



# Introduction to Apache Kafka



# Table of Contents

Overview

Replication

Why kafka is so Fast!!

Why do need Kafka

Replication

Zookeeper Overview

What is Kafka

Demo!

Demo!

Where it is used in Zeta



# Overview

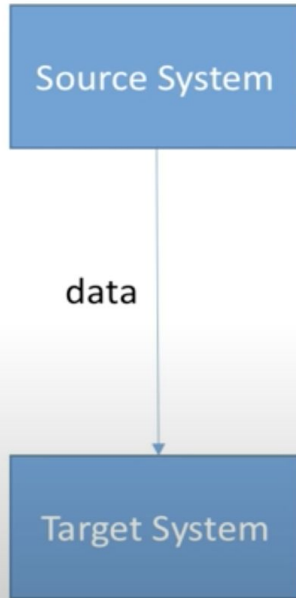
Apache Kafka is a realtime publish-subscribe based durable messaging system

- Process streams of records as they occur.
- Store streams of records in a fault-tolerant durable way.
- Maintains FIFO order for streams
- It is a high-throughput, highly distributed, fault-tolerant platform developed by linkedin and written in Scala and Java



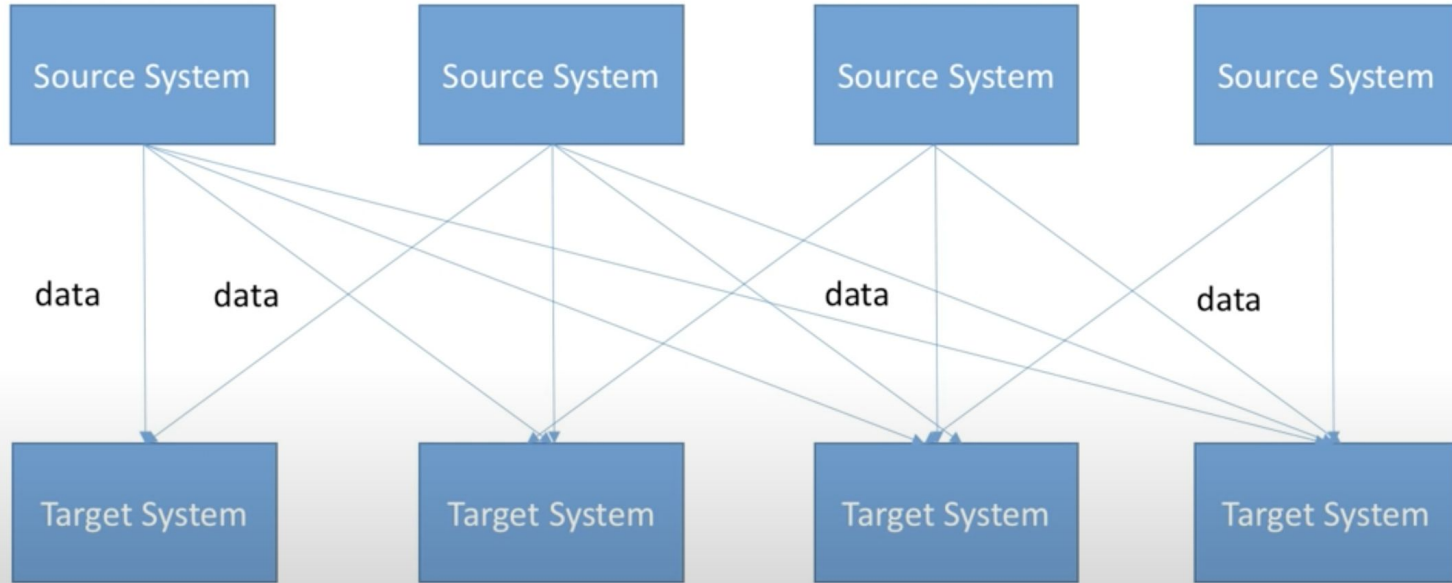
# Why do we need Kafka?

# How company starts



Simple at first!

# After a while..



Very complicated!



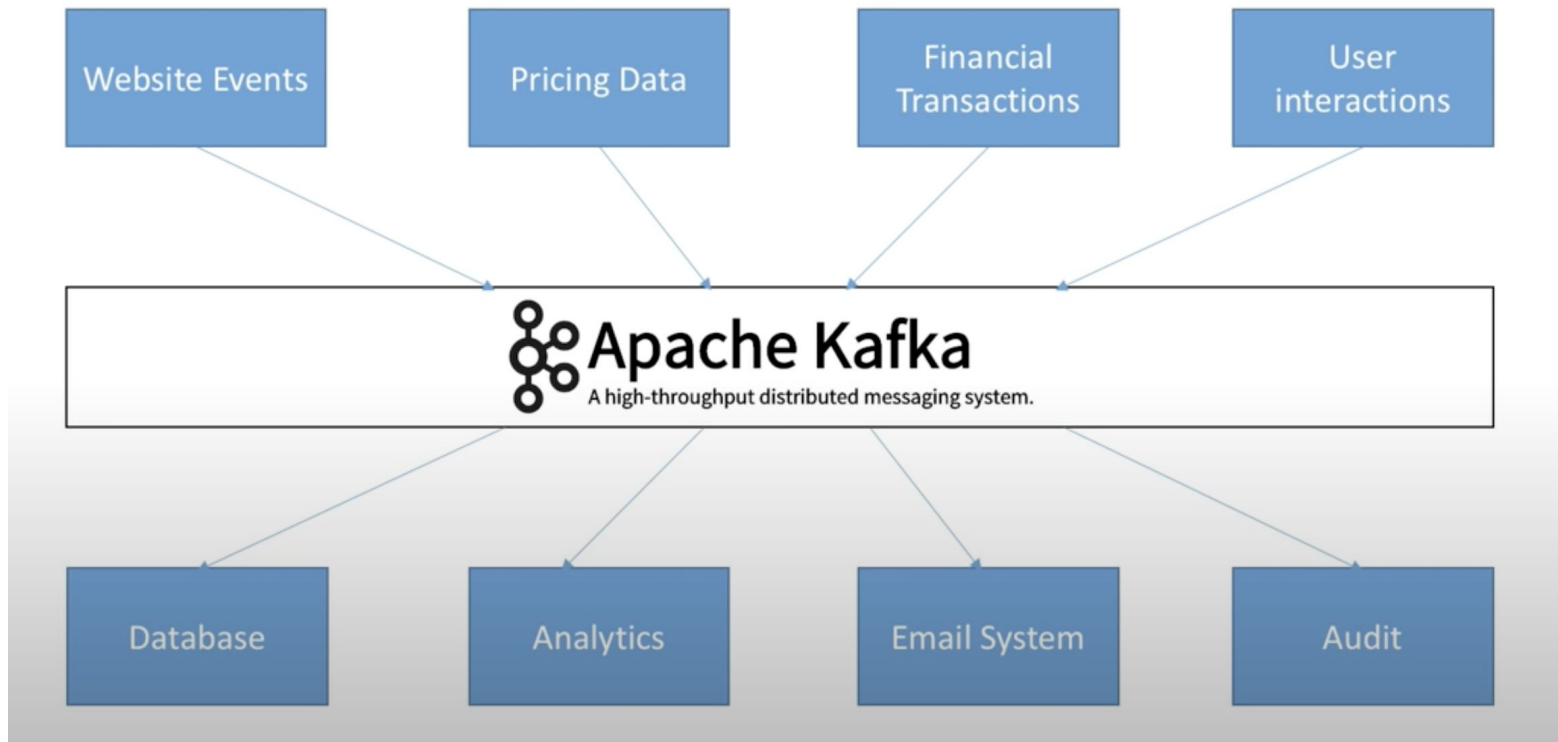


# Problems with previous architecture

- If you have 4 source systems, and 6 target systems, you need to write 24 integrations!
- Each integration comes with difficulties around
  - Protocol – how the data is transported (*TCP, HTTP, REST, FTP, JDBC...*)
  - Data format – how the data is parsed (*Binary, CSV, JSON, Avro...*)
  - Data schema & evolution – how the data is shaped and may change
- Each source system will have an increased load from the connections



# What we'd really like



# Why Apache Kafka

- Created by LinkedIn, now Open Source Project mainly maintained by Confluent
- Distributed, resilient architecture, fault tolerant
- Horizontal scalability:
  - Can scale to 100s of brokers
  - Can scale to millions of messages per second
- High performance (latency of less than 10ms) – real time
- Used by the 2000+ firms, 35% of the Fortune 500:



**NETFLIX**



**UBER**



# Kafka use cases

- **Netflix** uses Kafka to apply recommendations in real-time while you're watching TV shows
  - **Uber** uses Kafka to gather user, taxi and trip data in real-time to compute and forecast demand, and compute surge pricing in real-time
  - **LinkedIn** uses Kafka to prevent spam, collect user interactions to make better connection recommendations in real time.
- Remember that Kafka is only used as a transportation mechanism!



