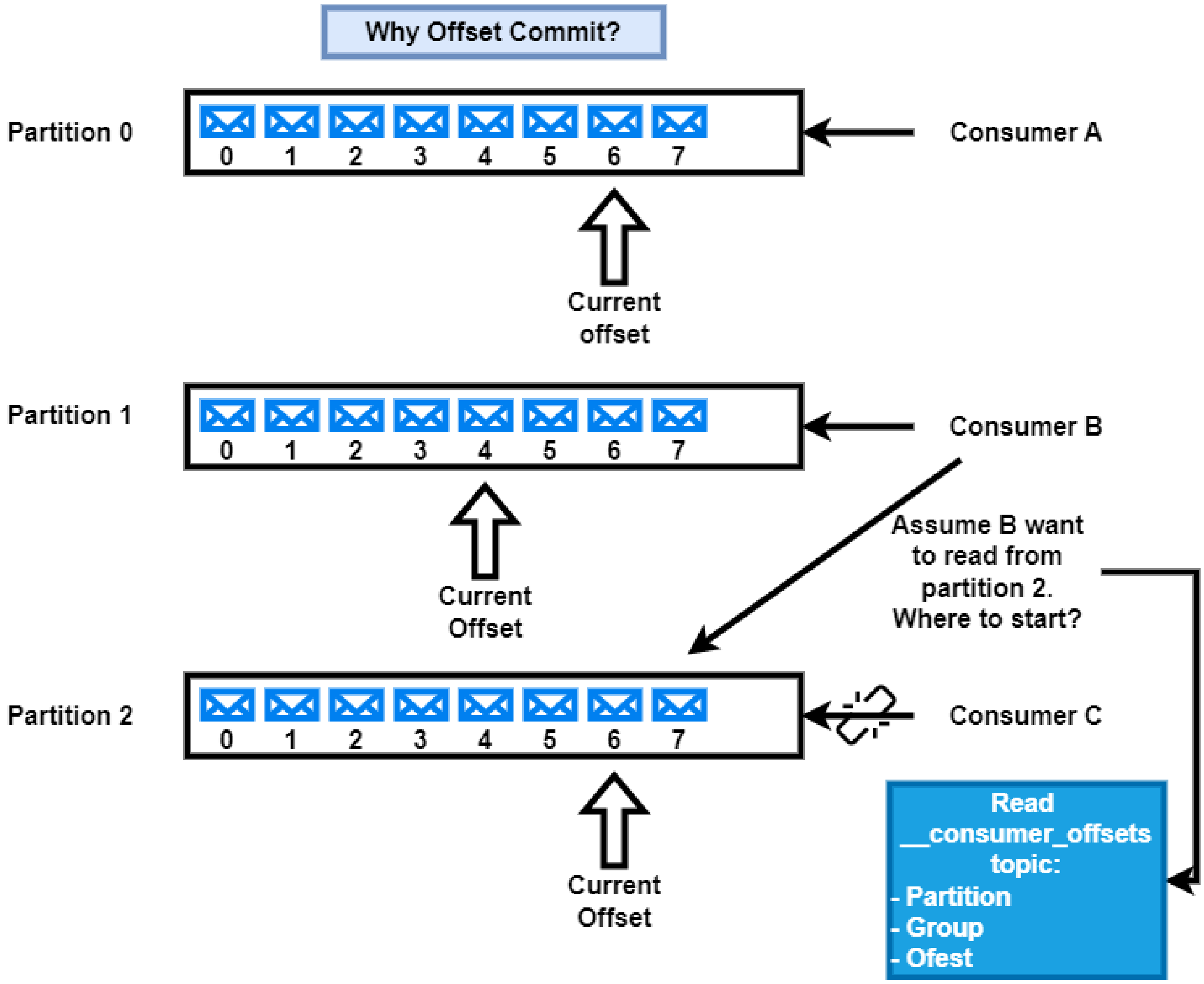
    enable.idempotence = true

  - This will ensure that a message is written exactly once in the designated topic.
  - Conditions to set enable.idempotence:
    - acks=all
    - retries > 0
    - max.in.flight.requests.per.connection <= 5

Note that **max.in.flight.requests.per.connection** is the maximum number of unacknowledged requests the client will send on a single connection before blocking.

