

Welcome



K. VENKATA RAMANA



Venkata Ramana

Over three decades of IT experience in Corporate Training, Software development and architectural designs involving web technologies, databases, SOA, Microservices & Cloud.

A Prayer of Gratitude to Mother Nature

Though I come with over three decades of experience, I know deeply that these are just figures.

All actions are performed by the grace of Mother Nature.
In my ignorance, I may sometimes believe:
"I am the doer." .. But no—never.

Mother Nature is the true doer.

I am merely an instrument through which Her work flows.
Knowledge, too, flows not from me, but from Her.
I surrender my ego at Her feet.
With this awareness, I step into this space—open, humble, and ready to share whatever flows through me.

Synchronous Vs Asynchronous

In synchronous operations tasks are performed one at a time and only when one is completed, the following is unblocked. In other words, we need to wait for a task to finish to move to the next one.

In asynchronous operations, on the other hand, we can move to another task before the previous one finishes.

It is possible to make the logic asynchronous in Java with the following concepts, but they have their own limitations.

Callbacks

- Complex
- No Return Value
- Code is hard to read and maintain

Futures

- Returns Future instance
- Hard to compose multiple async operations

Completable Future

- Introduced in Java 8
- Supports functional style API
- Not a great fit asynchronous call with multiple items

A messaging system is responsible for transferring data from one application to another so the applications.

Distributed messaging is based on the concept of reliable message queuing.

Messages are queued asynchronously between client applications and messaging system.

There are two types of messaging patterns. The first one is a point-to-point and the other one is “publish–subscribe” (pub-sub) messaging system. Most of the messaging systems follow the pub-sub pattern.



Activemq

vs



Kafka

ActiveMQ	Kafka
<p data-bbox="264 376 1207 496">It is a traditional messaging system that deals with a small amount of data. It has the following use cases:</p> <ul data-bbox="308 591 1245 1033" style="list-style-type: none"><li data-bbox="308 591 800 634">• Transactional Messaging<li data-bbox="308 668 1131 711">• High-Performance market data distribution<li data-bbox="308 745 1245 865">• Clustering and general-purpose async messaging model<li data-bbox="308 899 774 942">• Web Streaming of data<li data-bbox="308 976 1034 1033">• Restful API to messaging using HTTP	<p data-bbox="1302 376 2244 496">It is a distributed system meant for processing a huge amount of data. It has the following use cases:</p> <ul data-bbox="1345 591 1857 1119" style="list-style-type: none"><li data-bbox="1345 591 1589 634">• Messaging<li data-bbox="1345 668 1857 711">• Website Activity Tracking<li data-bbox="1345 745 1531 788">• Metrics<li data-bbox="1345 822 1702 865">• Log Aggregation<li data-bbox="1345 899 1735 942">• Stream Processing<li data-bbox="1345 976 1666 1019">• Event Sourcing<li data-bbox="1345 1053 1615 1119">• Commit Log

<p>It has transaction support. The two levels of transactions support are:</p> <ul style="list-style-type: none">• JMS Transactions• XA Transactions <p>It uses TransactionStore to handle transactions. TransactionStore will cache all messages and ACKS until commit or rollback occurs.</p>	<p>Kafka initially didn't support transactions, but since its 0.11 release, it does support transactions to some extent.</p>
<p>It maintains the delivery state of every message resulting in lower throughput.</p>	<p>Kafka producers don't wait for acknowledgments from the Brokers. So, brokers can write messages at a very high rate resulting in higher throughput.</p>

<p>In ActiveMQ, it's the responsibility of the producers to ensure that messages have been delivered.</p>	<p>In Kafka, it's the responsibility of the consumers to consume all the messages they are supposed to consume.</p>
<p>It cannot ensure that messages are received in the same order they were sent.</p>	<p>It can ensure that messages are received in the order they were sent at the partition level.</p>
<p>There is something called JMS API message selector, which allows a consumer to specify the messages it is interested in. So, the work of filtering messages is upto the JMS and not the applications.</p>	<p>Kafka doesn't have any concept of filters at the brokers that can ensure that messages that are picked up by consumers match a certain criterion. The filtering has to be done by the consumers or by the applications.</p>

It is a push-type messaging platform where the providers push the messages to the consumers.	It is a pull-type messaging platform where the consumers pull the messages from the brokers.
It is not possible to scale horizontally. There is also no concept of replication.	It is highly scalable. Due to replications of partitions, it offers higher availability too.
The performance of both queue and topic degrades as the number of consumers rises.	It doesn't slow down with the addition of new consumers.
It doesn't provide checksums to detect corruption of messages out of the box.	It includes checksums to detect corruption of messages in storage and has a comprehensive set of security features.

Kafka

Kafka is a distributed streaming platform with very good horizontal scaling capability. It allows applications to process and re-process streamed data on disk.

Due to its high throughput it's commonly used for real-time data streaming.

ActiveMQ

ActiveMQ is a general-purpose message broker that supports several messaging protocols such as AMQP, STOMP, MQTT. It supports more complicated message routing patterns as well as the Enterprise Integration Patterns. In general it is mainly used for integration between applications/services especially in a Service Oriented Architecture.

ActiveMQ Broker had to maintain the delivery state of every message resulting into lower throughput.

Kafka producer doesn't wait for acknowledgements from the broker unlike in ActiveMQ and sends messages as fast as the broker can handle. Overall throughput will be high if broker can handle the messages as fast as producer.

Kafka has a more efficient storage format. On average, each message had an overhead of 9 bytes in Kafka, versus 144 bytes in ActiveMQ.

In ActiveMQ, Producer send message to Broker and Broker push messages to all consumers. Producer has responsibility to ensure that message has been delivered. In Kafka, Consumer will pull messages from broker at its own time. It's the responsibility of consumer to consume the messages it has supposed to consume.

Slow Consumers in AMQ can cause problems on non-durable topics since they can force the broker to keep old messages in RAM which once it fills up, forces the broker to slow down producers, causing the fast consumers to be slowed down. A slow consumer in Kafka does not impact other consumers.

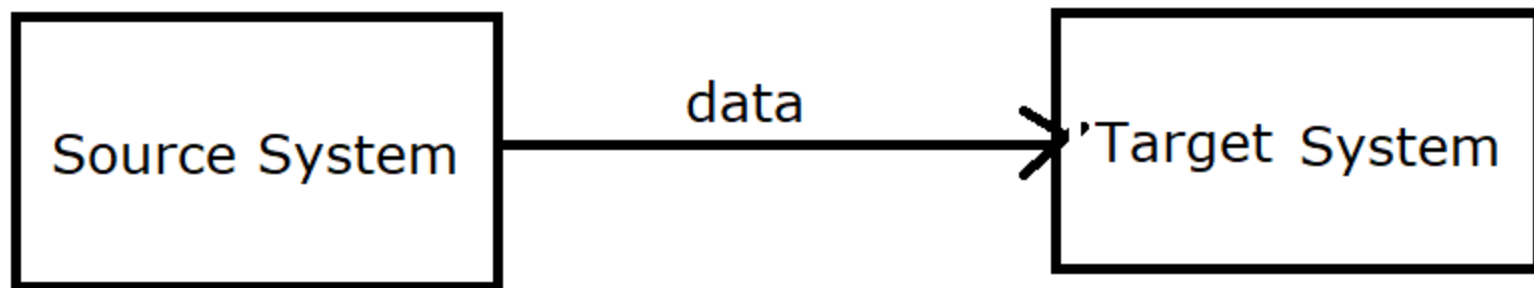
In Kafka - A consumer can rewind back to an old offset and re-consume data. It is useful when we fix some issue and decide to re-play the old messages post issue resolution.

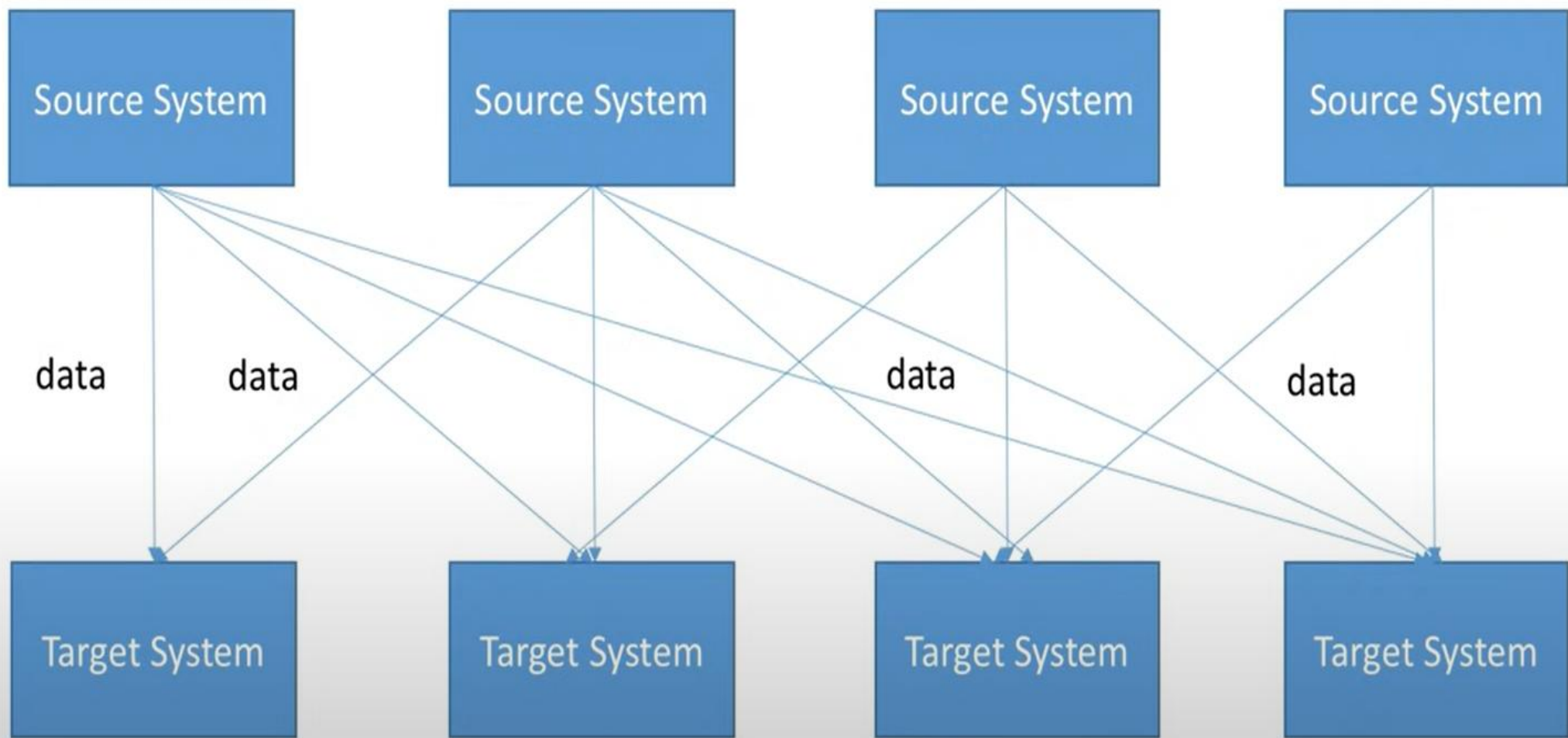
Performance of Queue and Topics may degrade with addition of more consumers in ActiveMQ. But Kafka does not have that dis-advantage with addition of more consumers.

Kafka is highly scalable due to replication of partitions. It can ensure that messages are delivered in a sequence within a partition.

ActiveMQ is traditional messaging system whereas Kafka is meant for distributed processing system with huge amount of data and effective for stream processing

Why Kafka?