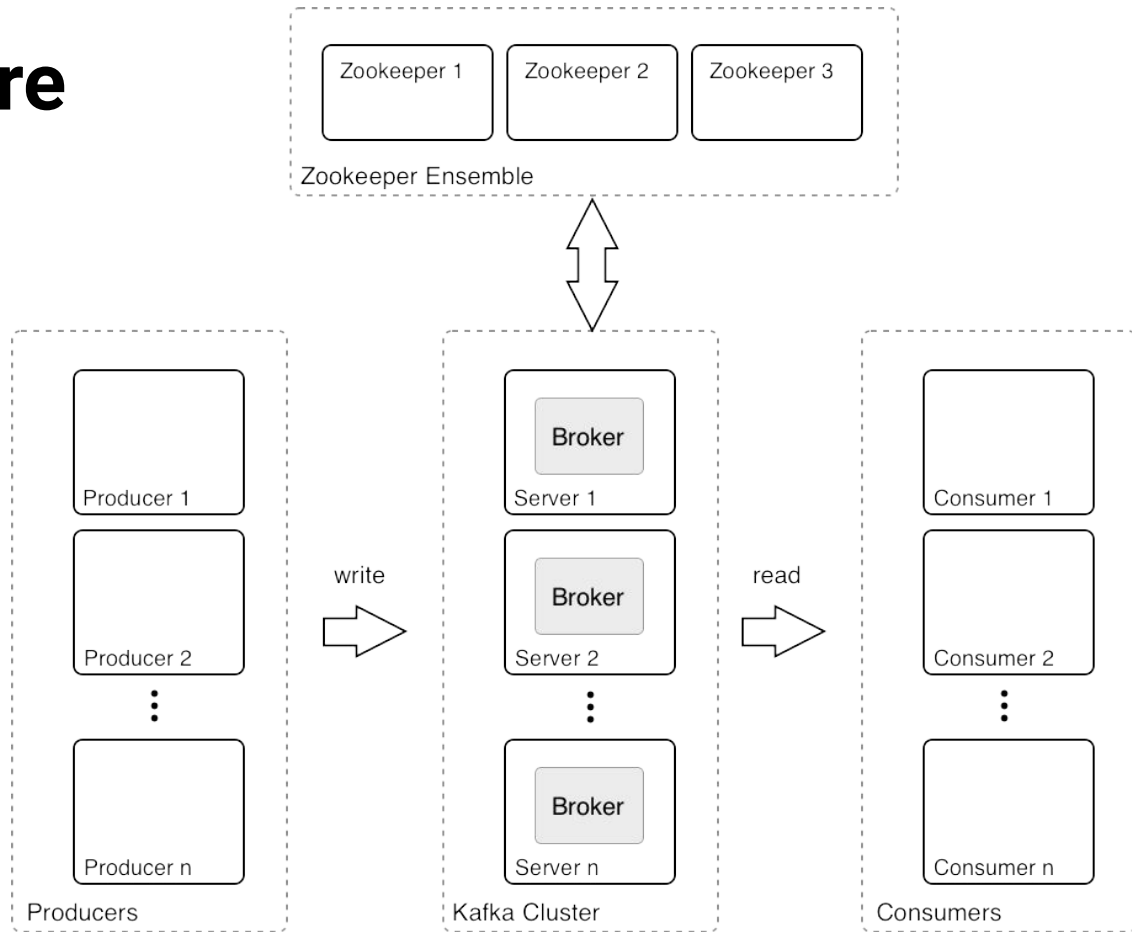
# How Kafka works?

# Kafka components

These are four main parts in a Kafka system:

- **Producer:** Sends records to a broker.
- **Broker:** Handles all requests from clients (produce, consume, and metadata) and keeps data replicated within the cluster. There can be one or more brokers in a cluster.
- **Consumer:** Consumes batches of records from the broker.
- **Zookeeper:** Keeps the state of the cluster (brokers, topics, users, offset).

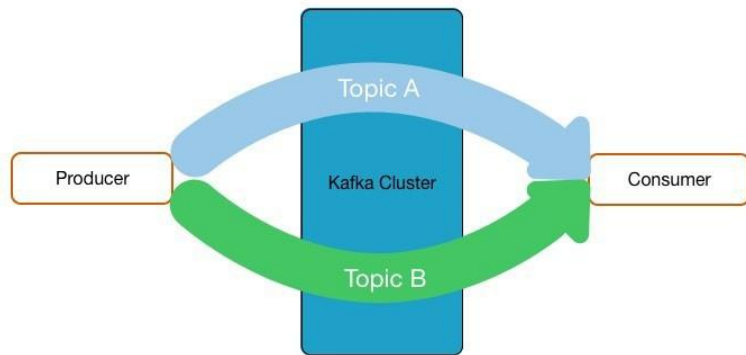
# Architecture



# Topics

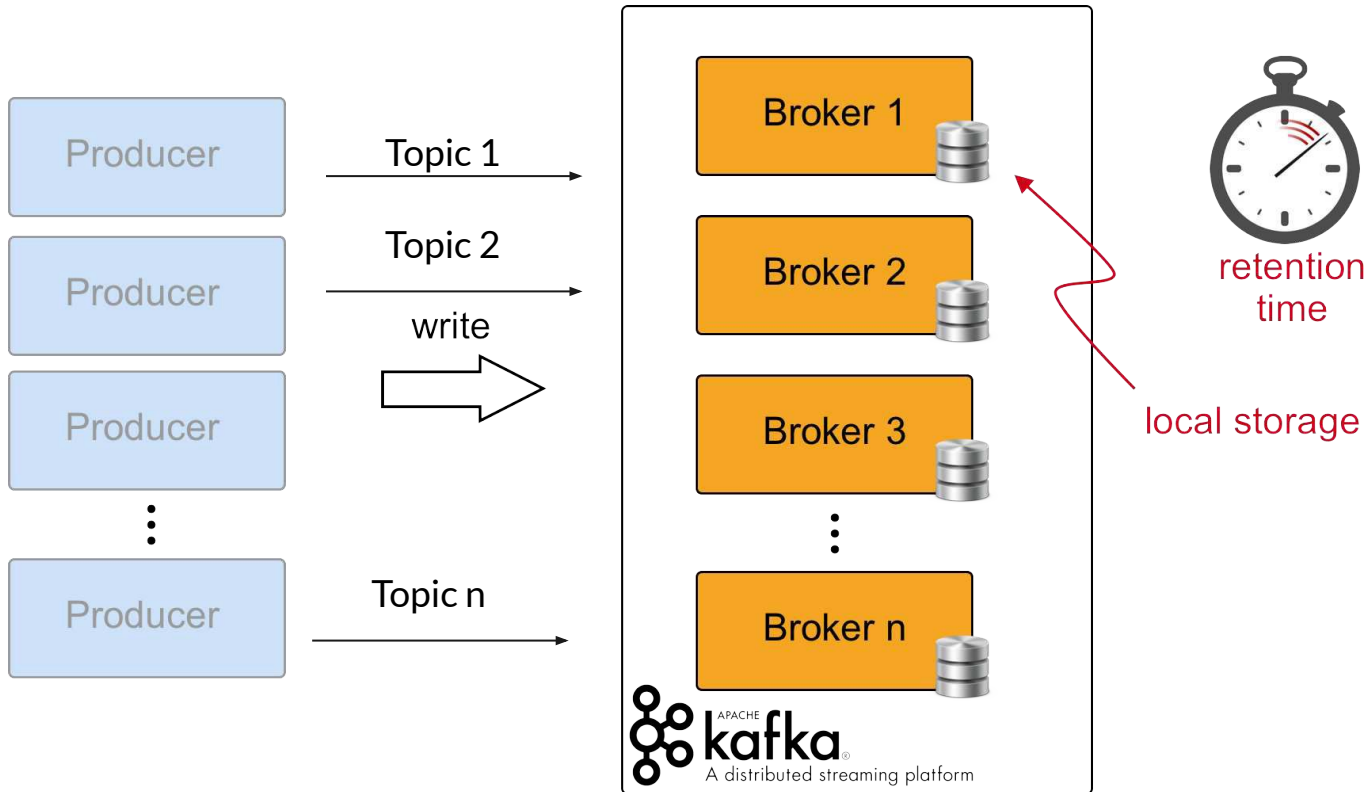
- Streams of “related” Messages in Kafka
  - Is a Logical Representation
  - Categorizes Messages into Groups
- Developers define Topics
- Producer write data to topics and consumer read from topics