- **Consuming Offsets**
    - Kafka does not keep track of what messages have been completely processed by consumers.
    - Offsets are unique within each partition
    - There are two types of offsets:
        - Producer offset:
            - The current position of new messages
        - Consumer offset:
            - It is used to prevent re-processing of messages by a group of consumers
            - There are two types of offsets:
                - Current offset
                - Commit Offset
        - Current offset (Position):
            - It is the offset from which next new message will be fetched (when it's available).
            - It is stored in a special topic:

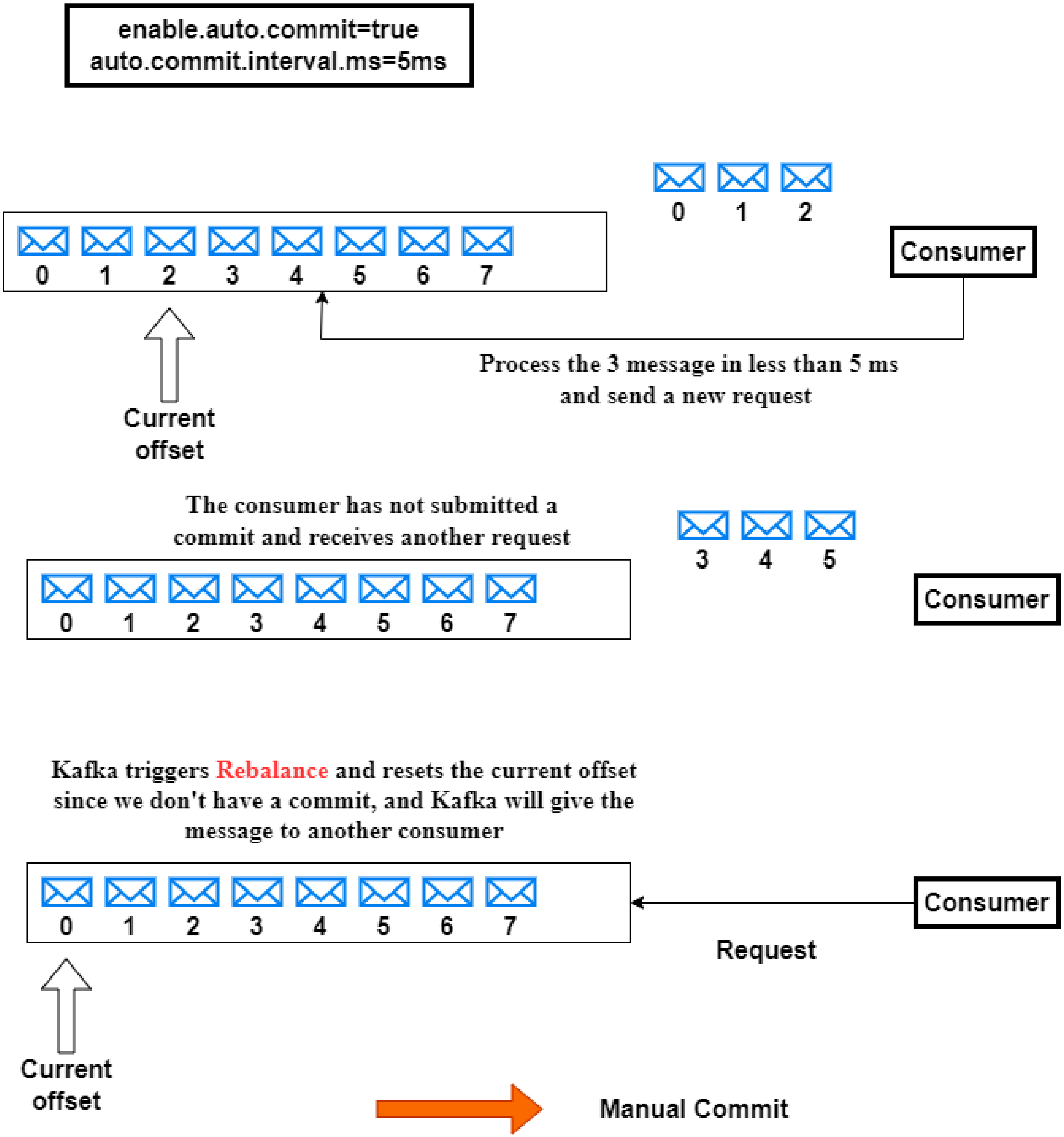
        - committed Offset:
            - To keep track of the last message processed by the consumer so Kafka cluster won't send the committed records for the same partition to another consumer of the same consumer group.
            - It is also used in case a second consumer is trying to read from the same partition – it should reprocess the same messages.
            - Committed offset is important in case of a consumer recovery or rebalancing.