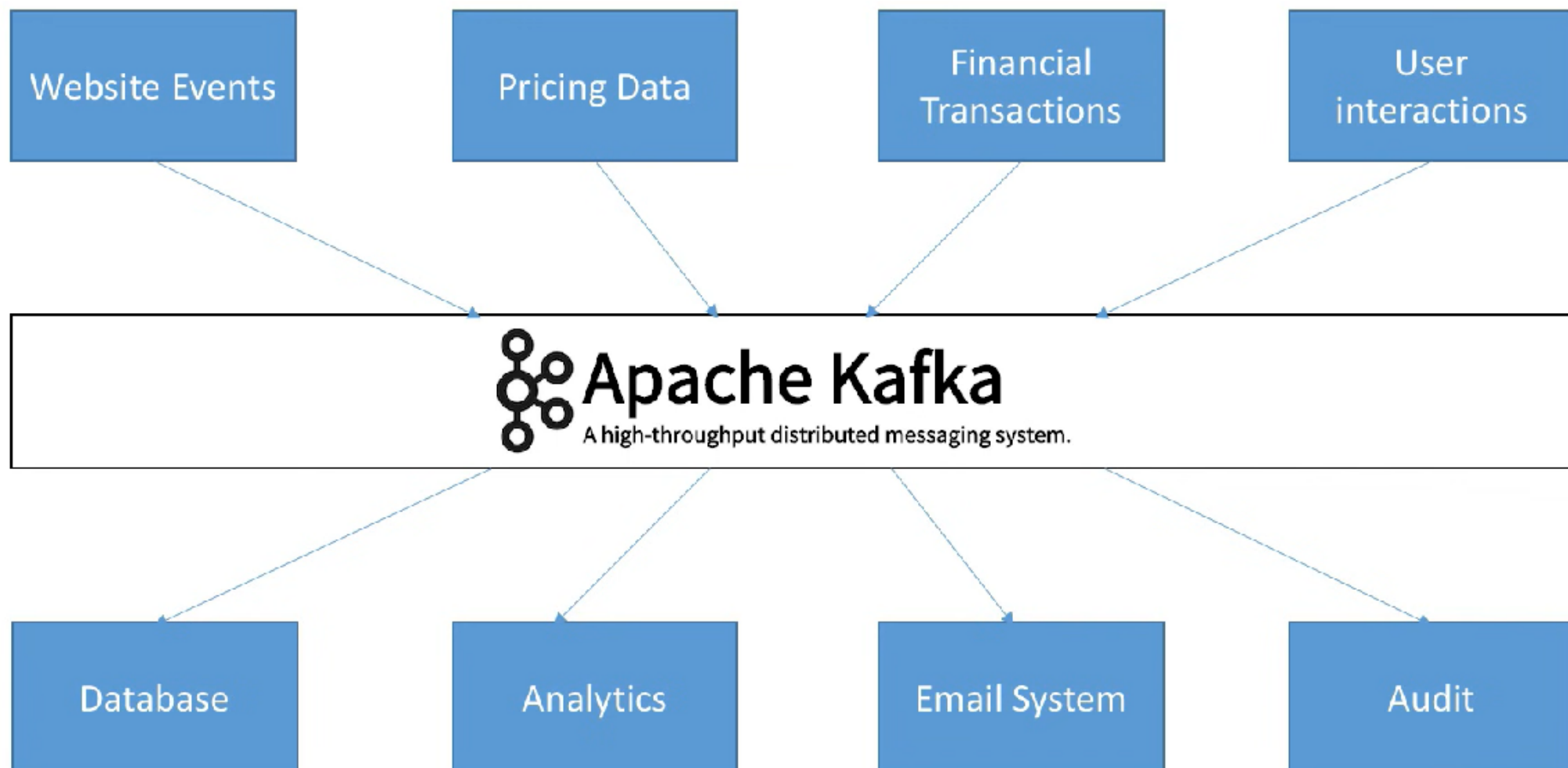


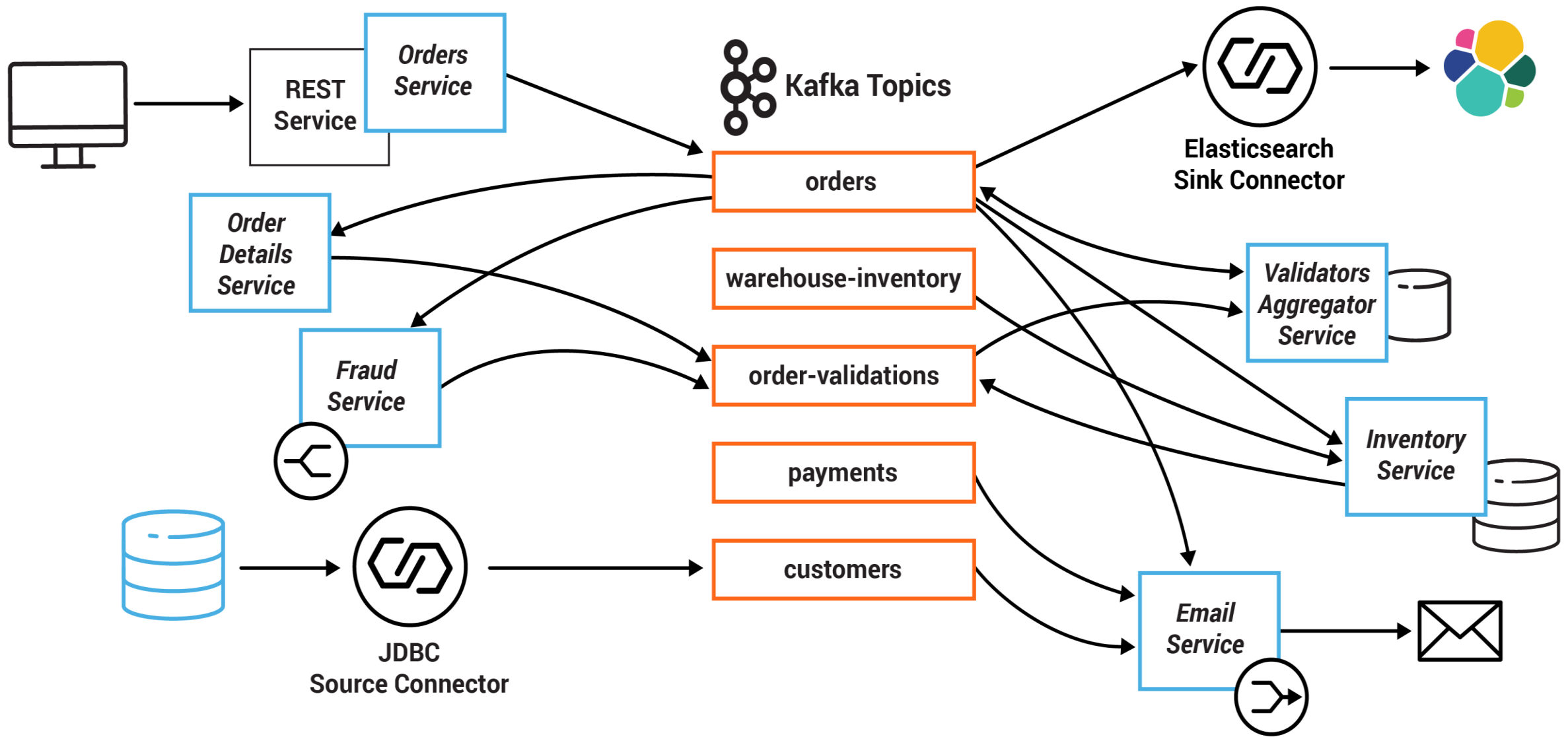
Problems organisations are facing with the previous architecture

- 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

Why Apache Kafka:

Decoupling of data streams & systems





Apache Kafka® is a distributed streaming platform.

- ☐ **Distributed**
- ☐ **partitioned,**
- ☐ **replicated commit log service.**

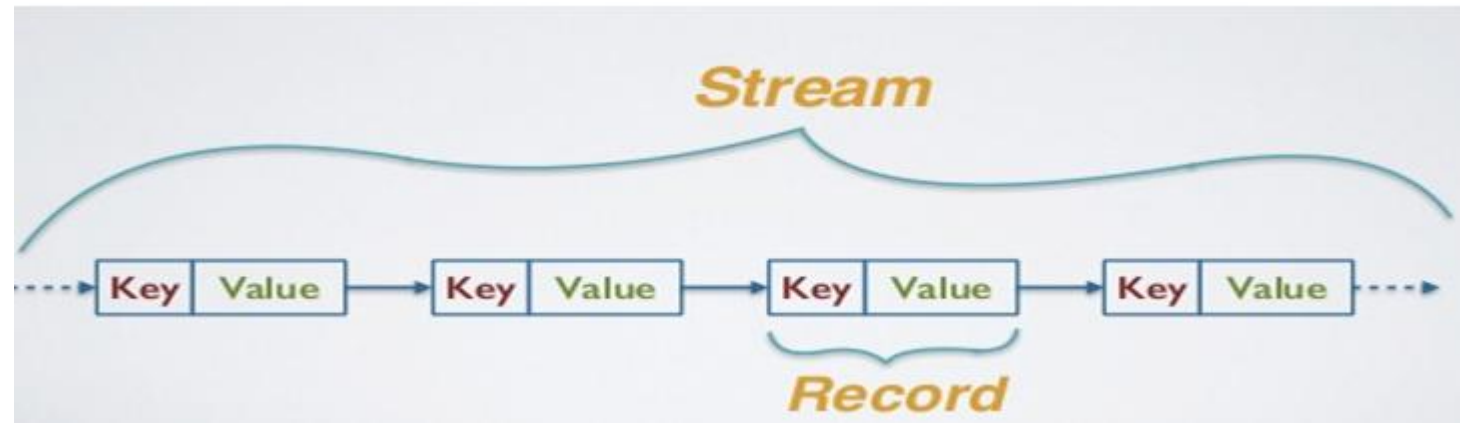
**It provides the functionality of a messaging system,
but with a unique design.**

What is a Stream?

Stream is the continuous high-speed transfer of large amounts of data from a source system to a target.

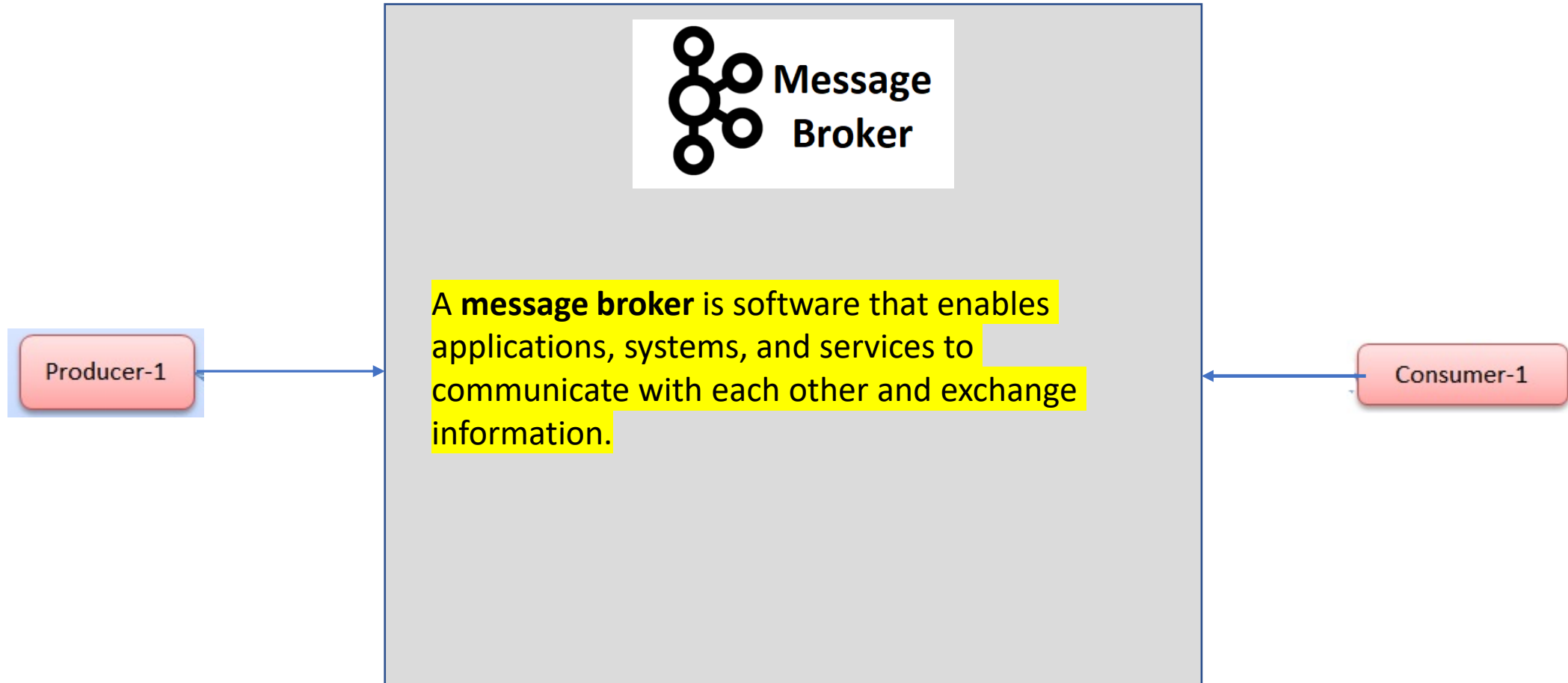
It is unbounded, continuous real-time flow of records.

Records are key-value pairs.





**Message
Broker**





Topic : Order



Topic : Order

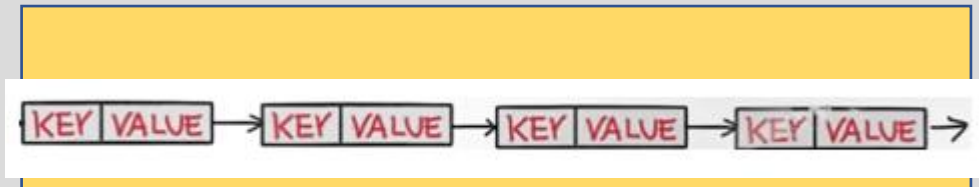
Topic is a category/feed name to which records are stored.

Topics are divided into a number of partitions.