Ex - User\_View, User\_Click, Order\_created, Product\_Updated,

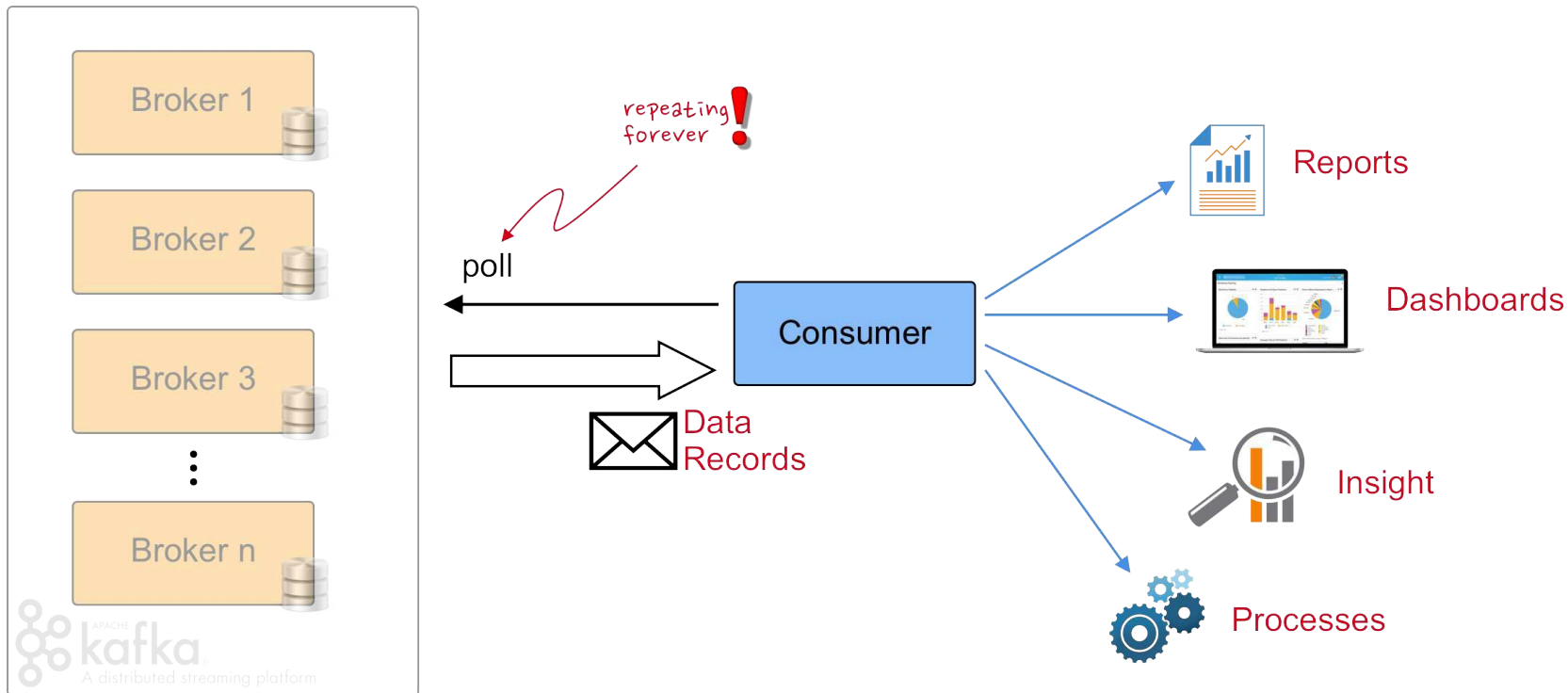




# Kafka Brokers



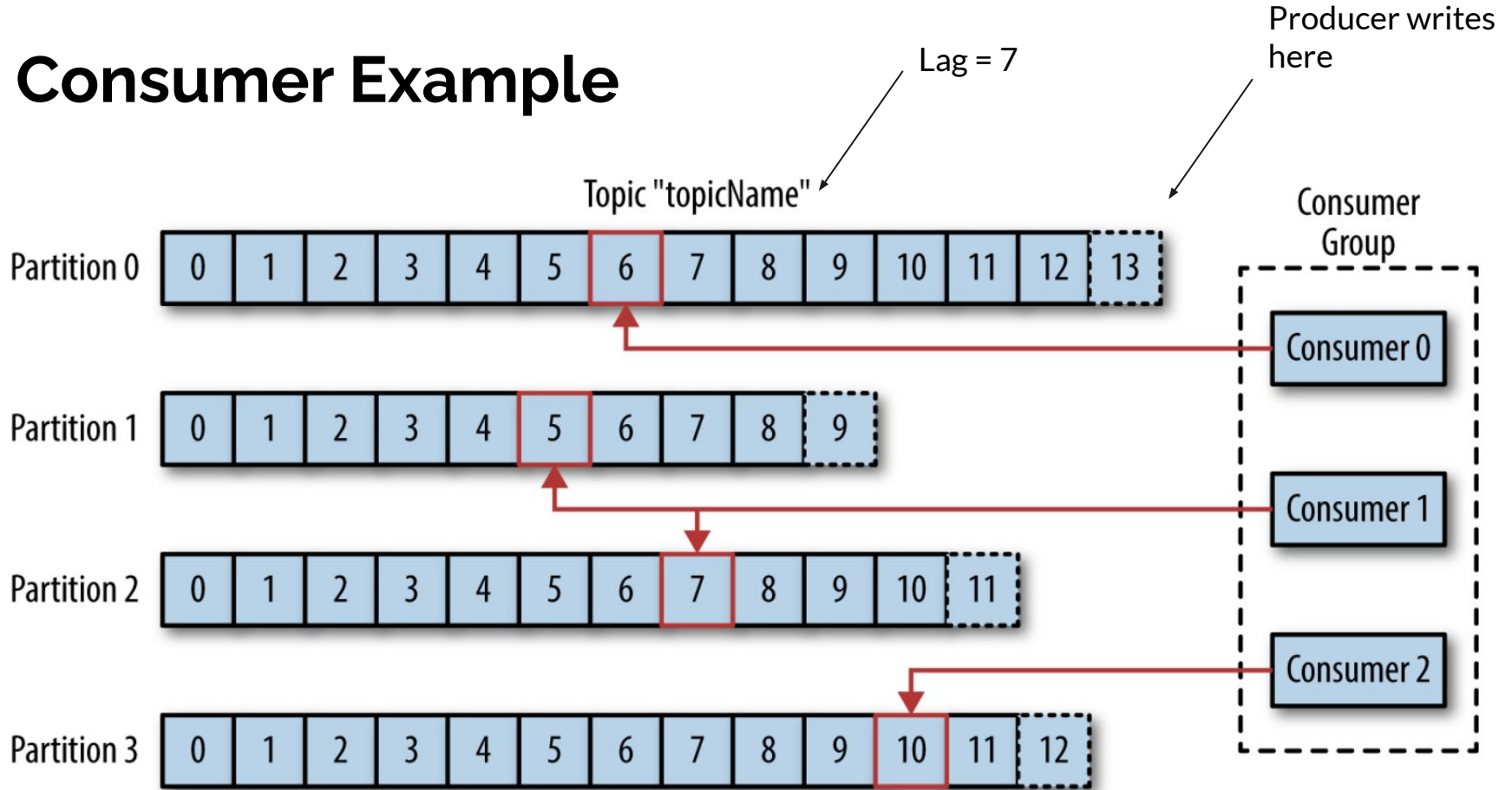
# Consumers



# Consumer Basics

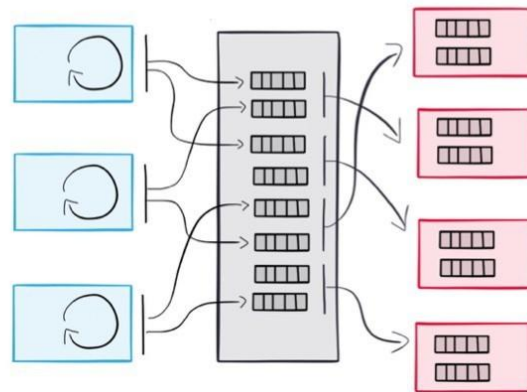
- Consumers **pull** messages from 1..n topics
- New inflowing messages are automatically retrieved
- Consumer will always read from leader
- Consumer committed offset - Keeps track of the last message committed
- Consumer current offset - Keeps track of the last message read

# Consumer Example



# Decoupling Producers and Consumers

- Producers and Consumers are decoupled