- There are two types of commit offsets:
  - Auto commit:
    - When messages are read, the commit offset is set to the offset of the last message read.
    - The auto commit is handled by wo variables:
      - enable.auto.commit
      - auto.commit.interval.ms
    - The Auto-commit is enabled true by default:

      enable.auto.commit =true
    - The consumer's offset will be periodically committed in the background.

- For a consumer, the property auto.commit.interval.ms, specifies the frequency in milliseconds that the consumer offsets are auto-committed to Kafka if enable.auto.commit is set to true.
- The auto.commit.interval.ms defines the interval of auto commit.
- Default is 5ms



Process the 3 message in less than 5 ms and send a new request

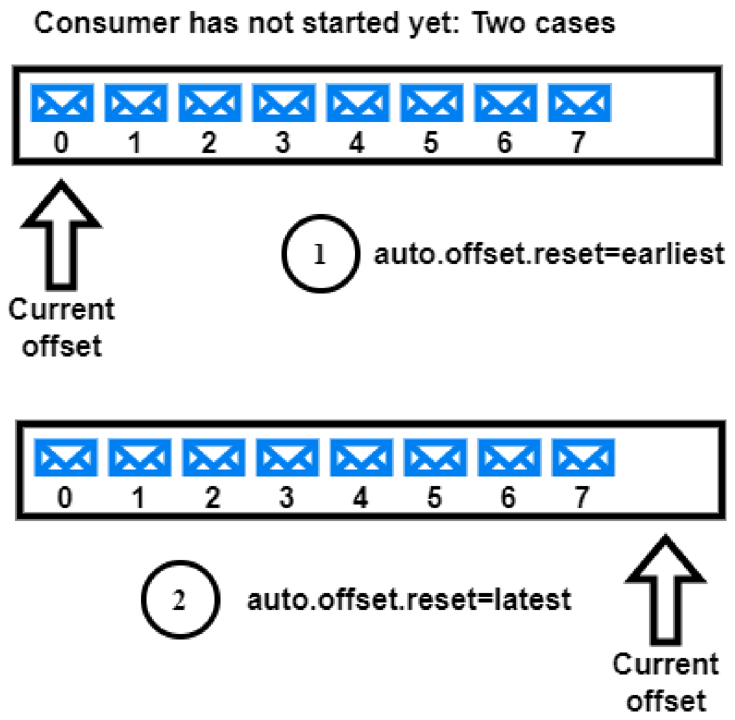
The consumer has not submitted a commit and receives another request