Topic : Order

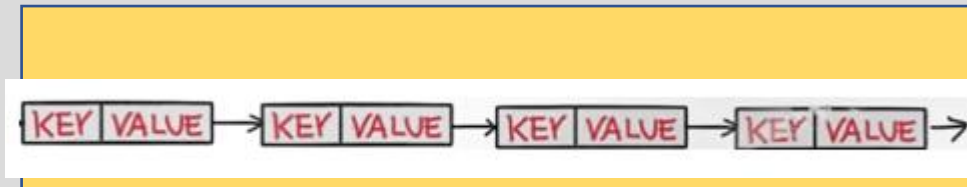
Partition





Topic : Order

Partition

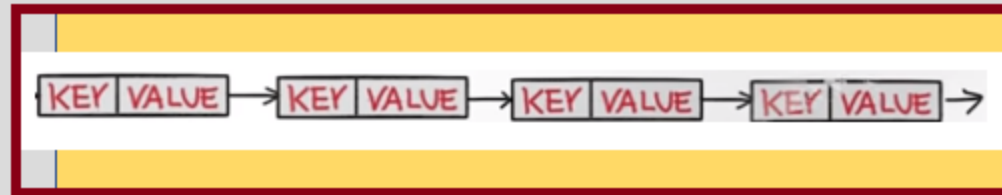


Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers

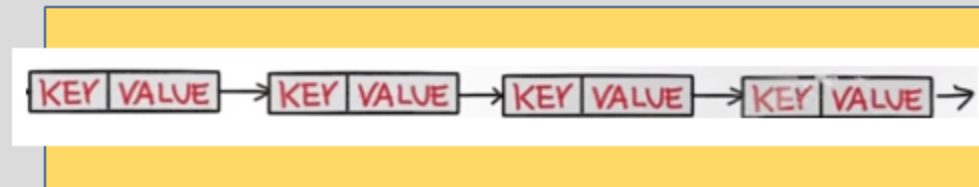


Topic : Order

Partition 0



Partition 1

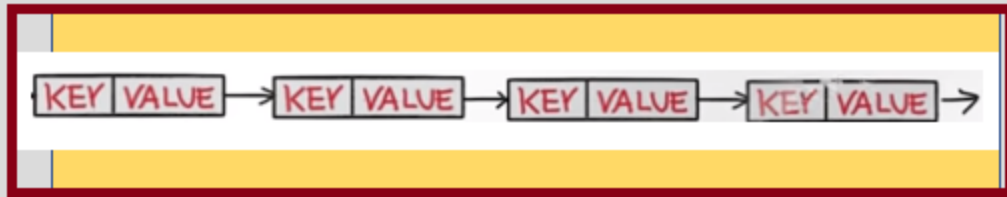




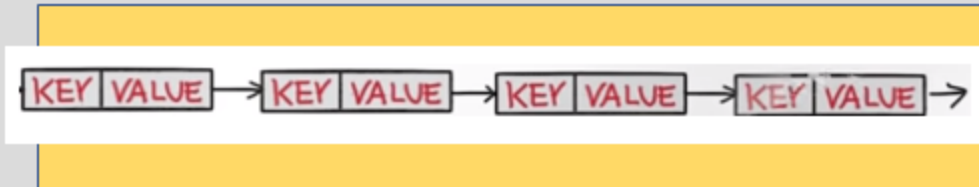
Message
Broker

Topic : Order

Partition 0



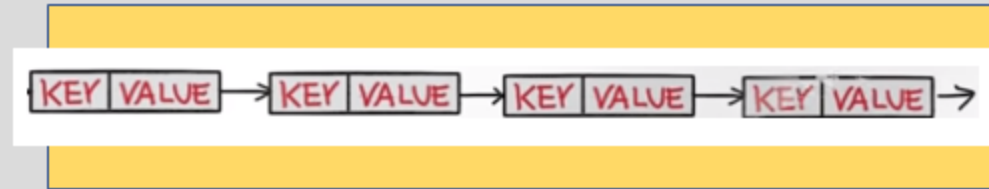
Partition 1



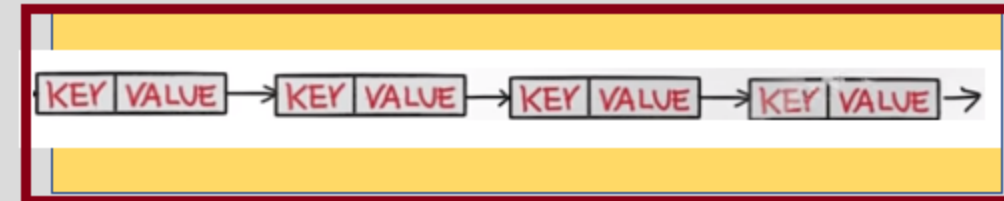
Message
Broker

Topic : Order

Partition 0



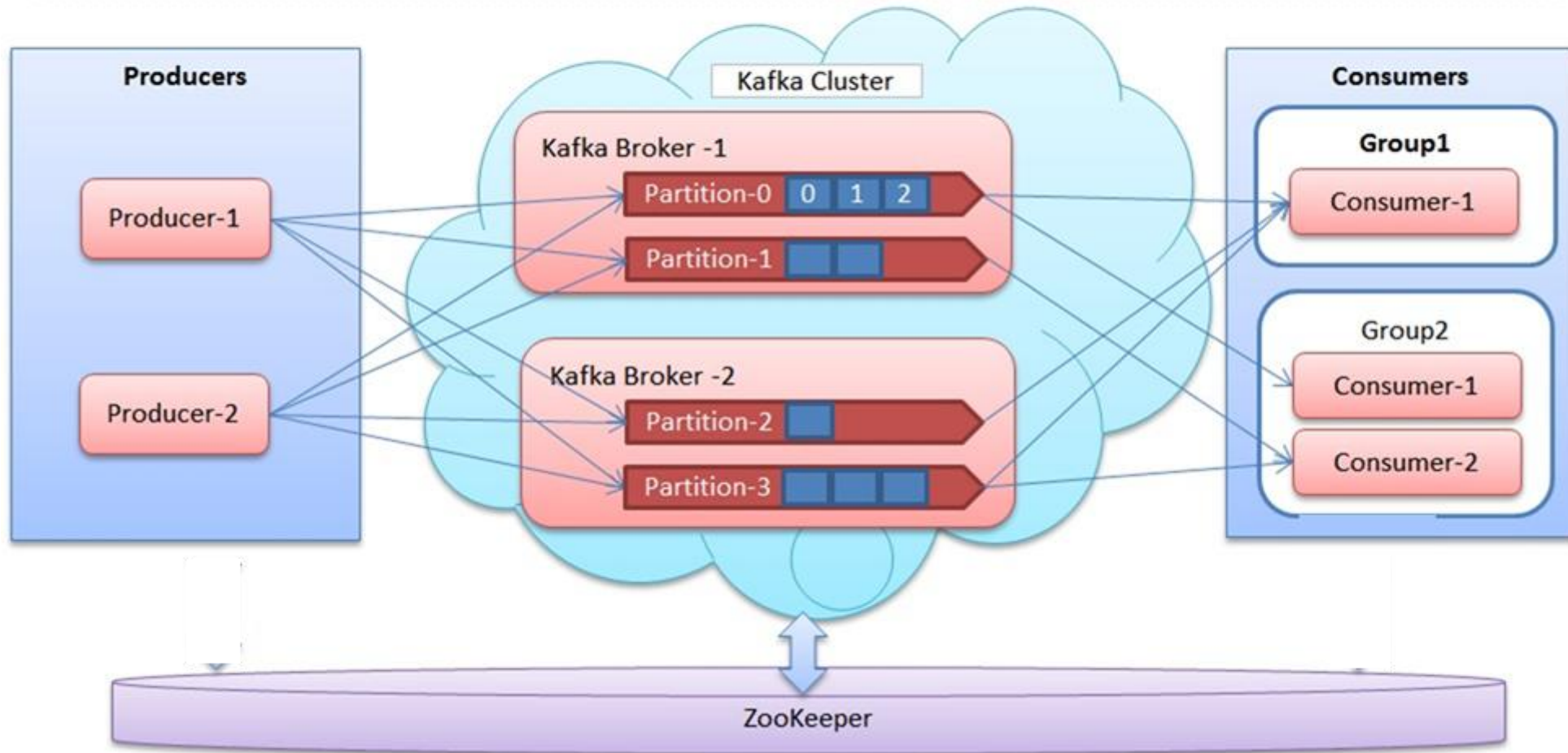
Partition 1

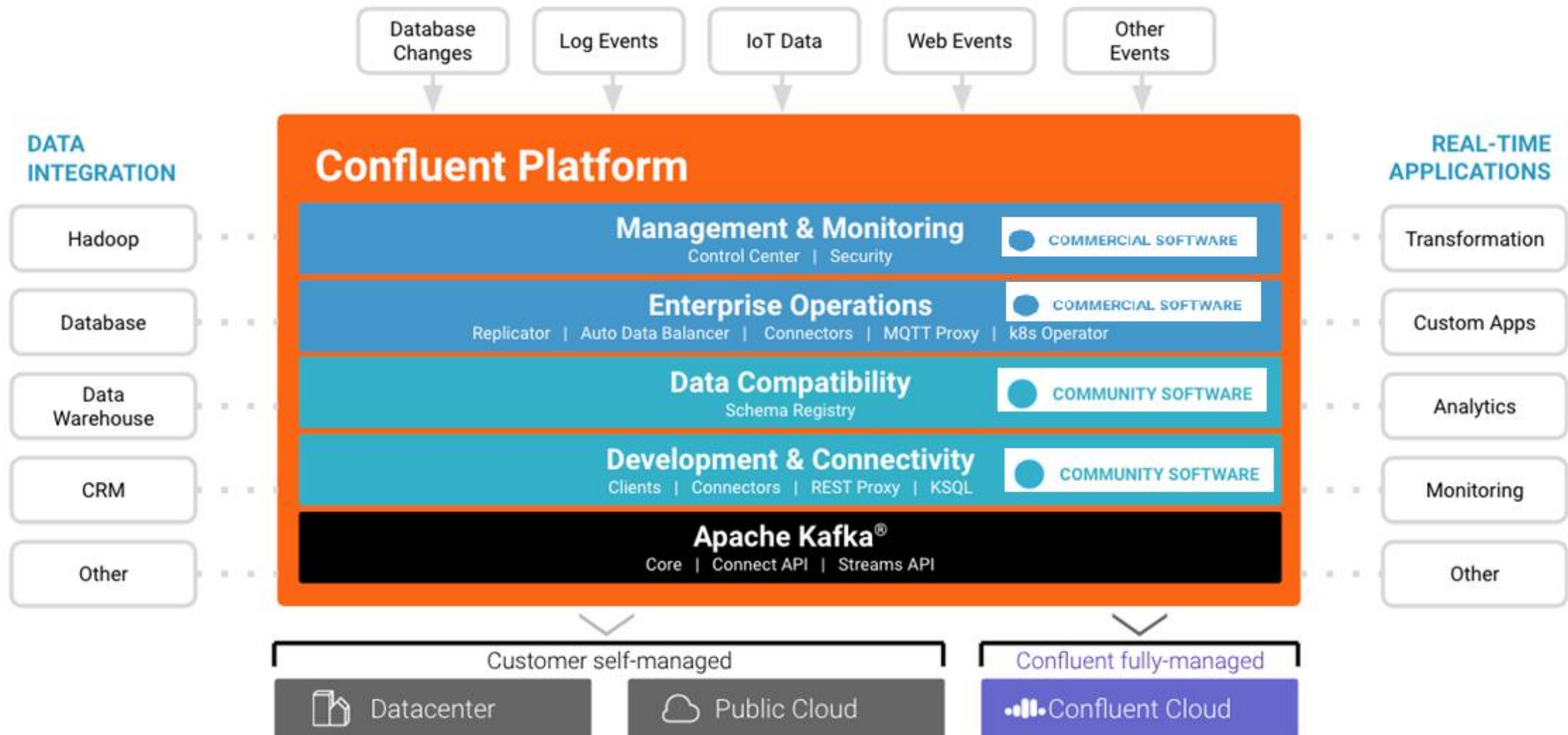




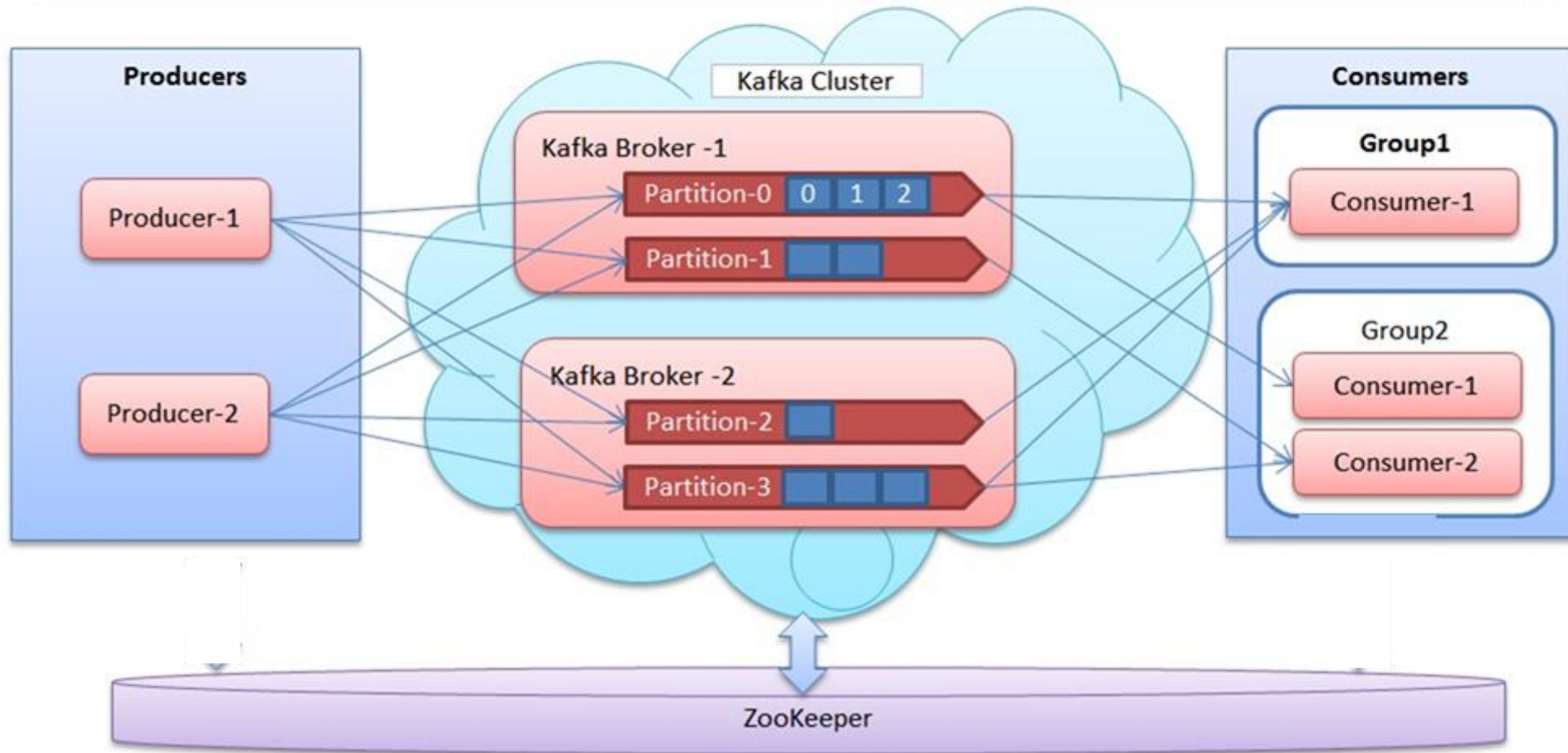
- ❑ Distributed
- ❑ **partitioned,**
- ❑ **replicated commit log service.**

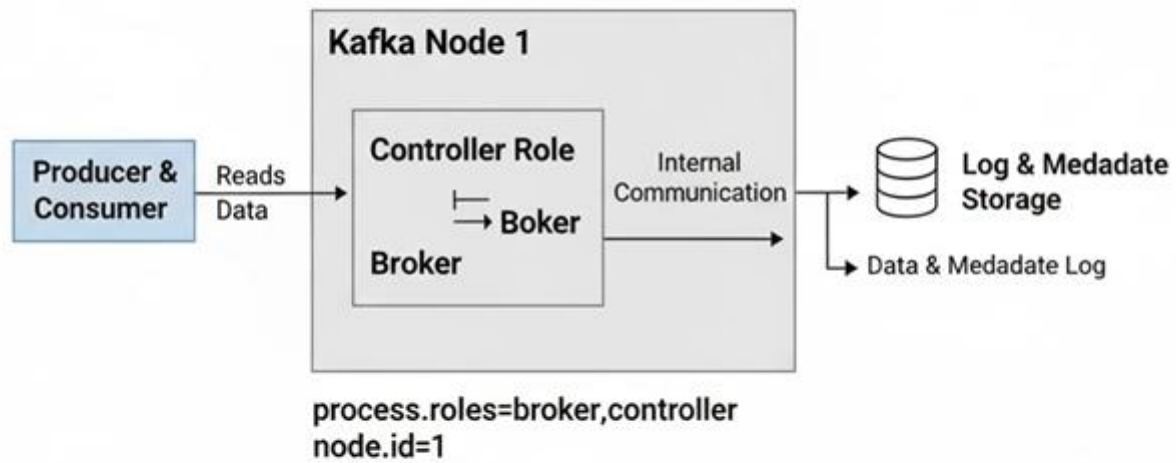
In this diagram, two brokers are shown in a kafka cluster. Two producers are publishing messages to one topic with 4 partitions. Each broker have two partitions. Two consumer groups are configured, group1 has one consumer and group2 has two consumers. For each consumer group, messages are guaranteed to be consumed at least once.





In this diagram, two brokers are shown in a kafka cluster. Two producers are publishing messages to one topic with 4 partitions. Each broker have two partitions. Two consumer groups are configured, group1 has one consumer and group2 has two consumers. For each consumer group, messages are guaranteed to be consumed at least once.





KRaft Architecture and Purpose

1. The Problem (ZooKeeper Dependency)

In traditional Kafka architecture, the Controller—the broker responsible for maintaining and propagating cluster metadata—relied entirely on **ZooKeeper** to store the cluster state, handle leader elections, and manage configuration changes. This created several problems:

- **Operational Overhead:** Required setting up and managing a separate ZooKeeper quorum.
- **Performance Bottleneck:** ZooKeeper's performance limitations often capped the size and scaling rate of very large Kafka clusters, especially during failovers.
- **Code Complexity:** It required two separate systems to be synchronized.

2. The Solution (KRaft)

KRaft integrates the metadata management directly into Kafka brokers by implementing the **Raft consensus algorithm**.

- **Raft Protocol:** Raft is a protocol designed to manage a replicated log to ensure consistency and availability. In Kafka, it is used to replicate the **metadata log** across a small set of designated nodes (the **Controller Quorum**).
- **Internal Consensus:** The Kafka nodes themselves elect a metadata leader (the **Controller Leader**) and manage all state changes, making the cluster entirely self-contained.

Key Benefits of KRaft

Benefit	Description
Simplicity	Eliminates the operational complexity of managing a separate ZooKeeper cluster. Kafka is now a single deployable entity.
Scalability	Achieves faster metadata operations (like broker registration, topic creation, and partition reassignments) by handling them directly in Kafka, allowing for much larger clusters.
Faster Failover	Controller failover time is significantly reduced because metadata updates are immediately available on the Raft log, improving overall cluster stability.
Architectural Unification	The design simplifies the codebase and allows for greater consolidation of responsibilities.

Deployment Modes

KRaft supports different deployment models based on how you assign the roles:

1. **Combined Role:** A single node acts as both the **Controller** (participates in metadata consensus) and the **Broker** (handles client data). This is ideal for development and small clusters.
2. **Separate Roles (Production):** A small, dedicated set of nodes act only as **Controllers** (metadata-only), while the remaining, larger set of nodes act only as **Brokers** (data-only). This provides the best isolation and performance for large production clusters.

1. 📁 Raft's Core Concepts

Every node in a Raft cluster operates in one of three states:

A. Leader 👑

- Handles all client requests (writes/reads).
- Manages the replicated log, ensuring all followers eventually match its log.
- Sends periodic **Heartbeat** messages to Followers to maintain its authority.

B. Follower 👤

- Passive state; responds to requests from the Leader and Candidate nodes.
- Log entries are strictly replicated from the Leader.
- If a Follower doesn't receive a Heartbeat from the Leader within a certain **election timeout**, it assumes the Leader has failed and transitions to the Candidate state.

C. Candidate 🗳️

- A node transitions to this state when it believes the Leader has failed.
 - It immediately votes for itself and sends **RequestVote** RPCs to all other nodes.
 - If it receives votes from a **Quorum** (a majority) of the nodes, it wins the election and becomes the new Leader.
-

Why the Candidate State Exists

1. **Detection of Failure:** A **Follower** waits for a heartbeat from the Leader. If the heartbeat stops for a certain timeout, the Follower assumes the Leader is dead.
2. **Transition to Candidate:** The Follower immediately transitions to the **Candidate** state to start a new election.
3. **Campaigning:** As a Candidate, the node's only job is to gather enough votes (**a quorum**) to become the new **Leader**. If it succeeds, it transitions to the **Leader** state. If another node wins the election, it reverts back to being a **Follower**.

So, a **Candidate** is a node that is actively trying to become the **Leader**, having temporarily abandoned its role as a **Follower**.