- Slow Consumers do not affect Producers
- Add Consumers without affecting Producers
- Failure of Consumer does not affect System





# How do you scale a topic?

# Partitions

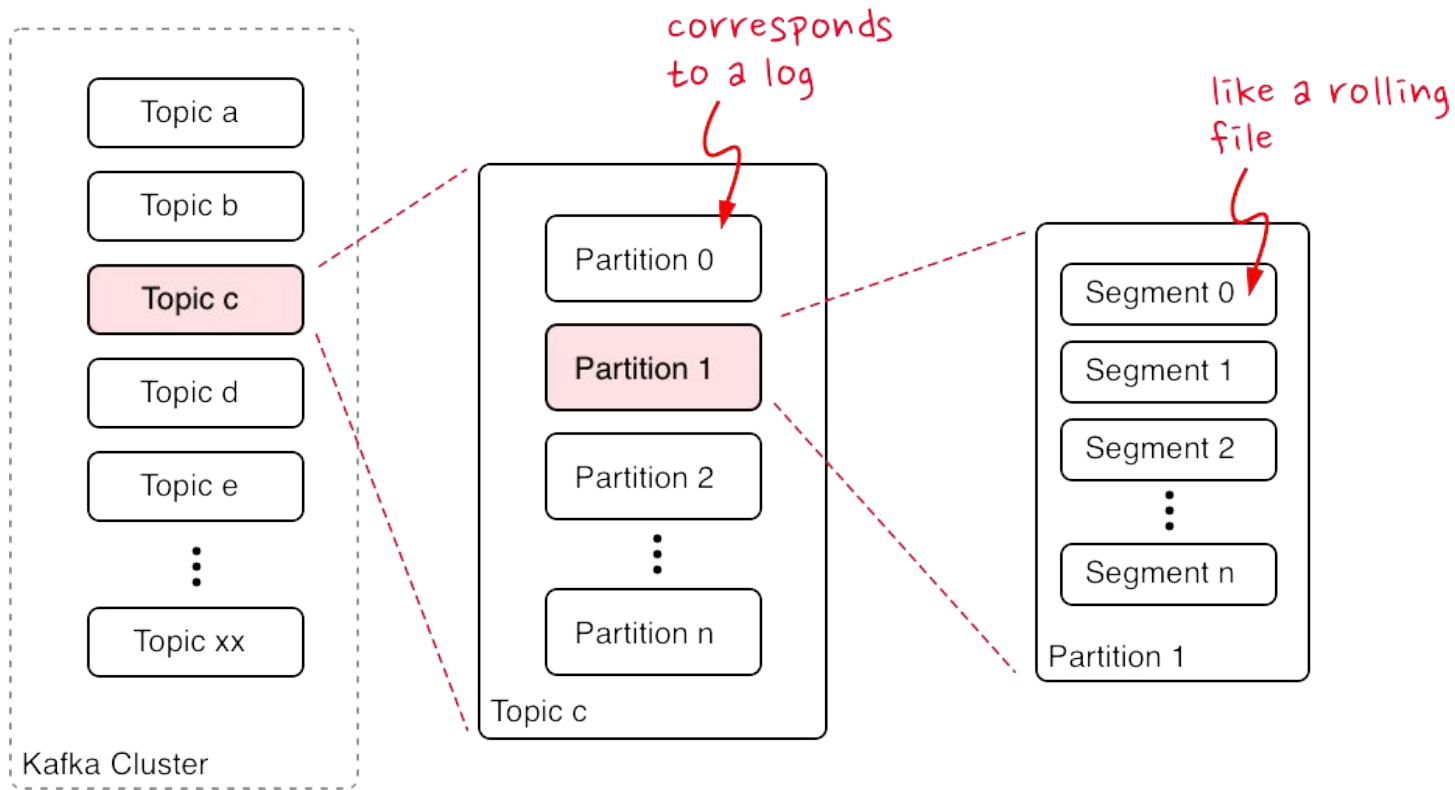
- To scale a particular topic, topics are further broken up into several partitions, usually by record key if the key is present.
- Each partition can run its own consumer in a single consumer group
- Each partition consumer combination will maintain its own offset
- Kafka can replicate partitions to multiple Kafka Brokers.
- Partitioning Strategy specified by Producer