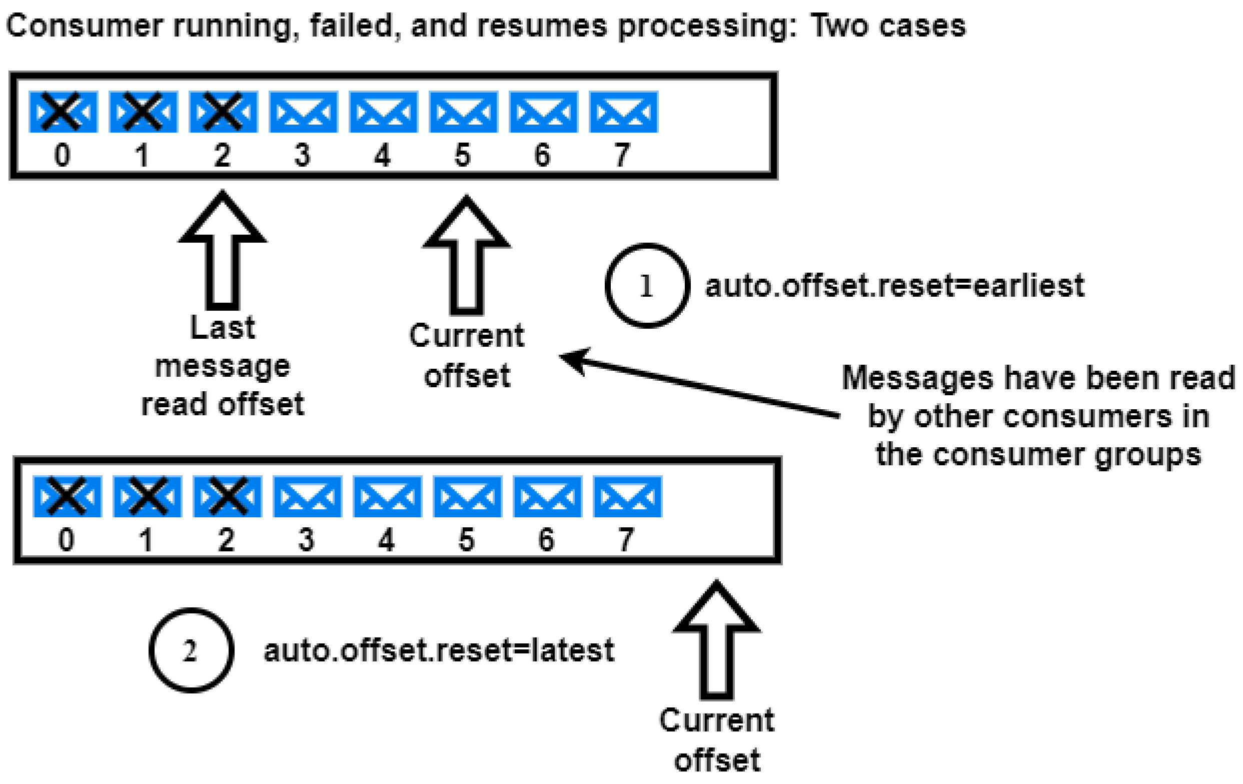
Kafka triggers Rebalance and resets the current offset since we don't have a commit, and Kafka will give the message to another consumer

- Manual Offset:
  - The enable.auto.commit = false
  - Sync commit:
    - Every time we read a message; the consumer will not read the next

        message unless it hears back from the topic offset.
    - This makes the processing slower.
  - Async commit
    - When the read message is consumer, the commit offset is set automatically.
    - The consumer does not have to wait for an acknowledgement.

- **Kafka Consumer Auto Offset Reset:**
  - In case the committed offset is not available, we can auto.offset.reset parameter.
  - There are three modes:
    - Read from the end of the partition: auto.offset.reset = latest
    - Read from the start of the partition: auto.offset.reset = earliest:
    - Throw and exception if no offset is found auto.offset.reset = latest




Consumer running, failed, and resumes processing: Two cases
0
1
2
3
4
5
6
7
Last message read offset
Current offset
1
auto.offset.reset=earliest
Messages have been read by other consumers in the consumer groups
0
1
2
3
4
5
6
7
2
auto.offset.reset=latest
Current offset

- **Consumer offset Retention**:
  - Once a Kafka consumer starts consuming data from a topic, it commits its last consumed message's offset in the Kafka broker's internal topic called __consumer_offsets.
  - This topic helps a consumer in identifying the offset from which it should start reading the topic on its next poll.
  - offset.retention.minutes parameter sets the retention of the committed offset of a consumer.
  - The consumer's committed offset is reset once the retention expires:
    - The consumer can either decide to read all data from the topic or the latest data from the topic based on the consumer config auto.offset.reset
  - If a consumer has not read new data in one day (Kafka < 2.0)

        - If a consumer has not read new data in 7 days (Kafka >=2.0)
        - Use retention parameter:
            - Offset.retention.minutes

- **Producer/Consumer Python Example**
    - Producer: (analyticshut.com)

      ```
      from kafka import KafkaProducer
      bootstrap_servers = ['localhost:9092']
      topicName = 'myTopic'
      producer = KafkaProducer(bootstrap_servers = bootstrap_servers)
      producer = KafkaProducer()
      ```

      We can start sending messages to this topic using the following code.