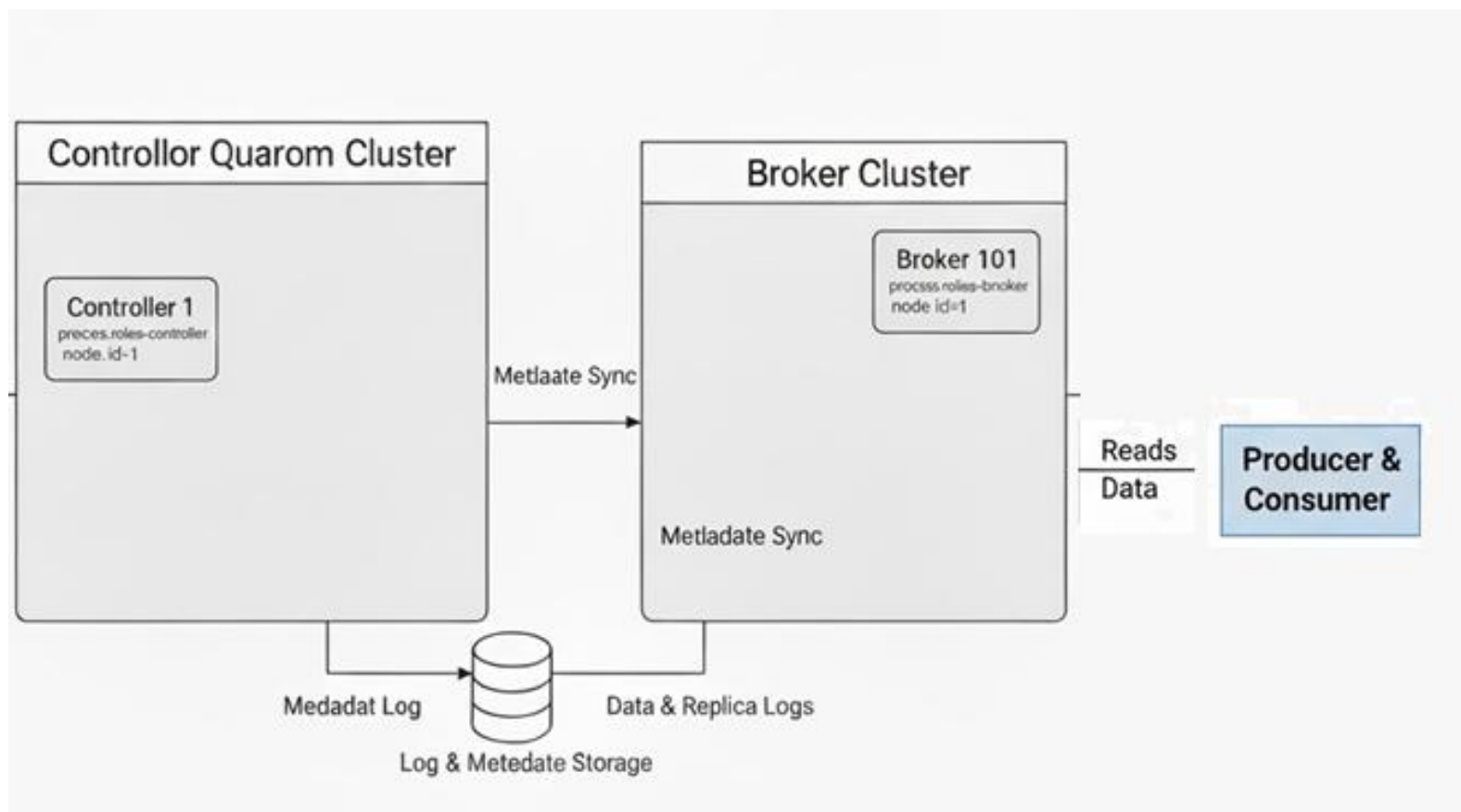
2. 🛡️ What is a Quorum?

In the context of Raft and distributed systems, the **Quorum** is the **minimum number of votes or nodes required to agree on a decision** (like electing a Leader or committing a log entry) for that decision to be considered valid and permanent. 🔗

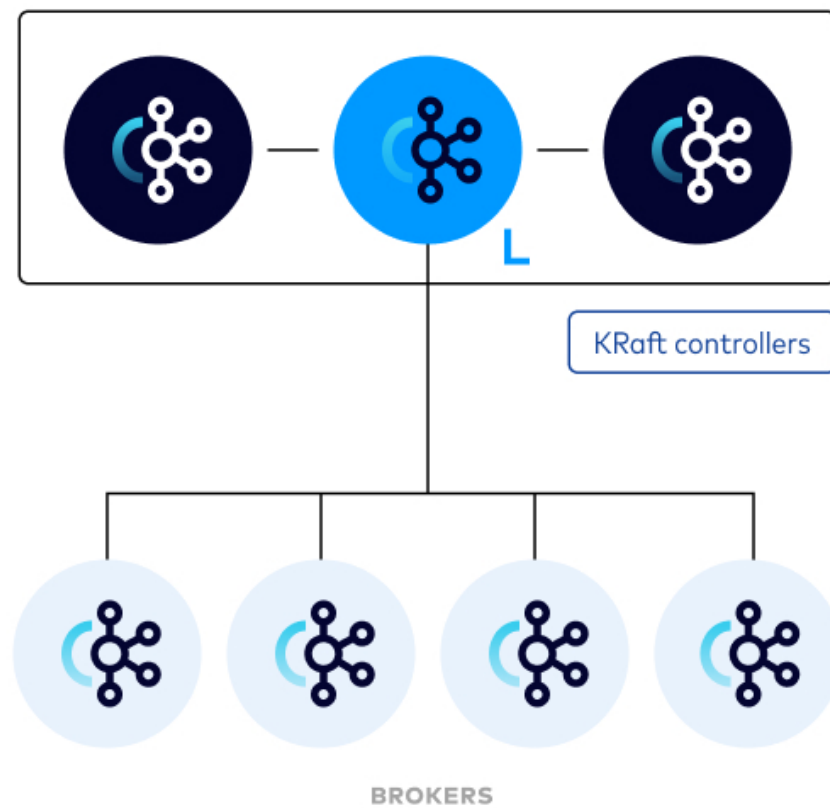
- **Rule of Majority:** A quorum is always a strict **majority** of the total number of nodes in the cluster (N).
- **Formula:** The minimum quorum size (Q) is calculated as: $Q = \lfloor N/2 \rfloor + 1$.

Quorum Examples

Total Nodes (N)	Quorum Size (Q)	Result
3	$\lfloor 3/2 \rfloor + 1 = 2$	Requires 2 votes to elect a Leader.
5	$\lfloor 5/2 \rfloor + 1 = 3$	Requires 3 votes to elect a Leader.
7	$\lfloor 7/2 \rfloor + 1 = 4$	Requires 4 votes to elect a Leader.



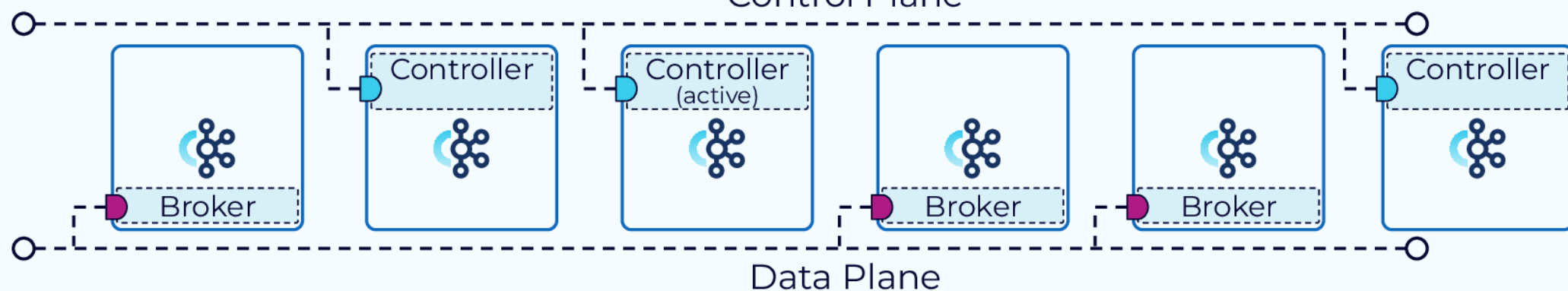
KRaft



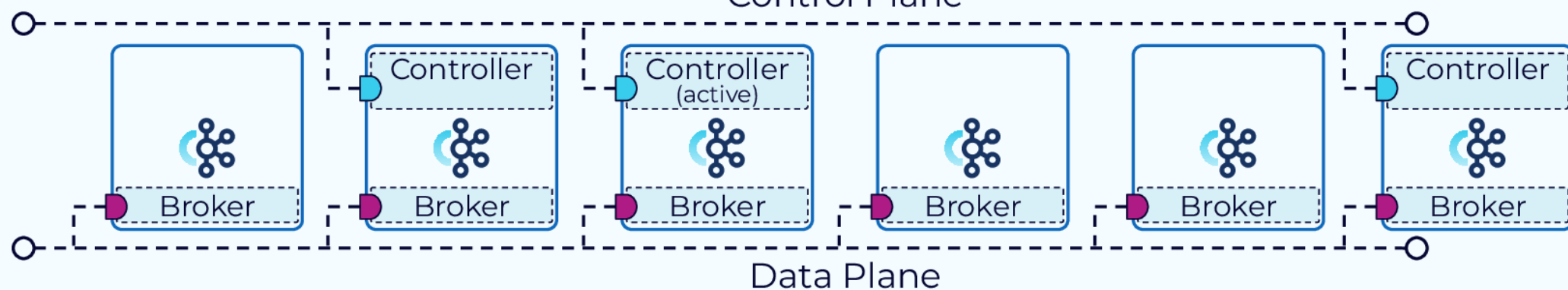
L denotes cluster metadata leader

Cluster node role can be **controller**
or **broker** or **controller, broker**

Kafka Cluster



Kafka Cluster



Two Types of Replication in Kafka

Type of replication	Who manages it?	What is replicated?	Where stored?
Metadata replication	KRaft Controller quorum (Raft)	Topics, partitions, configs, ACLs, ISR list	Metadata log (<code>__cluster_metadata</code>)
Data replication	Leader broker → follower brokers	User messages in partitions	Log segments on brokers

Step-by-Step Explanation

1 Broker starts and connects to the controller quorum

Broker sends a request:

```
arduino
```

```
→ "Give me cluster metadata"
```

Controller replies with metadata log state, including:

- Cluster ID
- List of brokers
- Partition → leader mapping
- Replica list for each partition

2 Broker stores this metadata in memory

Each broker maintains an internal metadata cache:

Example:

```
makefile
```

```
Topic: payments
```

```
Partition: 3
```

```
Leader: Broker-2
```

```
Replicas: [Broker-2, Broker-5, Broker-7]
```

```
ISR: [Broker-2, Broker-5, Broker-7]
```

So if **Broker-2 is leader**, it now knows:

```
CSS
```

```
→ I must replicate data to Broker-5 and Broker-7
```

3 Followers connect to the leader

Followers **don't wait for the leader to discover them** — they **pull** data from the leader.

Follower loop:

```
nginx
```

```
Follower → Leader:
```

```
"Give me new log entries since offset X"
```

So data replication is **pull-based**, not push-based.

4 Heartbeats + replication status is reported back to the controller

Example:

```
nginx
```

```
Follower → Controller:
```

```
"I'm alive, replication offset=10592"
```

This helps controller maintain ISR (In-Sync Replica list).