Default Strategy: `hash(key) % number_of_partitions`

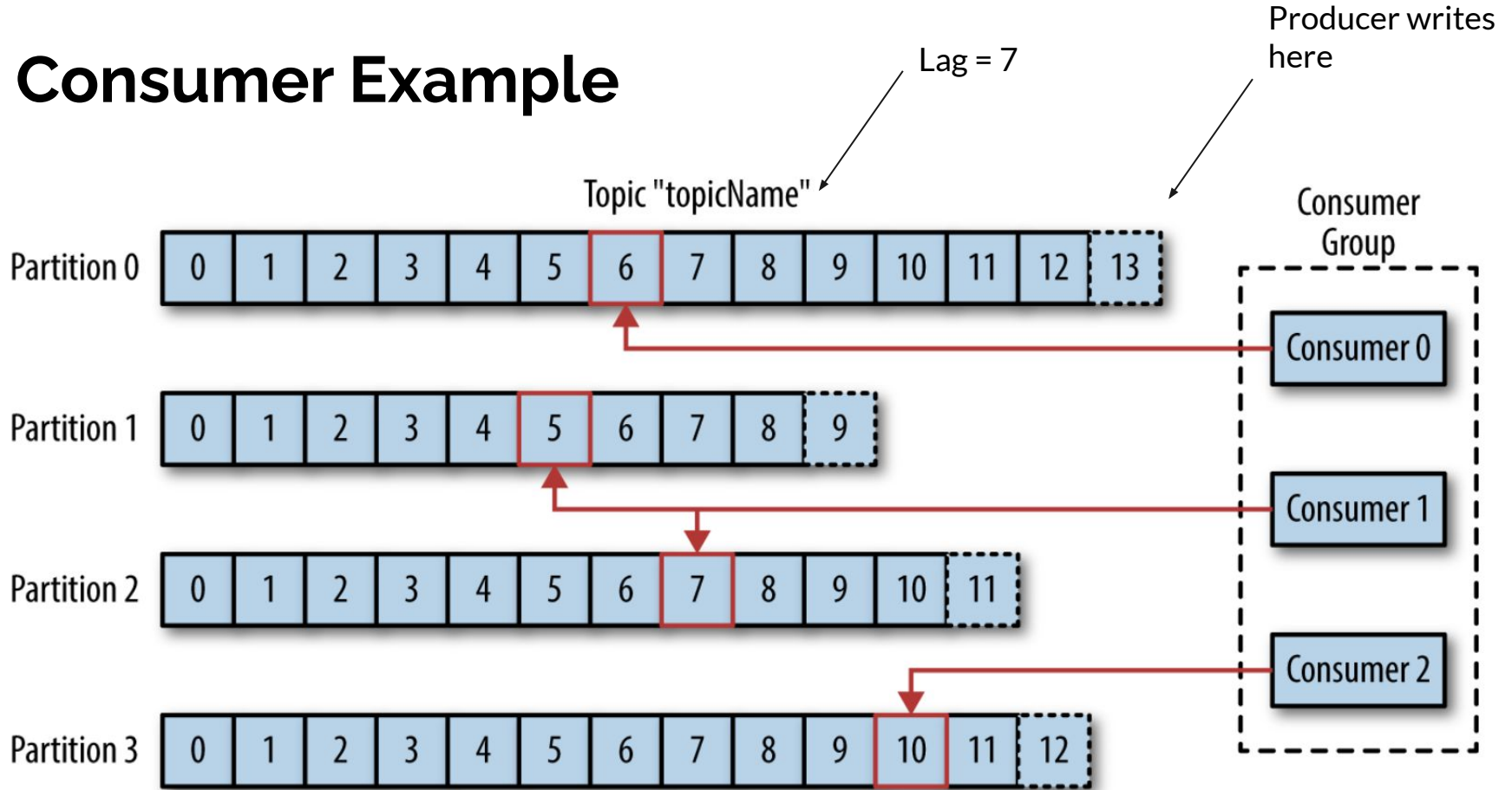
No Key Round-Robin



# Topics, Partitions, and Segments



# Consumer Example



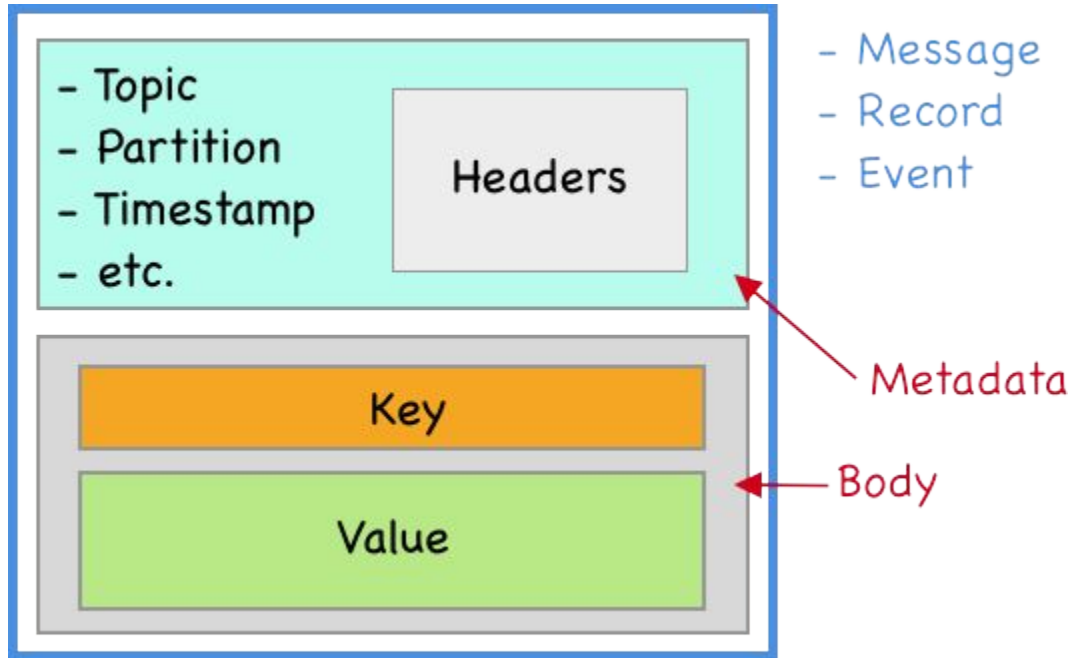
# Typical Producer API

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Precision Products",  
        "France"); ❶  
try {  
    producer.send(record); ❷  
} catch (Exception e) {  
    e.printStackTrace(); ❸  
}
```

# Typical Consumer API

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                               customer = %s, country = %s\n",
                               record.topic(), record.partition(),
                               record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

# Data Elements



# Consumers with Partitions

- Single consumer can read multiple partitions
- If consumer instances are more than partitions for a topic, then there will be no use of extra consumer instances.
- What if we want to consume a partition data from multiple services
  - Ex - We want to update data in table in RDBMS
  - We want to update in Elasticsearch
  - We want to update in Recommendation engine

# Consumer groups

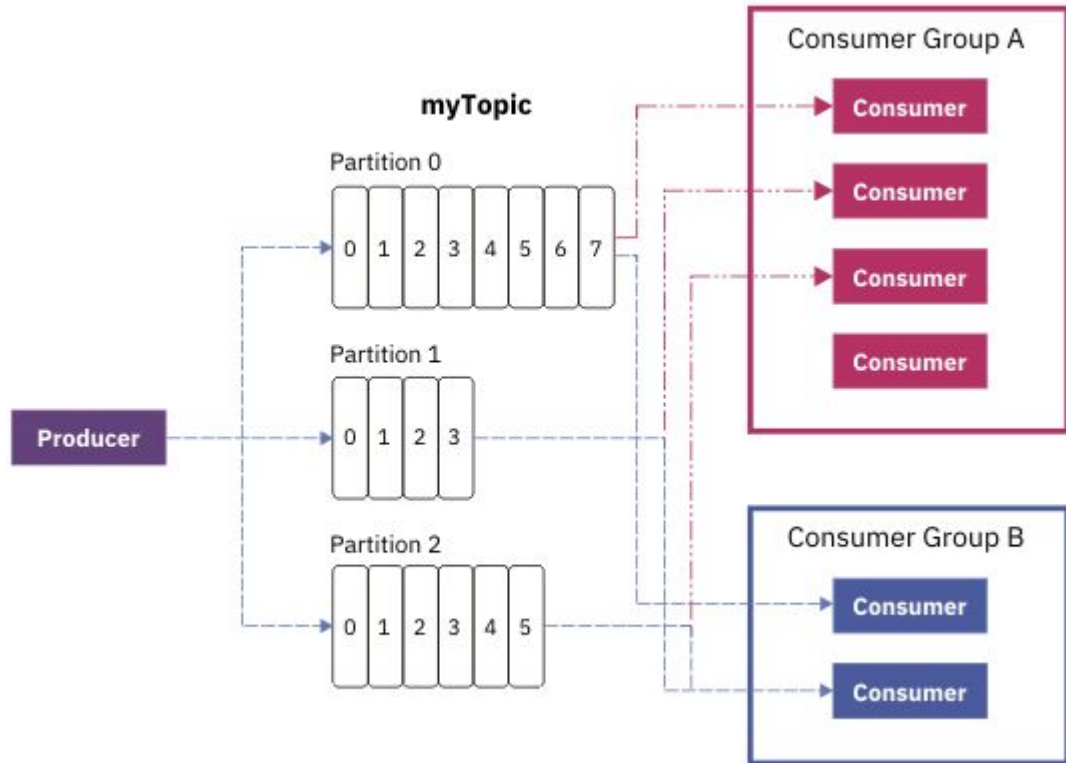
Consumers can be organized into logic consumer groups

Topic -> User Views of Product

Consumer Group A -> Add view in DB

Consumer Group B -> Update data in EL

Consumer Group C -> Send data for recommendation engine





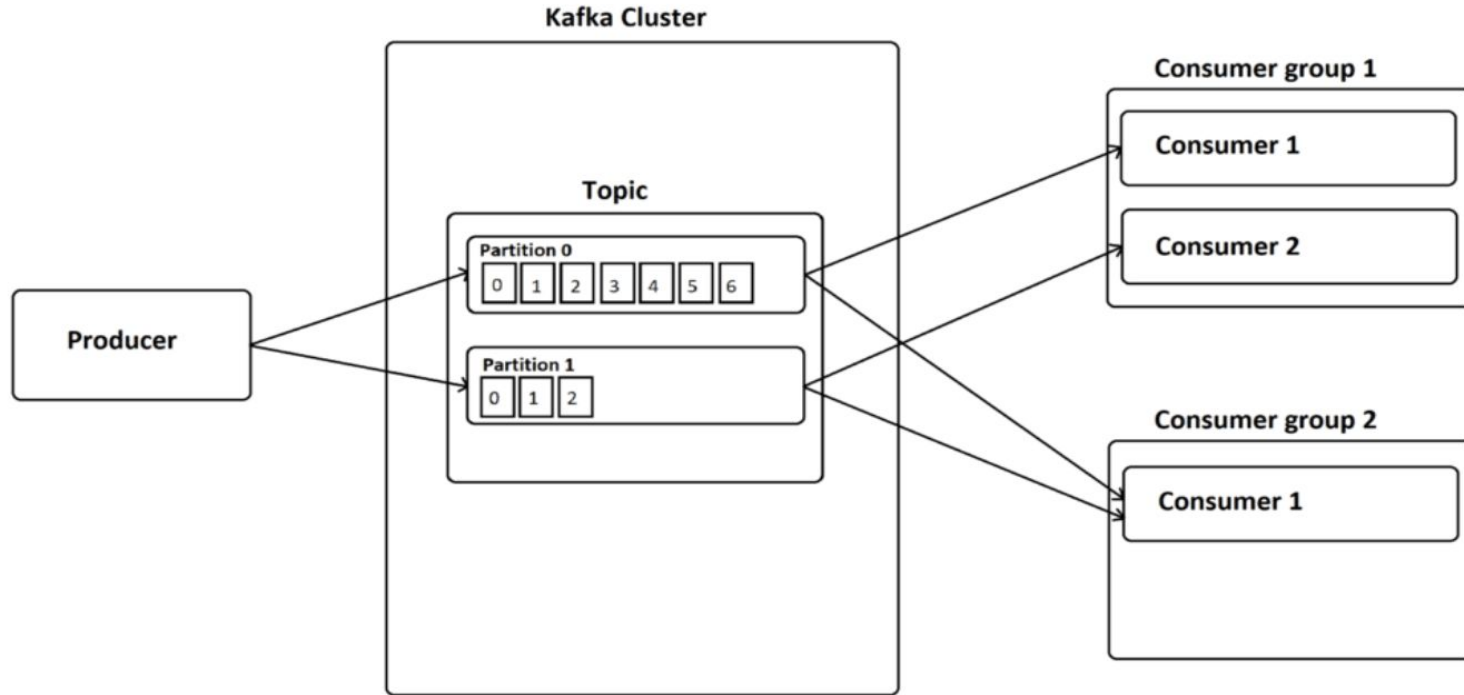


# Demo!



# Part 2

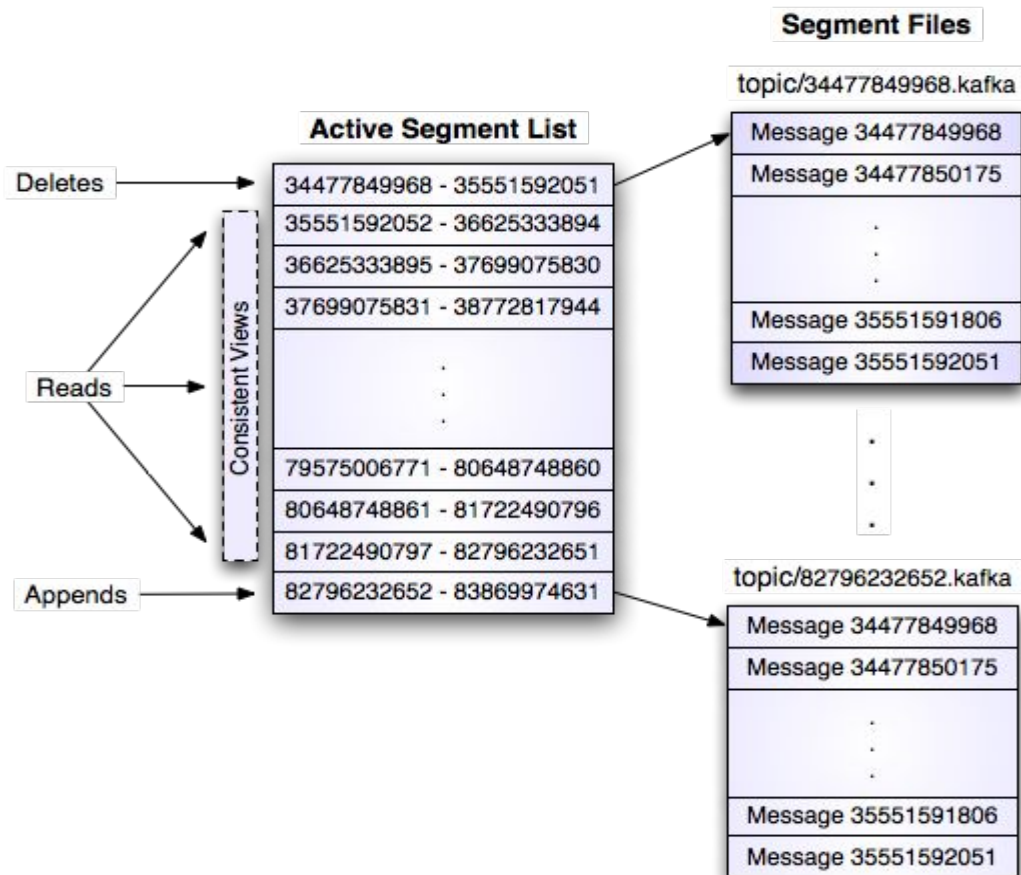
# Part1 Summary



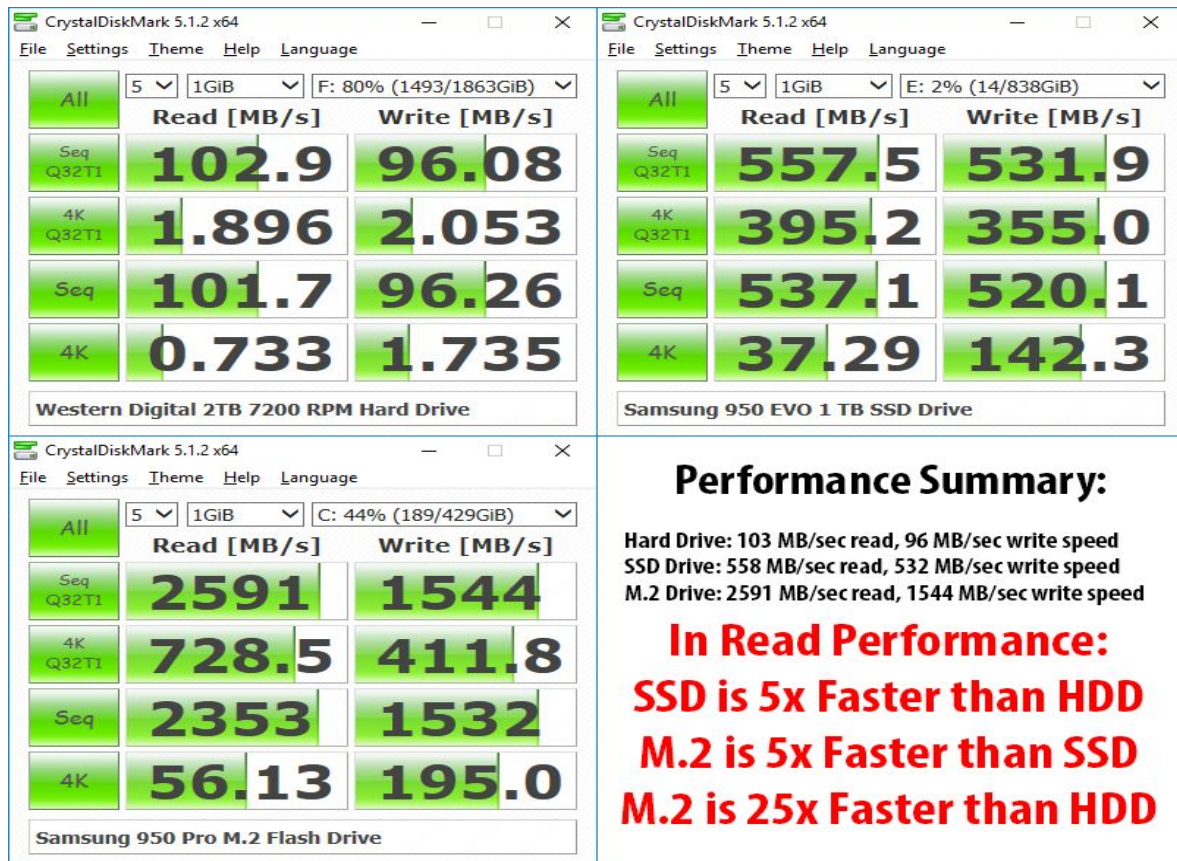
# Segments

## Kafka Log Implementation

13



# Sequential Vs Random read and write speed



## Performance Summary:

Hard Drive: 103 MB/sec read, 96 MB/sec write speed  
SSD Drive: 558 MB/sec read, 532 MB/sec write speed  
M.2 Drive: 2591 MB/sec read, 1544 MB/sec write speed

**In Read Performance:**  
**SSD is 5x Faster than HDD**  
**M.2 is 5x Faster than SSD**  
**M.2 is 25x Faster than HDD**



# Demo!



# Replication



# Why do we need replication

- Stronger durability
- Guarantee that any successfully published message will not be lost
- Higher availability
- Cluster can still accept message if one or more broker are down
- Failures can be caused by machine error, program error and network partitions.