      ```
      ack = producer.send(topicName, b'Hello World!!!!!!!!')
      metadata = ack.get()
      print(metadata.topic)
      print(metadata.partition)
      ```

    - Consumer Example:

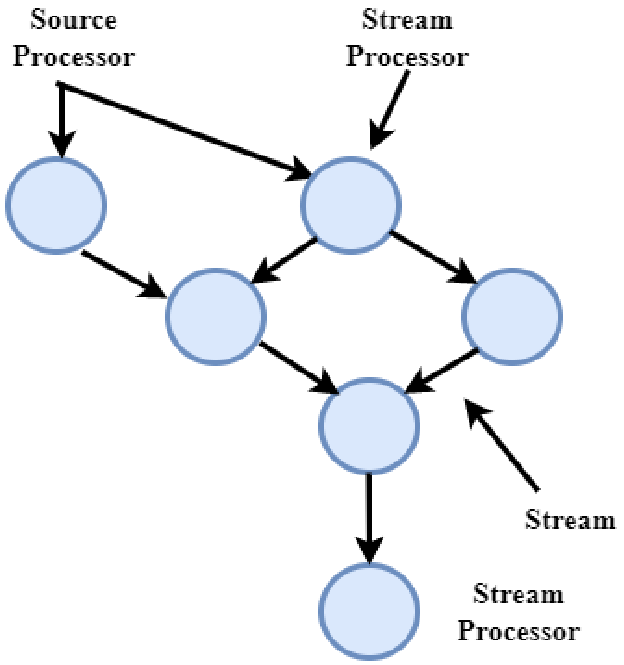
      ```
      from kafka import KafkaConsumer
      import sys
      bootstrap_servers = ['localhost:9092']
      topicName = 'myTopic'
      consumer = KafkaConsumer (topicName, group_id =
      'group1',bootstrap_servers = bootstrap_servers,
      auto_offset_reset = 'earliest')
      ```

      Notes:
      - auto_offset_reset = 'earliest' ➔ read messages from the beginning of the topic.
      Now we can start reading message from the topic.

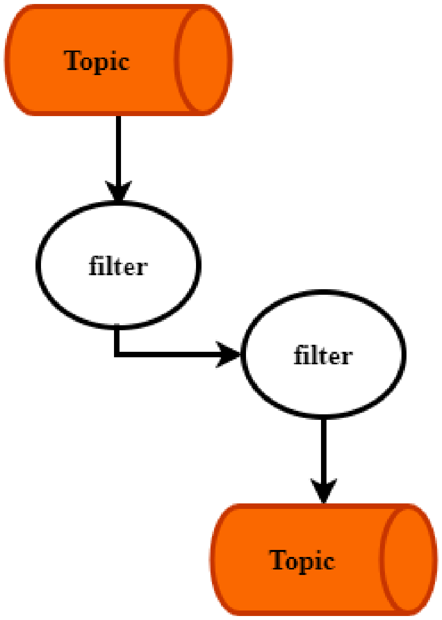
```
try:
# we are reading the message, its key, offset, and partion.
    for message in consumer:
        print ("%s:%d:%d: key=%s value=%s" % (message.topic,
message.partition,message.offset, message.key,message.value))
except KeyboardInterrupt:
    sys.exit()
```

- **Kafka Stream API**
    - Kafka Stream:
        - It is like Kafka topic
        - It is n unbound and continuous flow of data packets in real time. Packets are generated in the form of key-value pairs.
    - Kafka stream is an open-source client library used to build applications and micro-services.
    - Enables to consumer from Kafka topics, perform analytical or transformation work on data, and send to other topics
    - Examples:
        - Sensor data
        - Click streams
        - Transactions
        - Log entries
        - Etc.
    - It processes one message at a time and guarantees that each message is processed once and only one.
    - A Kafka stream reads from a Kaka topic and writes to a Kafka stream.
    - **Kafka Stream Processing Topology:**
        - It is a logical representation of the Kafka stream application.
        - It is a set of processor nodes where each node represents a transformation step in the application.