Client Producer

write

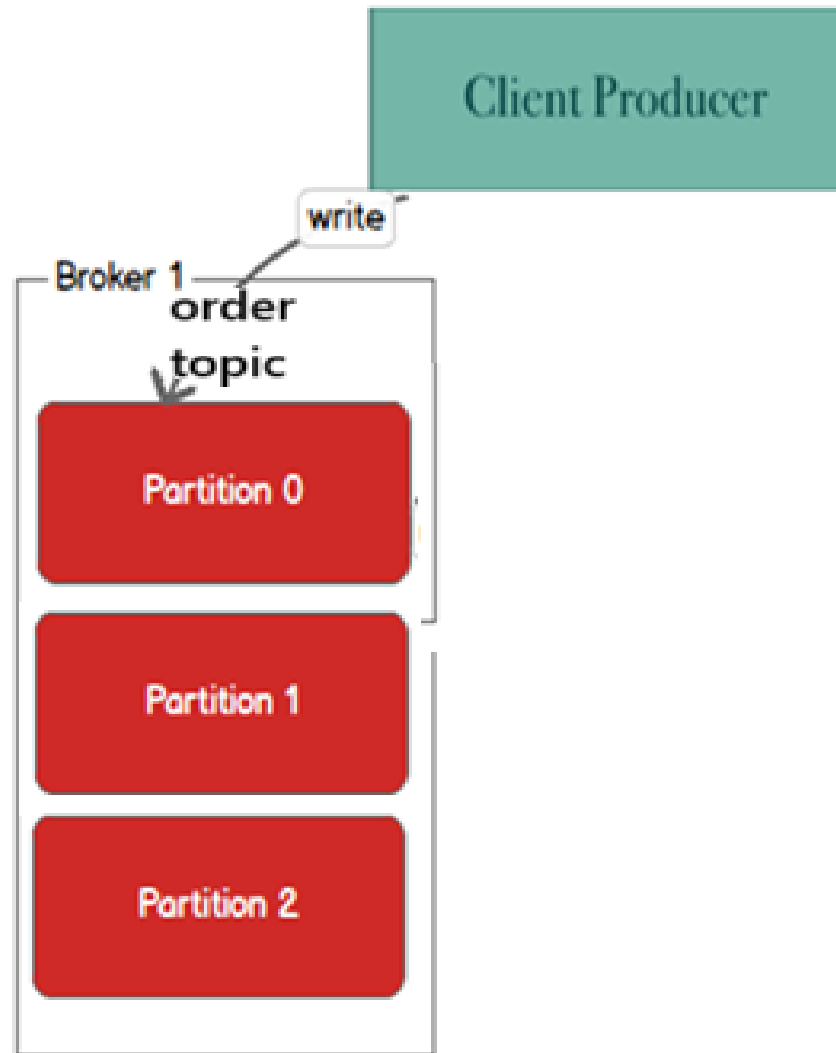
Broker 1

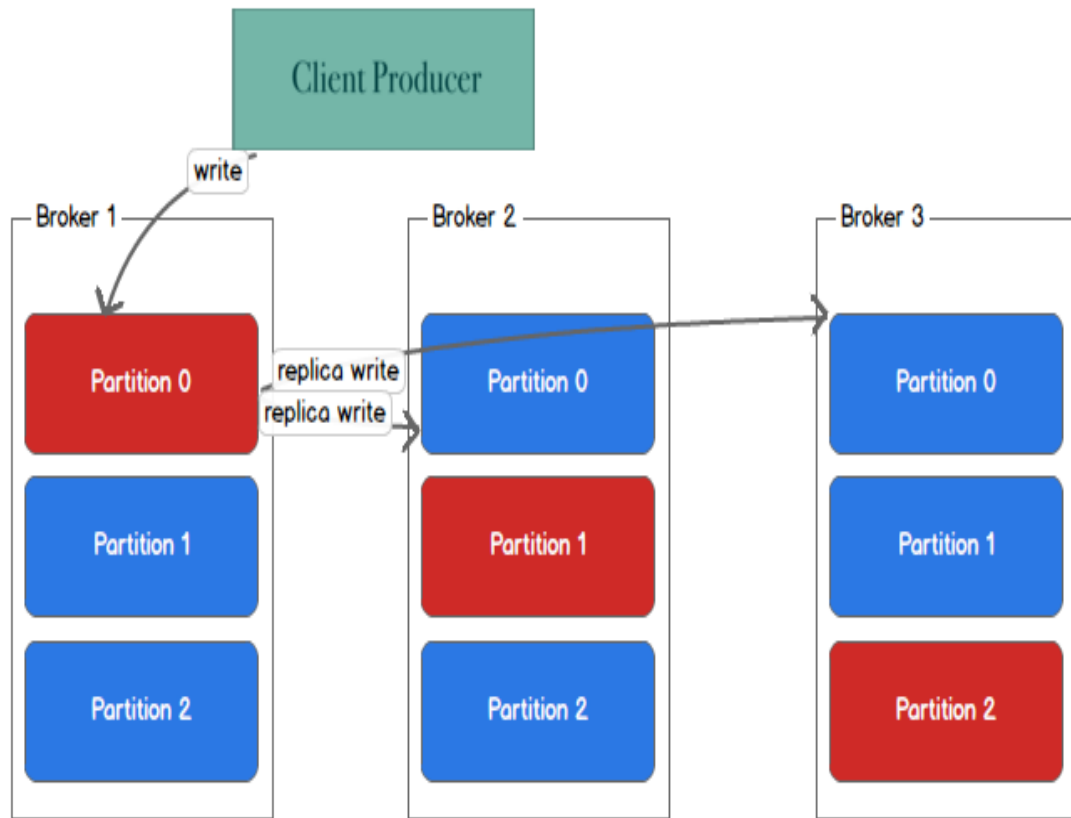
order
topic

Partition 0

Partition 1

Partition 2

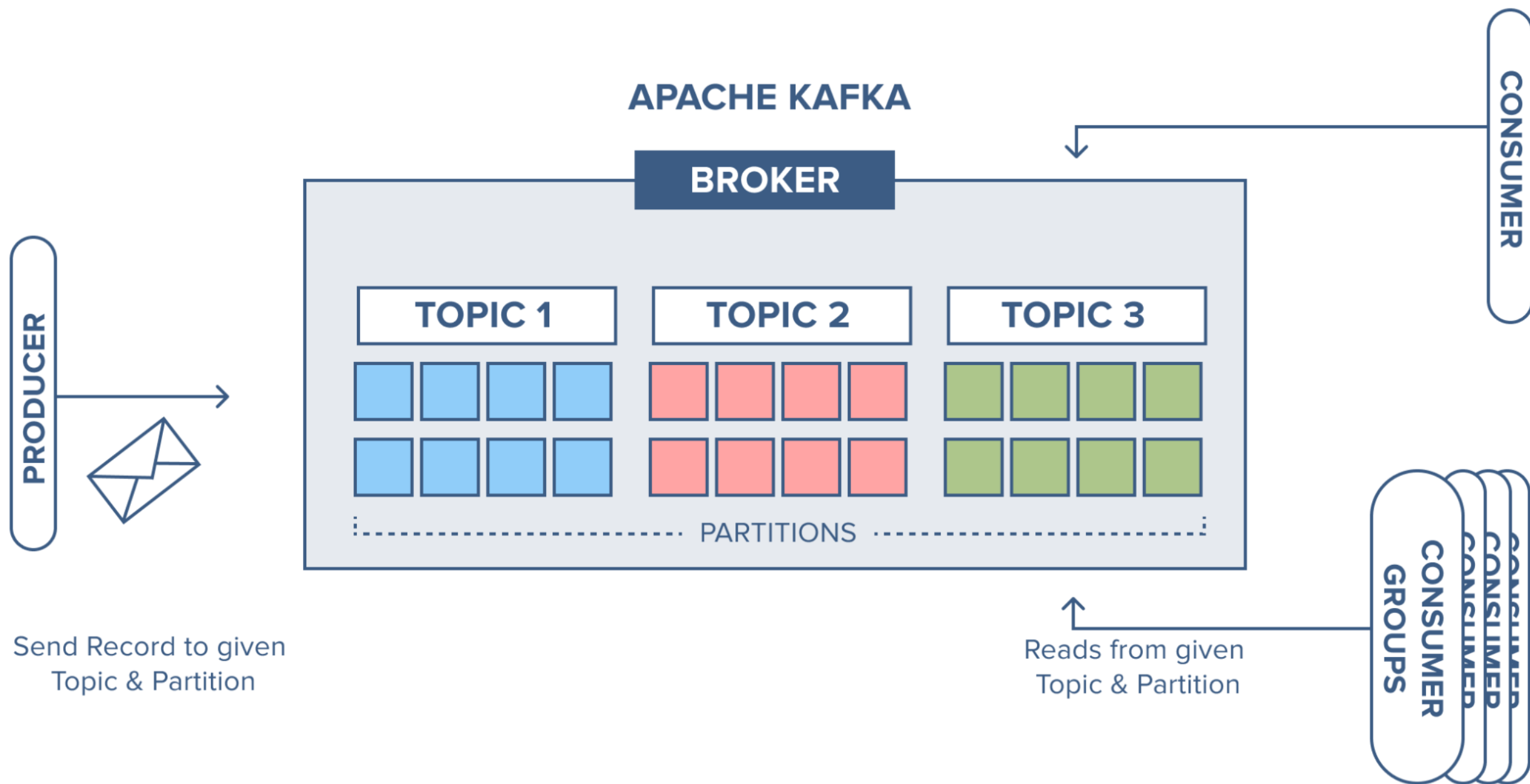




Leader (red)
replicas (blue)

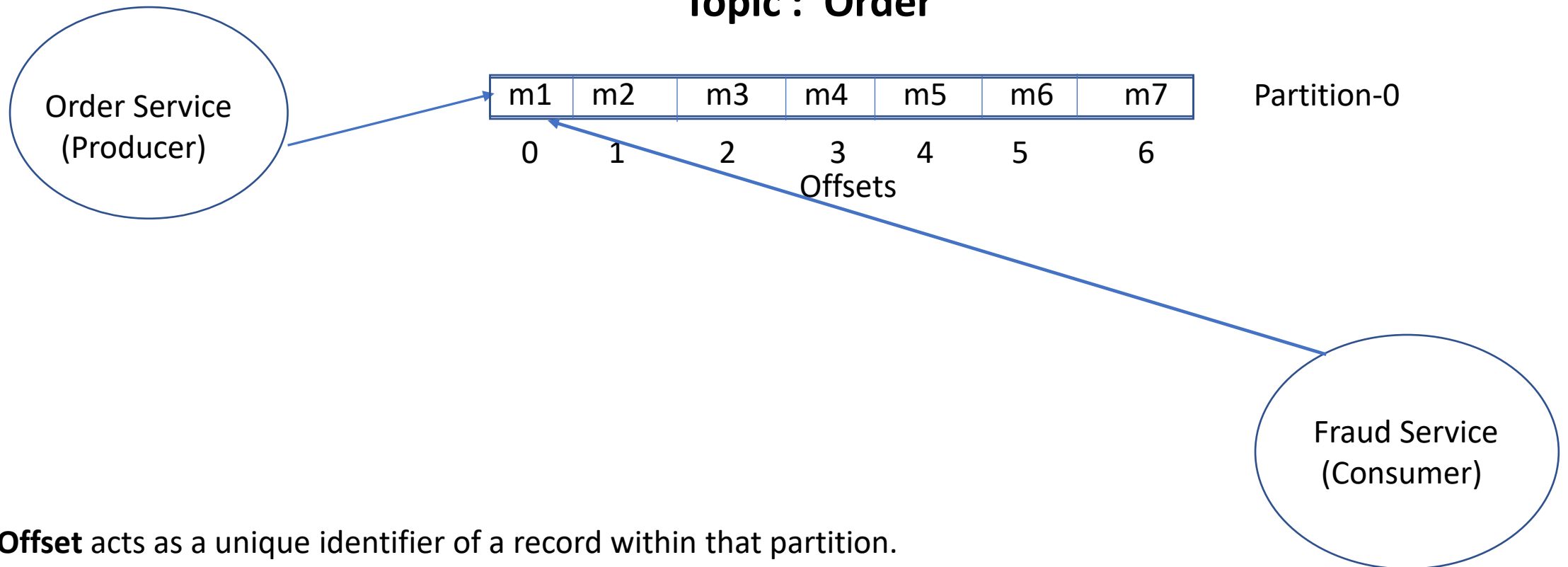
Record is considered "committed"
when all ISR's for partition
wrote to their log.

**Only committed records are
readable from consumer**



Kafka-Broker

Topic : Order



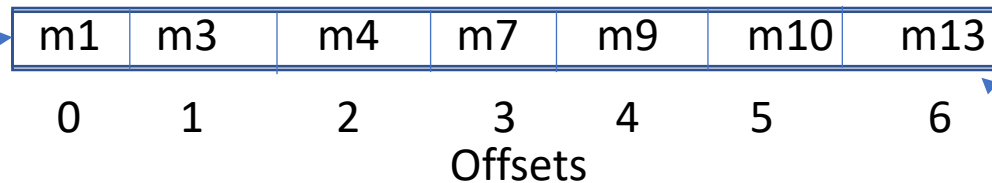
Offset acts as a unique identifier of a record within that partition.

Kafka-Broker

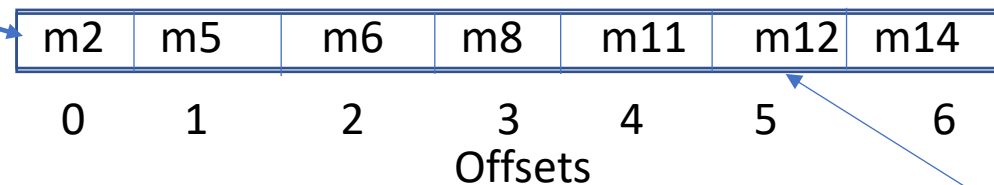
Topic : Order

Order Service
(Producer)

Partition 0



Partition 1



Fraud Service
(Consumer)

Message Partition

m1	->	0
m2	->	1
m3	->	0
m4	->	0
m5	->	1
m6	->	1

Kafka-Broker

Topic : Order

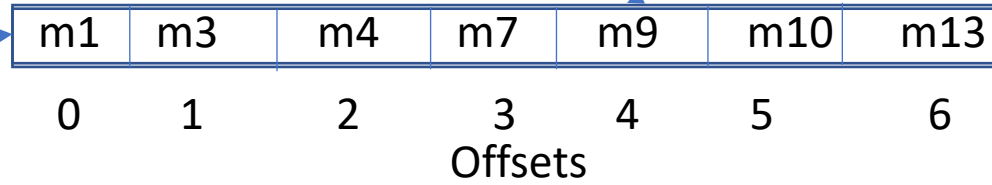
FSG

Fraud Service
(Consumer1)

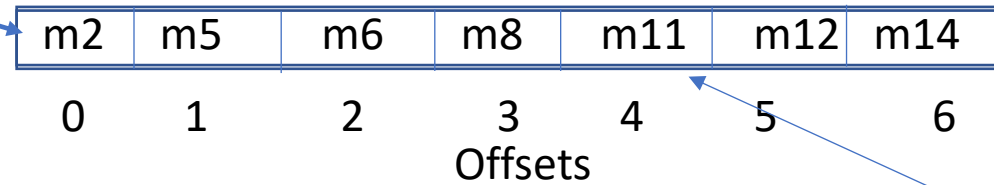
Fraud Service
(Consumer2)

Order Service
(Producer)

Partition 0



Partition 1



Message Partition

m1	->	0
m2	->	1
m3	->	0
m4	->	0
m5	->	1
m6	->	1

Kafka-Broker

Topic : Order

Order Service
(Producer)

Partition 0

m1	m3	m4	m7	m9	m10	m13
0	1	2	3	4	5	6
Offsets						

Partition 1

m2	m5	m6	m8	m11	m12	m14
0	1	2	3	4	5	6
Offsets						

Message Partition

m1	->	0
m2	->	1
m3	->	0
m4	->	0
m5	->	1
m6	->	1

FSG

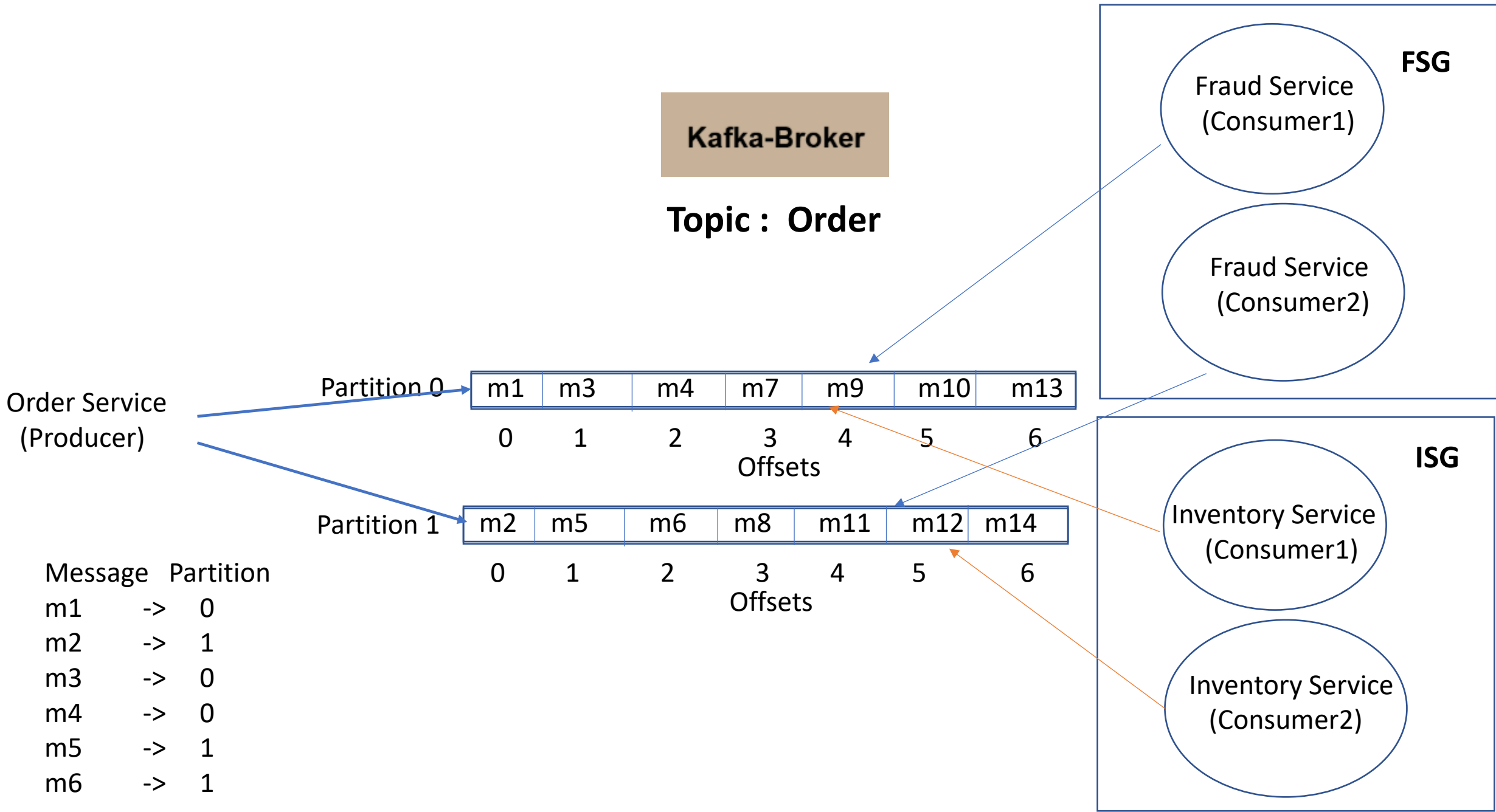
Fraud Service
(Consumer1)

Fraud Service
(Consumer2)

?

Fraud Service
(Consumer3)





Order-0

```
|_ segment-file(00000000000.log)
      |_ offsets  0  1  2  3 .....  9
|_ segment-file(00000000010.log)
      |_ offsets 10 11 12 ..... 19
|_ segment-file(00000000020.log)
|_ .....
```

Order-1

```
|_ segment-file(00000000000.log)
      |_ offsets  0  1  2  3 .....  9
|_ segment-file(00000000010.log)
      |_ offsets 10 11 12 ..... 19
|_ segment-file(00000000020.log)
|_ .....
```

Order-2

```
|_ segment-file(00000000000.log)
      |_ offsets  0  1  2  3 .....  9
|_ segment-file(00000000010.log)
      |_ offsets 10 11 12 ..... 19
|_ segment-file(00000000020.log)
```

* segment file default size is 1gb

segment-ms default time is 7 days

```
+-----+
| Order Service      |
| (Producer)        |
+-----+
```

|
v

```
+-----+
| Kafka Topic: orders |
|-----/
| Partition 0         |
| Partition 1         |
| Partition 2         |
+-----+
```

/

\

Payment Service
Consumer Group

```
+-----+
| P-Instance 1 |
| P-Instance 2 |
| P-Instance 3 |
+-----+
```

Inventory Service
Consumer Group

```
+-----+
| I-Instance 1 |
| I-Instance 2 |
| I-Instance 3 |
+-----+
```

```
+-----+
| Order Service |
| (Producer)    |
| [Docker Pod]  |
+-----+
```

|

v

```
+-----+
| Kafka Topic: orders |
|-----/
| Partition 0         |
| Partition 1         |
| Partition 2         |
+-----+
```

Payment Service
Consumer Group

```
+-----+
| Kubernetes Deployment (Payment) |
|-----/
| P-Instance 1 [Docker Pod] |
| P-Instance 2 [Docker Pod] |
| P-Instance 3 [Docker Pod] |
+-----+
```

Inventory Service
Consumer Group

```
+-----+
| Kubernetes Deployment (Inventory) |
|-----/
| I-Instance 1 [Docker Pod] |
| I-Instance 2 [Docker Pod] |
| I-Instance 3 [Docker Pod] |
+-----+
```



I have processed the messages from 0-3 offset successfully

Is there any way I can process remaining messages at a later time.

Kafka-Broker

Yes...

Shall I auto commit the offset; so that next time when you come back,
I can give you other messages?

(or) you can tell me to commit.

Great offset is committed!



Oh God...some thing went wrong.

Is there any way, I can once again read the messages from the offset 2?

Kafka-Broker

Yes...

You can read the messages from beginning, end or any position.