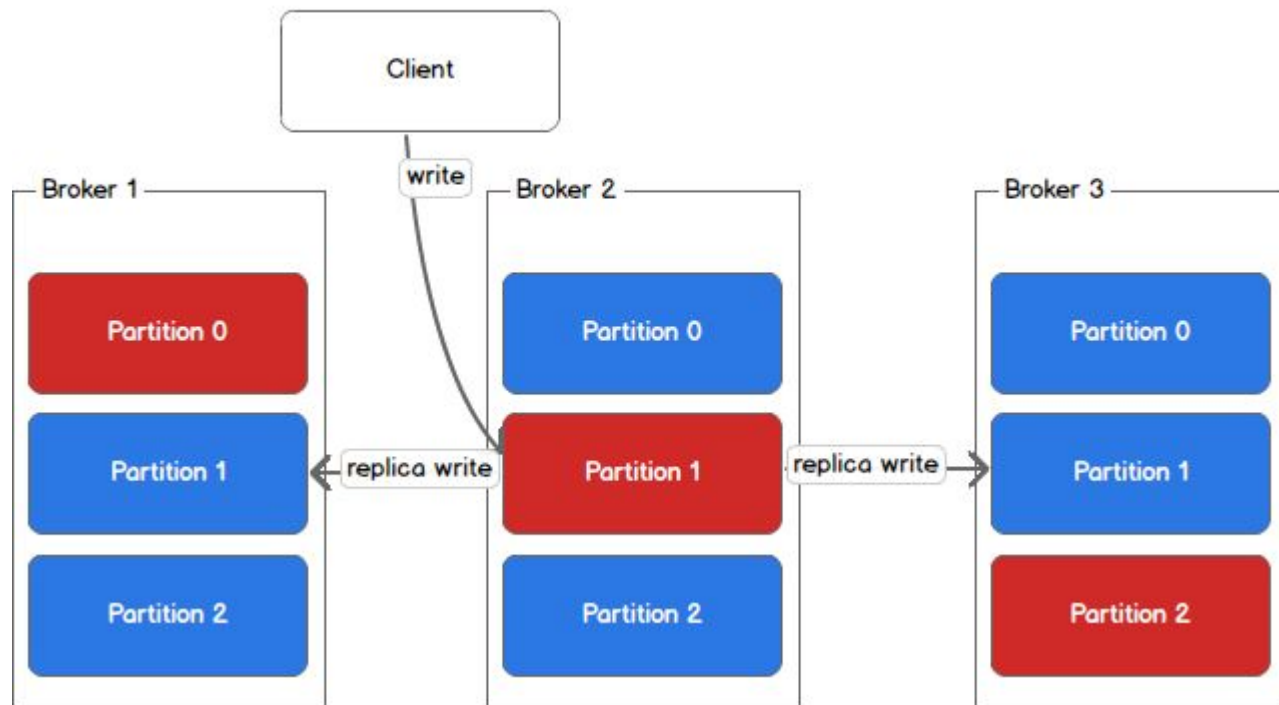
# Broker config in cluster

- There can be multiple brokers in a cluster
- Each broker will have unique broker id (broker.id)
- Each broker will store data in separate folder (logs.dir)
- Each broker will run on a different port (port)
- Default replication factor and partitions for a topic is specified in broker configs
- Data retention time params - log.retention.ms

# Replication

Replication factor = 3

Leader (red) and replicas (blue)



# Replication

- Replication happens on partition level not topic level
- For every partition, there are leader and followers
- Partition leaders are evenly distributed among brokers to avoid load on a particular broker
- Producer and Consumer both will write and read only on partition leader.
- Replica will poll and read from leader regularly
- Number of replicas can be specified globally or on topic level

# In Sync/Out of Sync replicas

- Followers replicate data from the leader to themselves by sending fetch Requests periodically, by default every 500ms.
- In-Sync Replicas are the replicated partitions that are in sync with its leader
- Replicate can go out of sync due to IO limitations, GC or slow network.
- The definition of “in-sync” means that a replica is or has been fully caught up with the leader in the last 30 seconds. (`replica.lag.time.max.ms` and `replica.lag.max.messages`)
- In-sync replicas are the subset of all the replicas for a partition having same messages as the leader

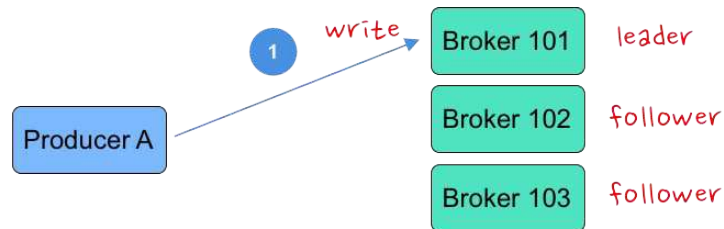
# Leader Election

<https://bravenewgeek.com/tag/leader-election/>

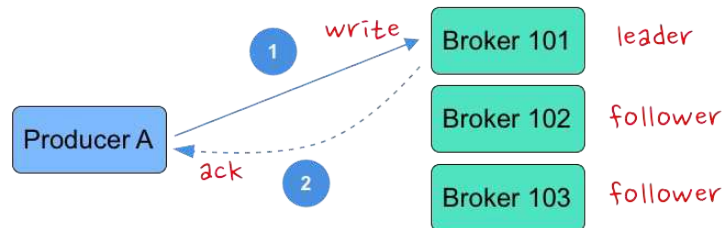
## Unclean leader election

- Unclean leader election flag to allow an out-of-sync replica to become the leader and preserve the availability of the partition.
- With unclean leader election, messages that were not synced to the new leader are lost.
- This provides balance between consistency (guaranteed message delivery) and availability.

# Producer Guarantees

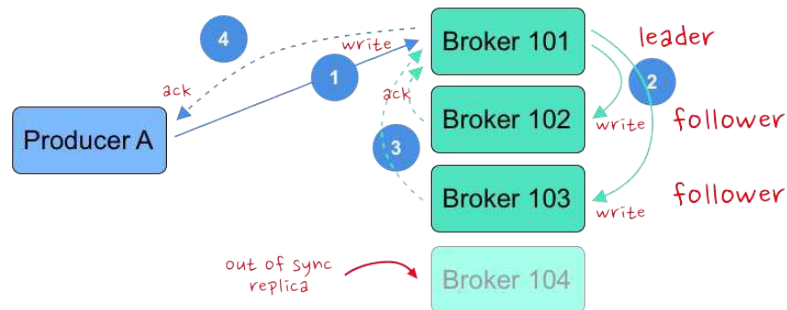


Acks 0 (NONE)



Acks 1 (LEADER)

Acks = all only needs acks from min In sync replicas



Acks -1 (ALL)

# Producer Delivery Semantics

- Producers can choose to receive acknowledgements for the data writes to the partition using the “acks” setting. (Default value is 1)