- Source Processor:
    - A stream processor that does not have any up stream processor
- Sink Processor:
    - A stream processor that does not have any down stream processor

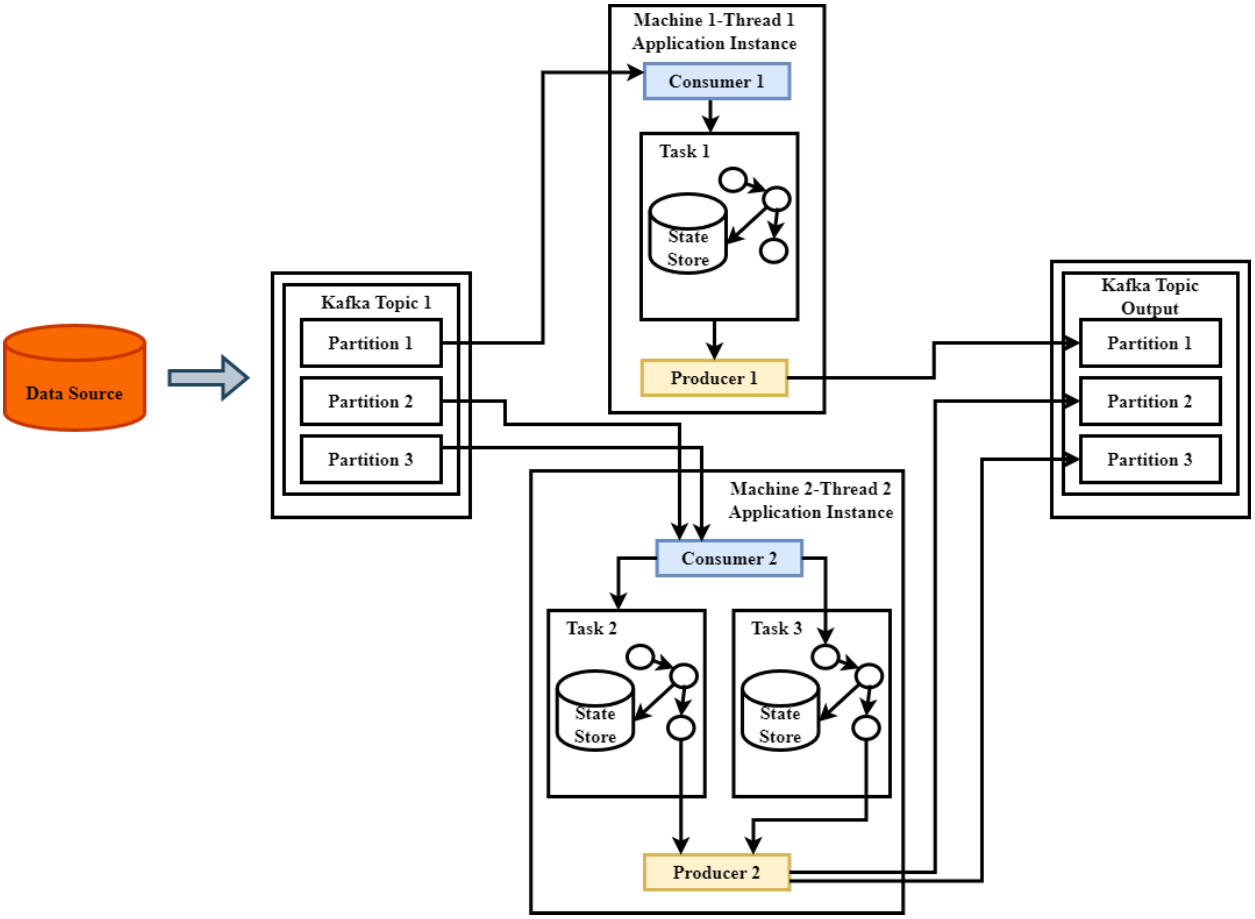


- Example

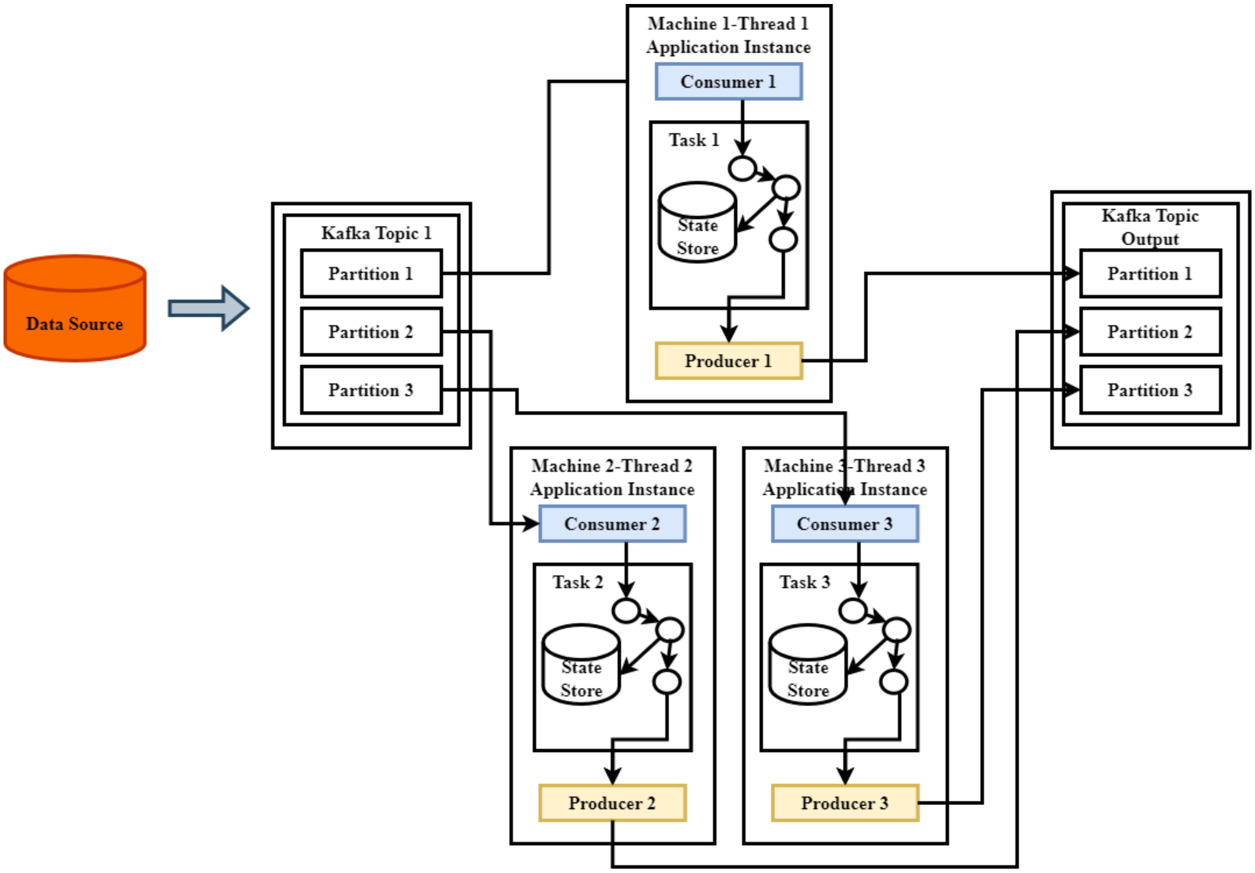


- **Kafka Stream DSL (Domain Specific Language)**
    - It supports declarative, functional programming style with stateless and stateful transformations.
    - They are the Kafka stream operations
    - Stateless Transformations:
        - map, mapValues, filter
    - Stateful Transformations:
        - Aggregation (count, reduce), joins, windowing
    - Build-in Abstractions:
        - StreamBuilder, Kstream, KTable, GlobalKTable