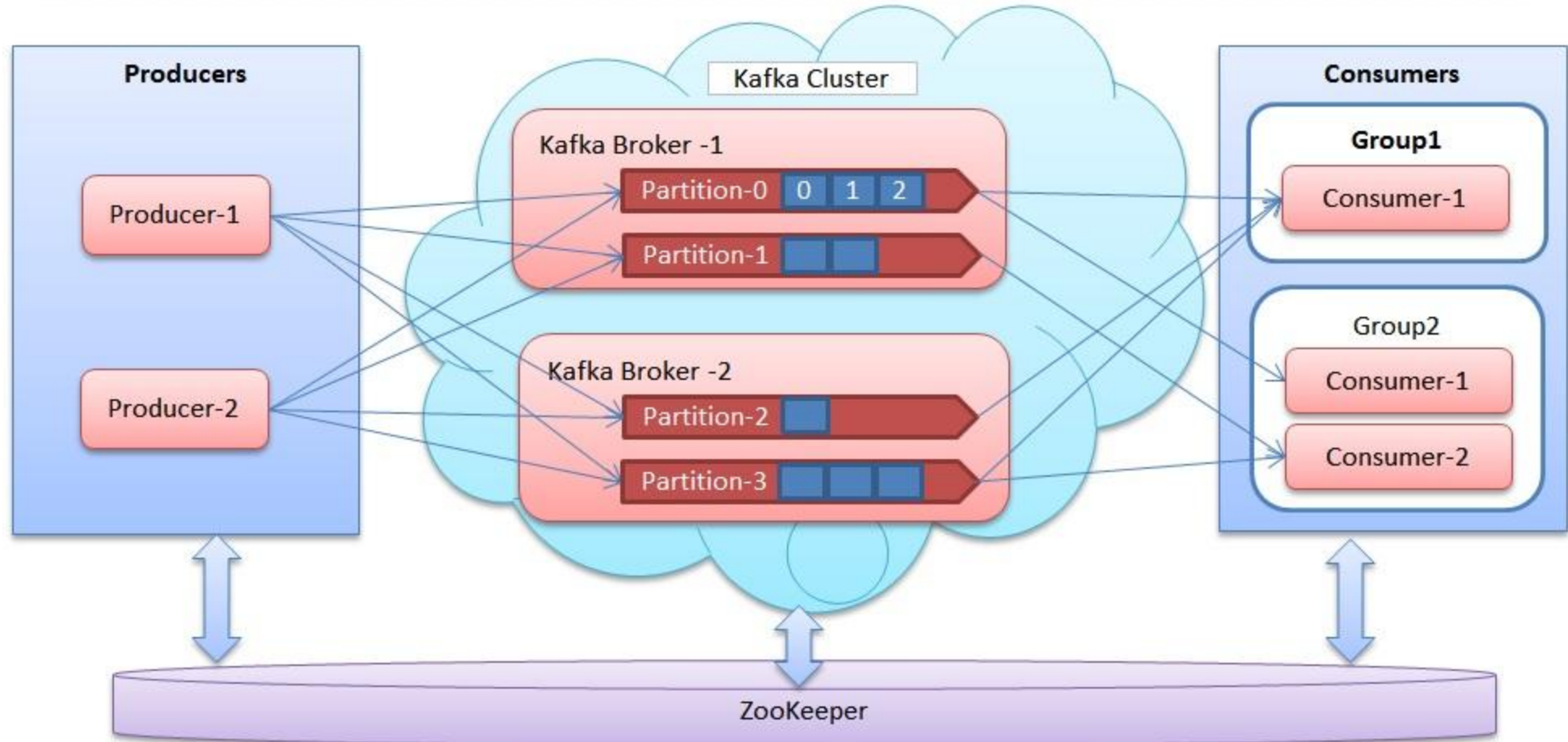
Not only now, any time. Any number of times you can read the Messages.

Kafka is a

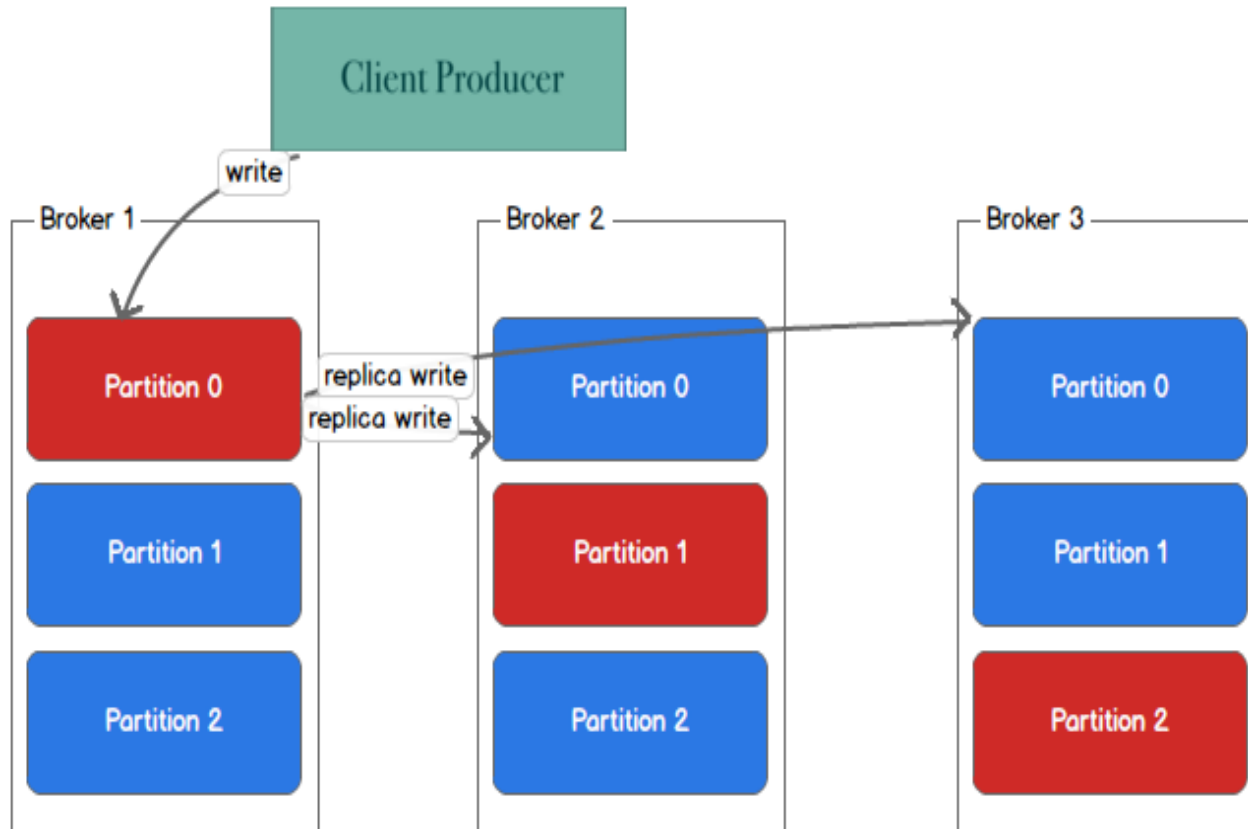
- ❑ Distributed
- ❑ partitioned,
- ❑ replicated commit log service.

It provides the functionality of a messaging system, but with a unique design.

In this diagram, two brokers are shown in a kafka cluster. Two producers are publishing messages to one topic with 4 partitions. Each broker have two partitions. Two consumer groups are configured, group1 has one consumer and group2 has two consumers. For each consumer group, messages are guaranteed to be consumed at least once.



Kafka Topic Architecture - Replication, Failover, and Parallel Processing



Leader (red)
replicas (blue)

Record is considered "committed"
when all ISRs for partition
wrote to their log.

**Only committed records are
readable from consumer**

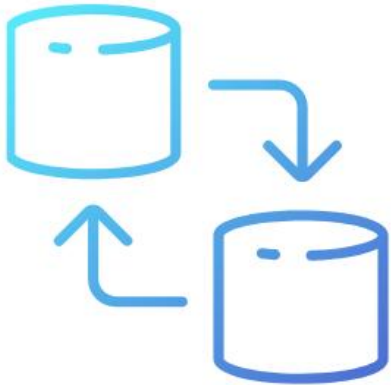
- ❑ Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day.
- ❑ Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log.
- ❑ Since being created and open sourced by LinkedIn in 2011, Kafka has quickly evolved from messaging queue to a full-fledged event streaming platform.

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Publish + Subscribe

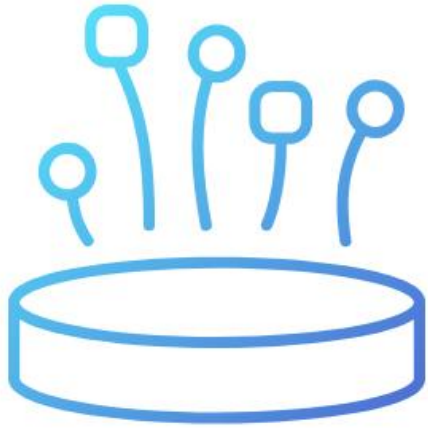
It is immutable commit log, and from there we can subscribe to it, and publish data to any number of systems or real-time applications.



Unlike messaging queues, Kafka is a highly scalable, fault tolerant distributed system, allowing it to be deployed for applications like:

- ❑ managing passenger and driver matching at Uber,
- ❑ providing real-time analytics and predictive maintenance for British Gas' smart home,
- ❑ and performing numerous real-time services across all of LinkedIn.

This unique performance makes it perfect to scale from one app to company-wide use.



Store

An abstraction of a distributed commit log commonly found in distributed databases(Oracle SGA), Apache Kafka provides durable storage.

Kafka can act as a 'source of truth', being able to distribute data across multiple nodes for a highly available deployment within a single data center or across multiple availability zones.

* single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is stored exactly once.



Process

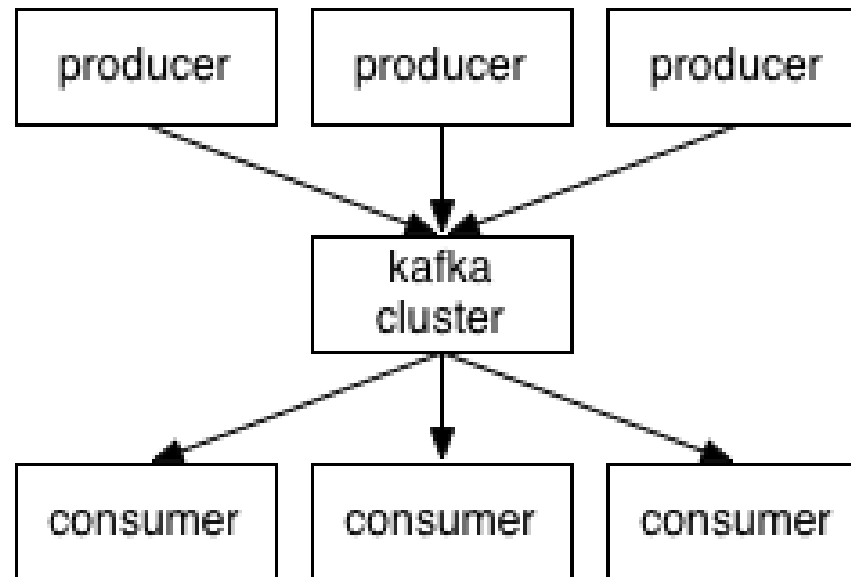
An event streaming platform would not be complete without the ability to manipulate that data as it arrives.

The Streams API within Apache Kafka is a powerful, lightweight library that allows for on-the-fly processing, letting you aggregate, create windowing parameters, perform joins of data within a stream, and more.

Perhaps best of all, it is built as a Java application on top of Kafka, keeping your workflow intact with no extra clusters to maintain.

Some basic messaging terminology:

- ❑ Kafka maintains feeds of messages in categories called topics.
- ❑ Producers publish the messages to a Kafka topic.
- ❑ Consumers will subscribe to topics and process the messages.
- ❑ Kafka cluster consists of one or more servers (Kafka brokers).



Kafka Use cases :

- ✓ **Website Activity Tracking**
- ✓ **Metrics**
- ✓ **Log Aggregation**
- ✓ **Stream Processing**
- ✓ **Event Sourcing**

Use Cases

Here is a description of a few of the popular use cases for Apache Kafka.

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc).

In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type.

These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Kafka can be used as a replacement for a log aggregation solution.

Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing.

Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption.

In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency

Stream Processing

Many users end up doing stage-wise processing of data where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka topics for further consumption.

For example a processing flow for article recommendation might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might help normalize or deduplicate this content to a topic of cleaned article content; a final stage might attempt to match this content to users.

This creates a graph of real-time data flow out of the individual topics.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log

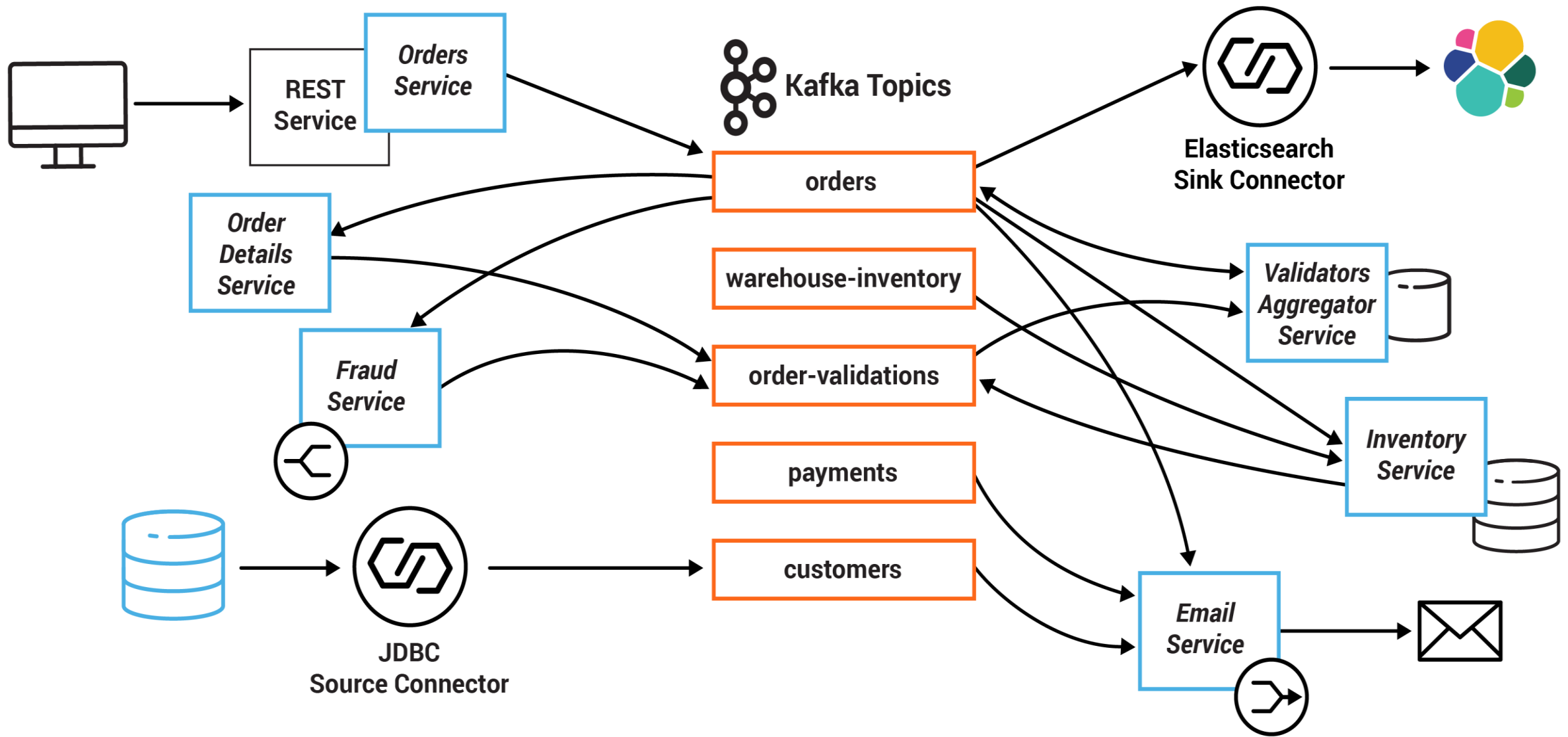
Event Sourcing

Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Event sourcing persists the state of a business entity such as an Order or a Customer as a sequence of state-changing events.