- **Acks = 0: At most once delivery, No retries!, fire and forget approach, High data loss, Ex - metrics collection, log collection**
- **Acks = 1: At least once, Retry happens in case producer does not get ACK, Moderate data loss, Moderate latency.**
- **Acks = all: acks from all min.insync.replica, exactly once delivery (Yes it is possible! In 0.11.x and later), no data loss, lower throughput and higher latency. Ex- financial applications, IoT applications, and other streaming applications**
- **Command - kafka-console-producer --broker-list localhost:9092,localhost:9093,localhost:9094 --topic demo8 --request-required-acks "all"**



# Producer Delivery Semantics

	At most once	At least once	Exactly once
Duplicates	No	Yes	No
Data loss	Yes	No	No
Processing	Zero or one time	One or more times	Exactly one time

Acks	Latency	Throughput	Durability
0	Low	High	No guarantee
1	Medium	Medium	Leader only
All	High	Low	All Replicas

# Why Kafka promises atleast once delivery not exactly once?

You can see the two categories of problems that could arise:

1) If the producer app writes to the log but fails to get the acknowledgement back over the network. This would put this producer in a bind: it could be that the write had actually succeed or it could be that it never got to Kafka at all. We don't know! If we retry and it turns out the write had succeeded we could then have a duplicate; if we don't retry and it turns out the write hadn't succeed we'll lose that write.

2) The second problem comes on the consumer side.

The consumer application could read some messages from the log and write the results to its database, but fail before updating its offset marking its position. When that consumer was restarted (potentially automatically on a different machine using Kafka group management mechanism) that could result in duplicates. If the application updates its stored offset first and then updates the database, the failure could instead lead to missing that update upon restart.

# Why Kafka promises atleast once delivery not exactly once?

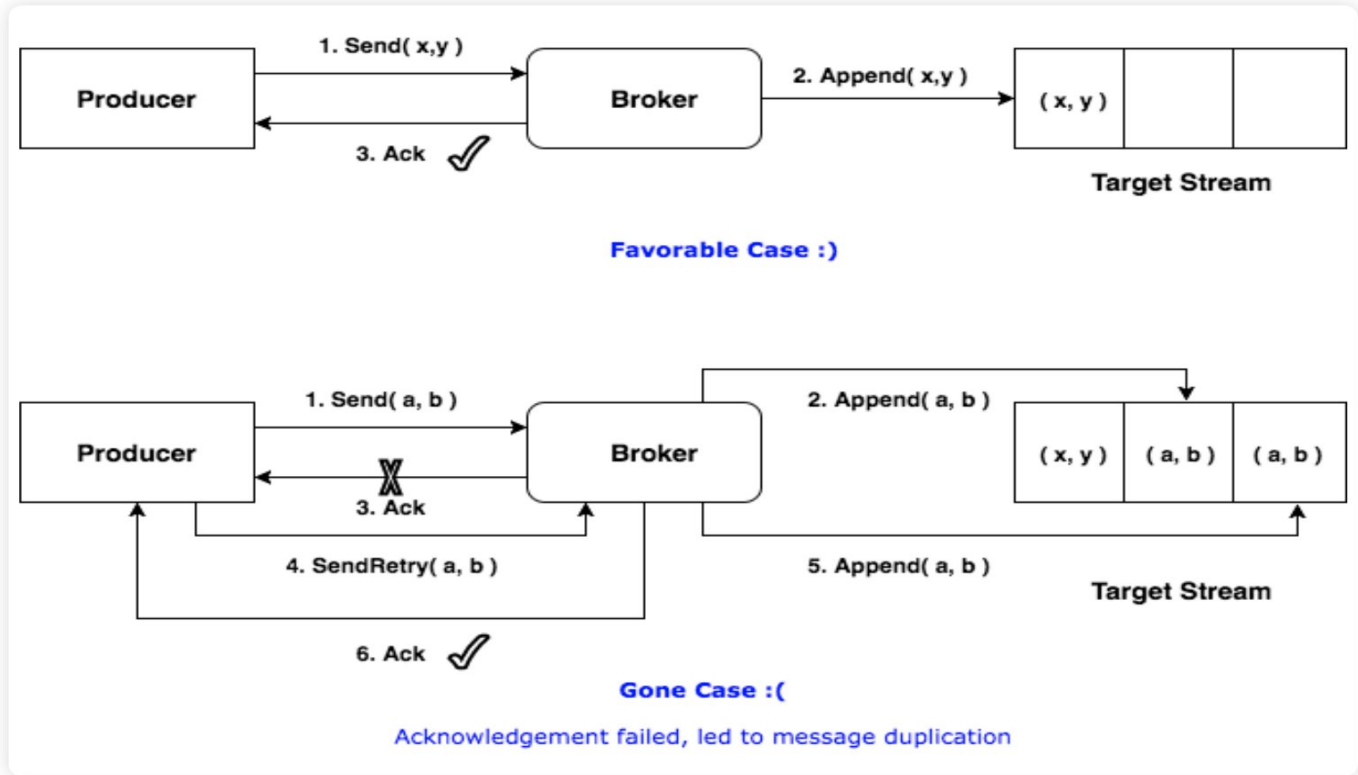
In order to achieve exactly once delivery, we will have to achieve the following -

1. Message from Producer to Broker should be delivered only once