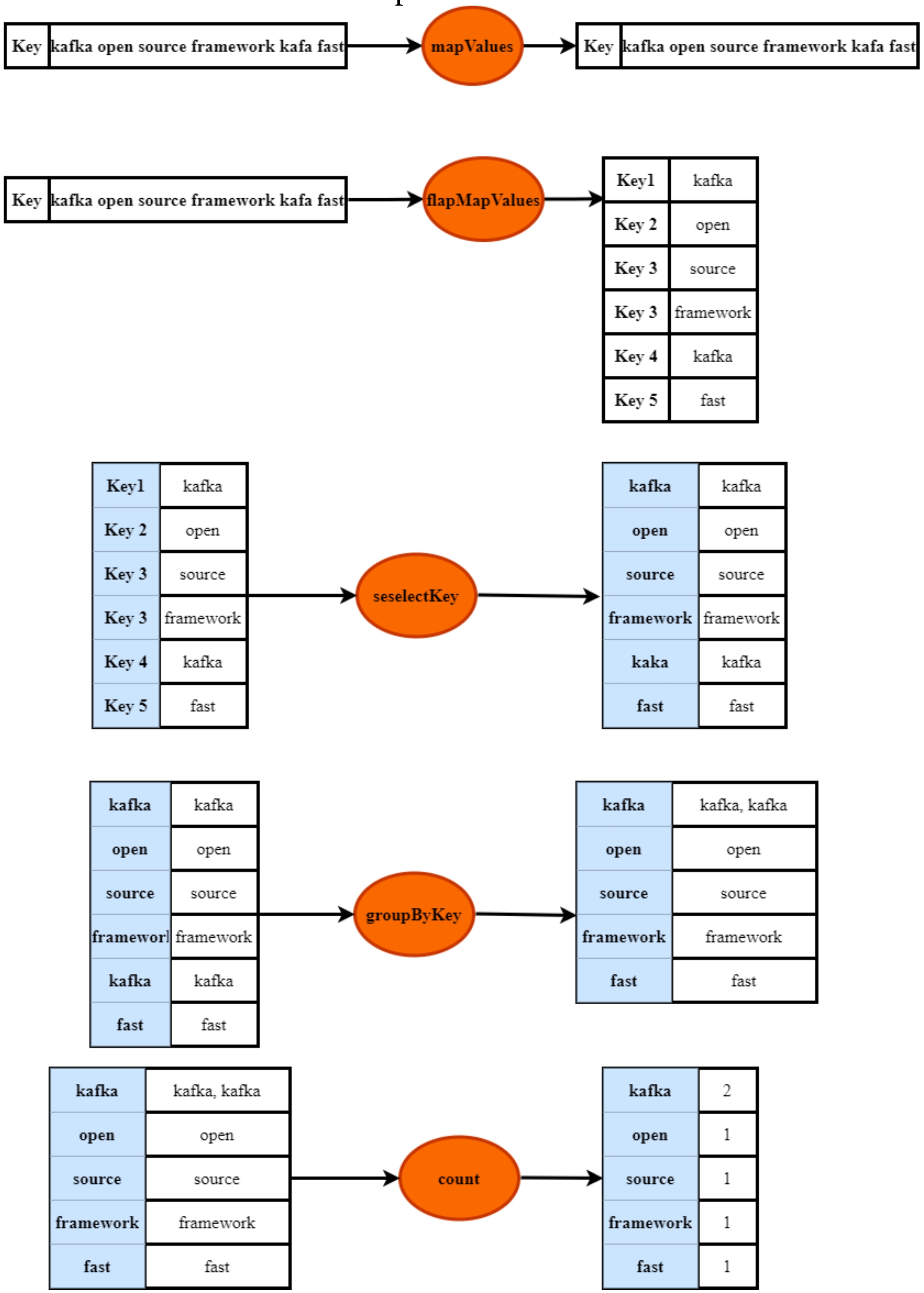
- Kafka Stream Architecture:
    - When we start a stream application, Kafka framework creates a number of tasks equals to the number of partitions.



- Rebalancing:
    - Happens when you scale your application by creating a new thread on a different machine.
    - Kafka automatically moves one task from machine 2 to machine 3.
    - If you have three partitions, you can only create three tasks. If you create a new thread, it will remain idle.



- **Kafka Application**
    - Word count example



Key	kafka open source framework kafa fast

Key	kafka open source framework kafa fast

Key	kafka open source framework kafa fast

Key1	kafka
Key 2	open
Key 3	source
Key 3	framework
Key 4	kafka
Key 5	fast

Key1	kafka
Key 2	open
Key 3	source
Key 3	framework
Key 4	kafka
Key 5	fast

kafka	kafka
open	open
source	source
framework	framework
kaka	kafka
fast	fast

kafka	kafka
open	open
source	source
framewor	framework
kafka	kafka
fast	fast

kafka	kafka, kafka
open	open
source	source
framework	framework
fast	fast

kafka	kafka, kafka
open	open
source	source
framework	framework
fast	fast

kafka	2
open	1
source	1
framework	1
fast	1