This is used in microservice architecture for distributed transactions.

Use case: Ecommerce-order-processing



In this example, the system centres on an **Orders Service** which exposes a REST interface to POST and GET Orders.

Posting an Order creates an event in Kafka that is recorded in the topic orders.

This is picked up by different validation engines (**Fraud Service, Inventory Service and Order Details Service**), which validate the order in parallel, emitting a PASS or FAIL based on whether each validation succeeds.

The result of each validation is pushed through a separate topic: Order Validations, so that we retain the single writer status of the Orders Service.

The results of the various validation checks are aggregated in the **Validation Aggregator Service**, which then moves the order to a Validated or Failed state, based on the combined result.

There is a simple service that sends emails, and another that collates orders and makes them available in a search index using Elasticsearch.

Apache ZooKeeper

ZooKeeper is a distributed co-ordination service to manage large set of hosts.

Co-ordinating and managing a service in a distributed environment is a complicated process.

ZooKeeper solves this issue with its simple architecture and API.

What is Apache ZooKeeper Meant For?

Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintain shared data with robust synchronization techniques.

ZooKeeper is itself a distributed application providing services for writing a distributed application.

The common services provided by ZooKeeper are as follows –

Naming service – Identifying the nodes in a cluster by name. It is similar to DNS, but for nodes.

Configuration management – Latest and up-to-date configuration information of the system for a joining node.

Cluster management – Joining / leaving of a node in a cluster and node status at real time.

Leader election – Electing a node as leader for coordination purpose.

Locking and synchronization service – Locking the data while modifying it. This mechanism helps you in automatic fail recovery while connecting other distributed applications like Apache HBase.

Highly reliable data registry – Availability of data even when one or a few nodes are down.

Why Is Kafka So Fast

Avoid Random Disk Access

Kafka writes everything onto the disk in order and consumers fetch data in order too. So disk access always works sequentially instead of randomly.

For traditional hard disks(HDD), sequential access is much faster than random access.

Here is a comparison:

hardware	sequential writes	random
writes		
6 * 7200rpm SATA RAID-5	300MB/s	50KB/s

Kafka Writes Everything Onto The Disk Instead of Memory

Kafka writes everything onto the disk instead of memory.

Isn't memory supposed to be faster than disks? Typically it's the case, for Random Disk Access. But for sequential access, the difference is much smaller.

But still, sequential memory access is faster than Sequential Disk Access, why not choose memory?

Because Kafka runs on top of JVM, which gives us two disadvantages.

- ❑ The memory overhead of objects is very high, often doubling the size of the data stored (or even higher).
- ❑ Garbage Collection happens every now and then, so creating objects in memory is very expensive as in-heap data increases because we will need more time to collect unused data.
- ❑ So writing to file systems may be better than writing to memory. Even better, we can utilize MMAP (memory mapped files) to make it faster.

Memory Mapped Files(MMAP)

Basically, MMAP(Memory Mapped Files) can map the file contents from the disk into memory. And when we write something into the mapped memory, the OS will flush the change onto the disk sometime later.

So everything is faster because we are using memory actually, but in an indirect way.

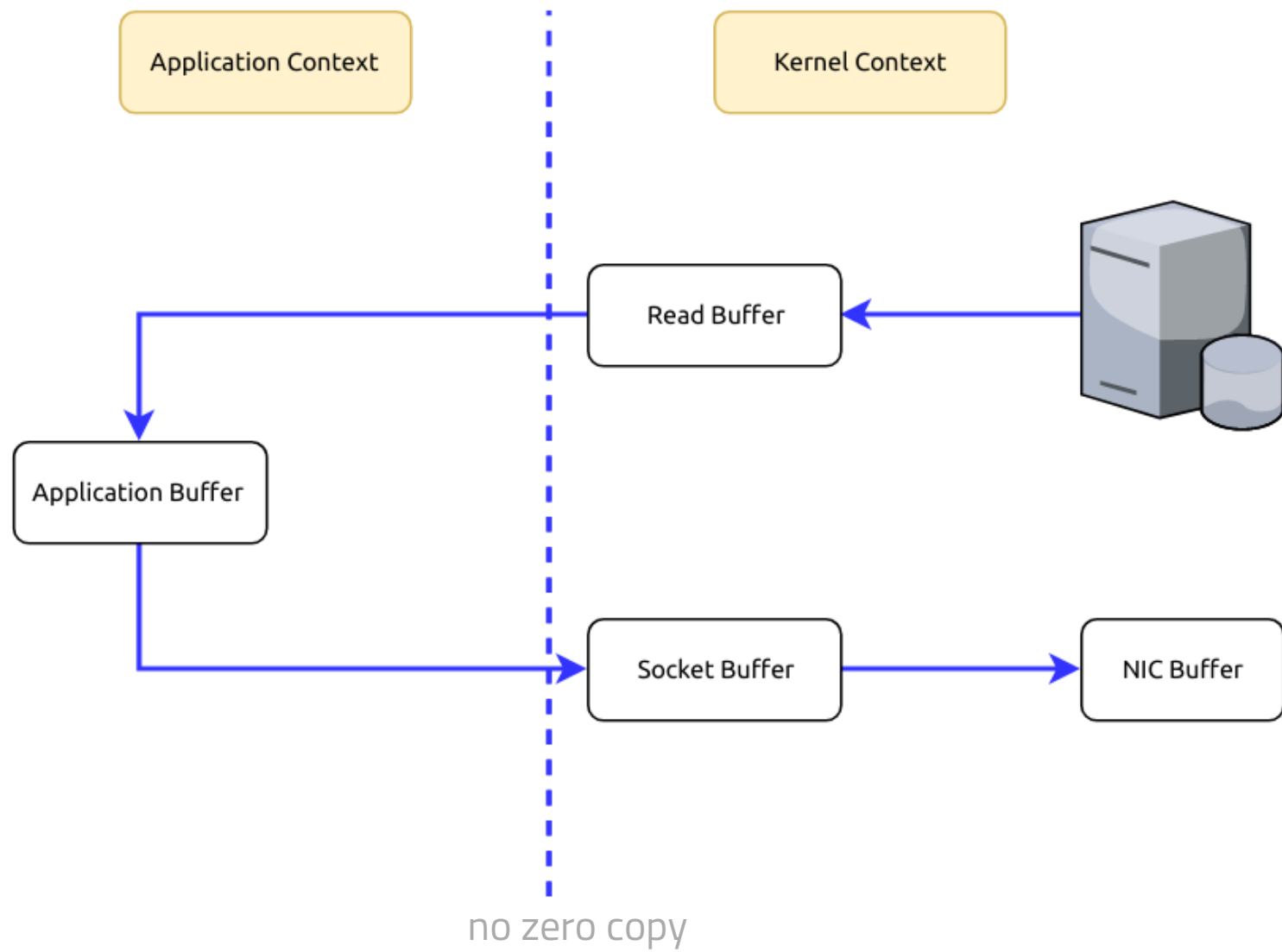
No Zero Copy

Suppose that we are fetching data from the memory and sending them to the Internet.

What is happening in the process is usually twofold.

To fetch data from the memory, we need to copy those data from the Kernel Context into the Application Context.

To send those data to the Internet, we need to copy the data from the Application Context into the Kernel Context.

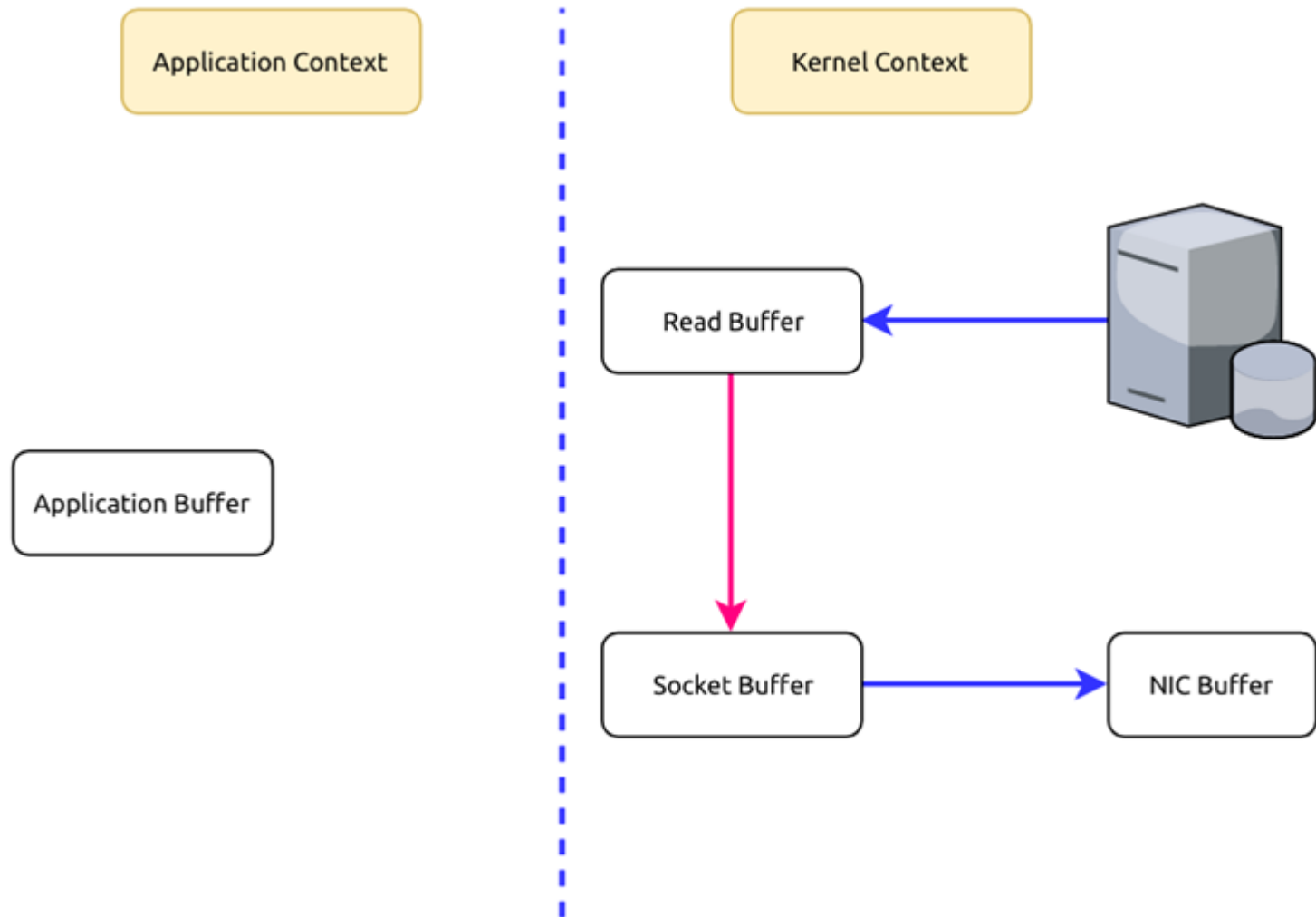


Zero Copy

As you can see, it's redundant to copy data between the Kernel Context and the Application Context.

Can we avoid it? Yes,

using Zero Copy we can copy data directly from the Kernel Context to the Kernel Context.



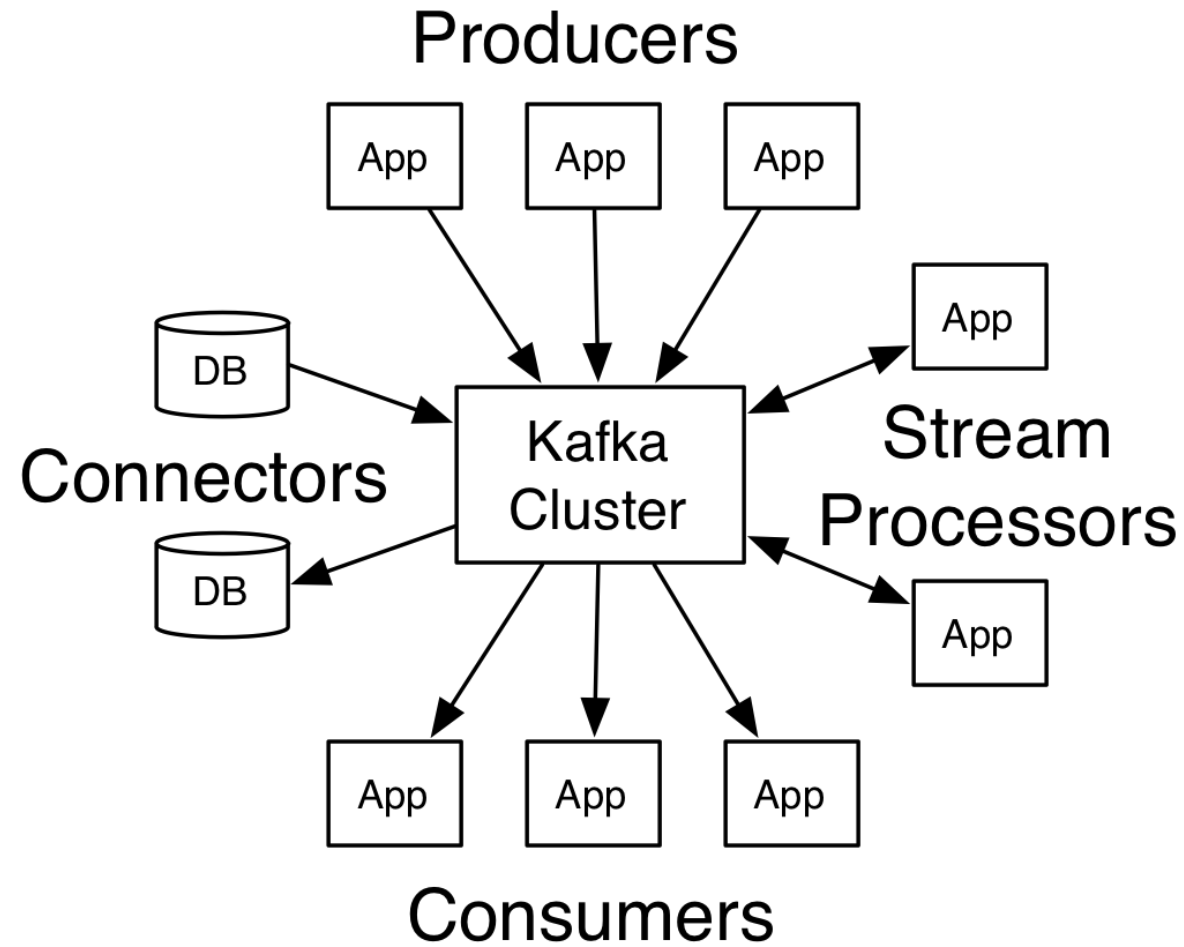
zero copy

Batch Data

Kafka only sends data when batch.size is reached instead of one by one. Assuming the bandwidth is 10MB/s, sending 10MB data in one go is much faster than sending 10000 messages one by one (assuming each message takes 100 bytes).

Kafka core APIs:

Kafka has four core APIs:



❑ **The Producer API** allows an application to publish a stream of records to one or more Kafka topics.

❑ **The Consumer API** allows an application to subscribe to one or more topics and process the stream of records produced to them.

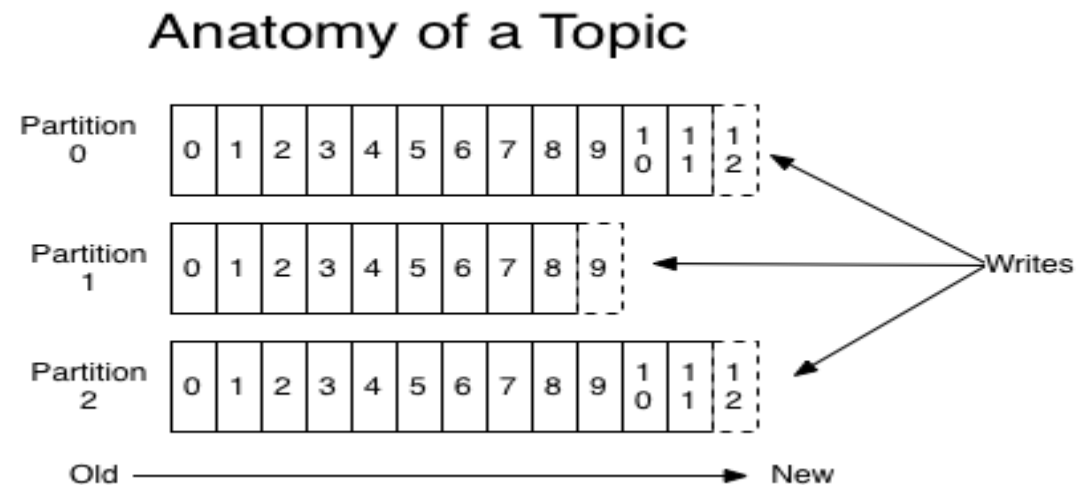
❑ **The Streams API** allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

❑ **The Connector API** allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

For example, a connector to a relational database might capture every change to a table.