2. Message should be consumed only once by consumer

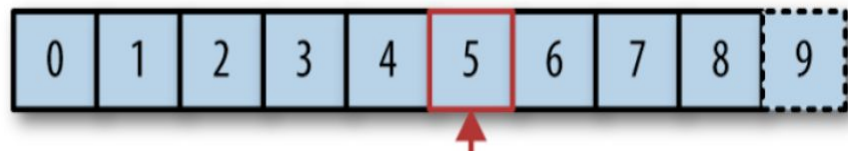
Let's see what are the problems we face in order to achieve above

# Why kafka promises atleast once delivery not exactly once?

## Producer side Guarantee



# Why kafka promises atleast once delivery not exactly once?



## Consumer side Guarantee

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                               customer = %s, country = %s\n",
                               record.topic(), record.partition(),
                               record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

# Kafka Topic Properties

- `bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic yourTopicName --partitions 1 --replication-factor 3 --config min.insync.replicas=2`
- `--replication-factor = 3` means data partition data should be present in 1 Leader + 2 Followers
- `min.insync.replicas` specifies the minimum number of replicas that must acknowledge a write in order to consider this write as successful.
- `min.insync.replicas = 3` means 1 Leader + 2 Followers

# Min.insync.replicas and Acks Parameter

- `min.insync.replicas` can be specified on creating topics.
- `min.insync.replicas` and `acks` to “all” (or “-1”) specifies the minimum number of replicas that must acknowledge a write in order to consider this write as successful
- If this minimum cannot be met, then the producer will raise an exception (either `NotEnoughReplicas` or `NotEnoughReplicasAfterAppend`).
- `min.insync.replicas` and `acks` allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of 3, set `min.insync.replicas` to 2, and produce with `acks` of “all”. This will ensure that the producer raises an exception if a majority of replicas do not receive a write.



# Demo!



# More to explore

- Producer design
- Consumer properties
- Why Kafka is so Fastt

# References

- <https://book.huihoo.com/pdf/confluent-kafka-definitive-guide-complete.pdf>
- [https://docs.cloudera.com/documentation/kafka/latest/topics/kafka\\_ha.html](https://docs.cloudera.com/documentation/kafka/latest/topics/kafka_ha.html)
- <https://kafka.apache.org/documentation.html#replication>
- <https://bravenewgeek.com/tag/leader-election/>
- <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it>