Topics and Logs

A topic is a category or feed name to which messages are published. For each topic, the Kafka cluster maintains a partitioned log that looks like this:



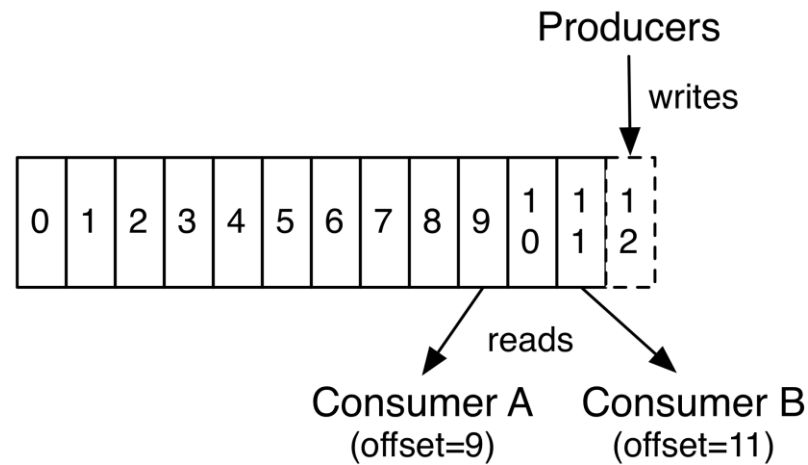
- ❑ Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log.
- ❑ The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each message within the partition.

- ❑ The Kafka cluster retains all published messages—whether or not they have been consumed—for a configurable period of time (default: 168 hours)

For example if the log retention is set to two days, then for the two days after a message is published it is available for consumption, after which it will be discarded to free up space.

- ❑ Kafka's performance is effectively constant with respect to data size so retaining lots of data is not a problem.

- ❑ In fact the only metadata retained on a per-consumer basis is the position of the consumer in the log, called the "offset".
- ❑ This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads messages, but in fact the position is controlled by the consumer and it can consume messages in any order it likes.



Note: This combination of features means that **Kafka consumers are very cheap**—they can come and go without much impact on the cluster or on other consumers.

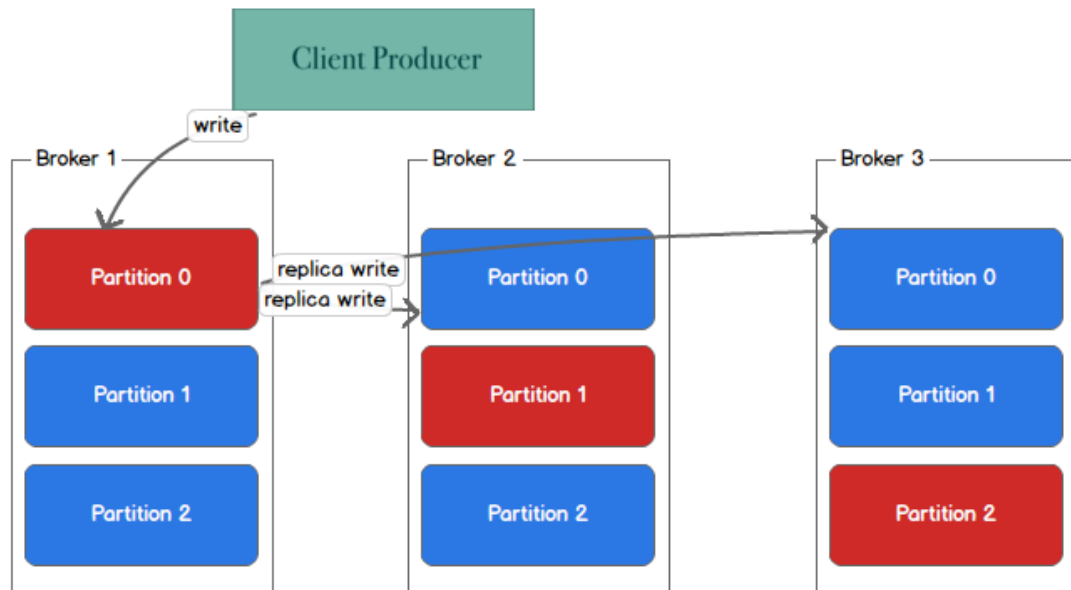
The partitions in the log serve several purposes.

First, they allow the log to scale beyond a size that will fit on a single server.

Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data.

Second they act as the **unit of parallelism**—more on that in a bit.

Kafka Topic Architecture - Replication, Failover, and Parallel Processing



Leader (red)
replicas (blue)

Record is considered "committed"
when all ISRs for partition
wrote to their log.

**Only committed records are
readable from consumer**

- ❑ The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions.
- ❑ Each partition is replicated across a configurable number of servers for fault tolerance.
- ❑ Each partition has one server which acts as the "leader" and zero or more servers which act as "followers".

- ❑ The leader handles all read and write requests for the partition while the followers passively replicate the leader.
- ❑ If the leader fails, one of the followers will automatically become the new leader.
- ❑ Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

auto.leader.rebalance.enable (default: true)

Enables auto leader balancing. A background thread checks the distribution of partition leaders at regular intervals, configurable by ``leader.imbalance.check.interval.seconds``.

If the leader imbalance exceeds ``leader.imbalance.per.broker.percentage`` (default: 10), leader rebalance to the preferred leader for partitions is triggered.

Producers

- ❑ Producers publish data to the topics of their choice.
- ❑ The producer is responsible for choosing which message to assign to which partition within the topic.
- ❑ This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (based on some key in the message).

Consumers

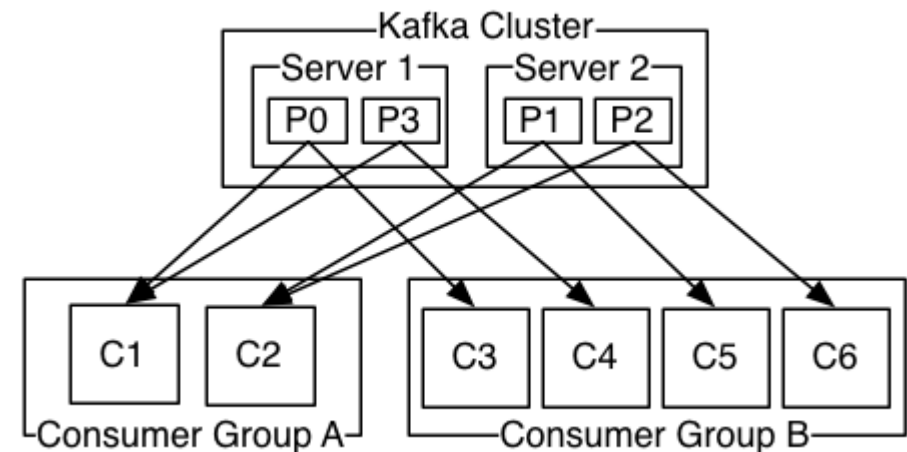
- ❑ Messaging traditionally has two models: queuing and publish-subscribe.
- ❑ In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers.
- ❑ Kafka offers a single consumer abstraction that generalizes both of these—the consumer group.

- ❑ Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.
- ❑ If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers.
- ❑ If all the consumer instances have different consumer groups, then this works like publish-subscribe and all messages are broadcast to all consumers.

Topics have a small number of consumer groups, one for each "logical subscriber".

Each group is composed of many consumer instances for scalability and fault tolerance.

This is nothing more than publish-subscribe semantics where the subscriber is cluster of consumers instead of a single process.



Comparisons with Traditional Message Queue

A **traditional queue** retains messages in-order on the server, and if multiple consumers consume from the queue then the server hands out messages in the order they are stored.

However, although the server hands out messages in order, the messages are delivered asynchronously to consumers, so they may arrive out of order on different consumers.

This effectively means the ordering of the messages is lost in the presence of parallel consumption.

Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is **no parallelism in processing**.

Kafka does it better. By having a notion of parallelism—the partition—within the topics, Kafka is able to provide both ordering guarantees and load balancing over a pool of consumer processes.

This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group.

By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances.

Note however that there cannot be more consumer instances than partitions.

- ❑ Kafka only provides a total order over messages within a partition, not between different partitions in a topic.
- ❑ Per-partition ordering combined with the ability to partition data by key is sufficient for most applications.
- ❑ However, if you require a total order over messages this can be achieved with a topic that has only one partition, though this will mean only one consumer process.

Guarantees

- ❑ At a high-level Kafka gives the following guarantees:
- ❑ Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a message $M1$ is sent by the same producer as a message $M2$, and $M1$ is sent first, then $M1$ will have a lower offset than $M2$ and appear earlier in the log.
- ❑ A consumer instance sees messages in the order they are stored in the log.
- ❑ For a topic with replication factor N , we will tolerate up to $N-1$ server failures without losing any messages committed to the log.

Topic configurations

Brokers have defaults for all the topic configuration parameters

These parameters impact performance and topic behavior

Some topics may need different values than the defaults

- > Replication Factor
- > no of partitions
- > Message size
- > Compression level
- > Log cleanup policy
- > Min Insync Replicas
- >

Refer to : <https://kafka.apache.org/documentation/#brokerconfigs>

**LET'S
PRACTICE**

01_Kafka_Lab

Graceful shutdown

Graceful shutdown

The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine.

This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes.

When a server is stopped gracefully it has two optimizations it will take advantage of:

It will sync all its logs to disk to avoid needing to do any log recovery when it restarts (i.e. validating the checksum for all messages in the tail of the log). Log recovery takes time so this speeds up intentional restarts.

It will migrate any partitions the server is the leader for to other replicas prior to shutting down. This will make the leadership transfer faster and minimize the time each partition is unavailable to a few milliseconds.

Syncing the logs will happen automatically whenever the server is stopped other than by a hard kill, but the controlled leadership migration requires using a special setting:

```
controlled.shutdown.enable=true
```

Note that controlled shutdown will only succeed if all the partitions hosted on the broker have replicas (i.e. the replication factor is greater than 1 and at least one of these replicas is alive).

LET'S PRACTICE

Graceful Shutdown

\$ kafka-server-stop (stop all the kafka instances)

```
D:\MST\software\kafka\kafka_2.12-1.1.0\bin\windows>kafka-server-stop.bat
Deleting instance \\DESKTOP-P5DJBAL\ROOT\CIMV2:Win32_Process.Handle="16088"
Instance deletion successful.
Deleting instance \\DESKTOP-P5DJBAL\ROOT\CIMV2:Win32_Process.Handle="38304"
Instance deletion successful.
```

Partitions, Segments & Indexes

Partitions and Segments

Topics are made of Partitions

Partitions are made of segments (files)

Ex :

Partition-0		
segment 0:	segment: 1	segment: 2
Offset 0-200	Offset 201-400	Offset 401-? (Active)

Note : Only one segment is ACTIVE per partition

Segment and Indexes

Segments come with two indexes (files)

- > An offset to position index: allows Kafka where to read to find a message
- > A timestamp to offset index: allows Kafka to find messages with a timestamp

This will help Kafka to find the data in a constant time.

Segment 0		Segment 2		Segment 4		Segment 6		Segment 8	
Msg0	Msg1	Msg2	Msg3	Msg4	Msg5	Msg6	Msg7	Msg8	Msg9

testapp-2

Index File		Log File			
Offset	Position	Offset	Position	Time	Message
0	0	0	0	1222333333333	Faith
1	200	1	200	1222233333444	Trust

log.segment.bytes : the max size of a single segment in bytes (default: 1GB)

A smaller log.segment.bytes means :

- More segments per partitions

- Log compaction happens more often

- But Kafka has to keep more files opened (Error: Too many open files)

log.segment.ms: the time kafka will wait before committing if the segment is not full (default: 1 week)

A smaller log.segment.ms means:

You set a max frequency for log compaction (more frequent triggers)

Maybe you want daily compaction instead of weekly?

Log Cleanup Policies

Kafka data will get expired, based on the policies:

Policy 1 : `log.cleanup.policy=delete` (default for all topics)

- > Delete based on age of data (Default is a week)
- > Delete based on max size of log (default -1 i.e. infinite)

Policy 2: `log.cleanup.policy=compact` (Kafka default for `topic_Consumer_offsets`)

- > Delete based on keys of your messages
- > will delete old duplicate keys after that active segment is committed

log.retention.hours

number of hours to keep data for (default is 168 hrs / one week)

log.retention.bytes

max size in bytes for each partition [default is infinite (-1)]

Realtime usecases:

1. One week of retention:

log.retention.hours=168 and log.retention.bytes=-1

2. Infinite time retention bounded by 500MB

log.retention.hours=19999 and log.retention.bytes=524288000

Compaction takes the snap shot of the latest value to the key and whenever the failure of happens consumer recovers from the compacted logs as shown below.

Kafka Log Compaction Process

Before Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

Cleaning

Only keeps latest version of key. Older duplicates not needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After Compaction

**LET'S
PRACTICE**

02_Kafka_Lab

Producers, Consumers & Consumer Groups

Producers

- ❑ write data to topics (i.e., partitions)
- ❑ Producer automatically know to which broker and partition to write to
- ❑ In case of Broker failure, producers will automatically recover

Producers can choose to receive acknowledgement of data writes:

- ❑ acks=0: Producer won't wait for acknowledgment (chances of losing the data)
- ❑ acks=1: Producer will wait for leader acknowledgment (limited data loss)
- ❑ acks=all : Leader + replicas acknowledgment (no data loss)

Producers : Message Keys

- ☐ Producers can choose to send a key with message (string, number etc.,)
- ☐ If key=null, data is sent round robin to brokers
- ☐ If a key is set, then all messages for that key will always go to the same partition
- ☐ A key is basically sent if you need message ordering for a specific field (ex: truck_id)

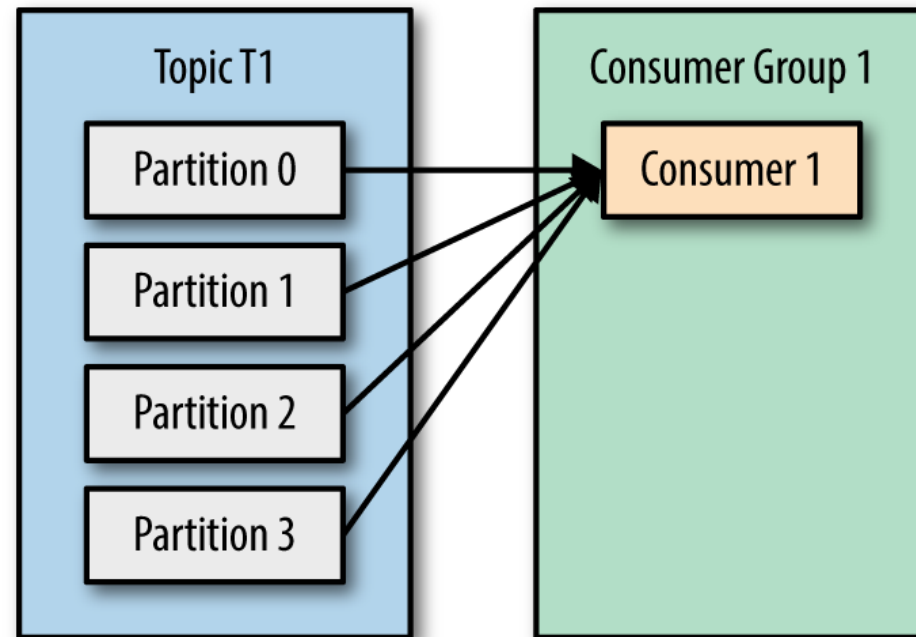
Consumers

- ❑ consumers read data from a topic
- ❑ consumers know which broker to read from
- ❑ In case of broker failures, consumers know how to recover
- ❑ Data is read in order within each partitions

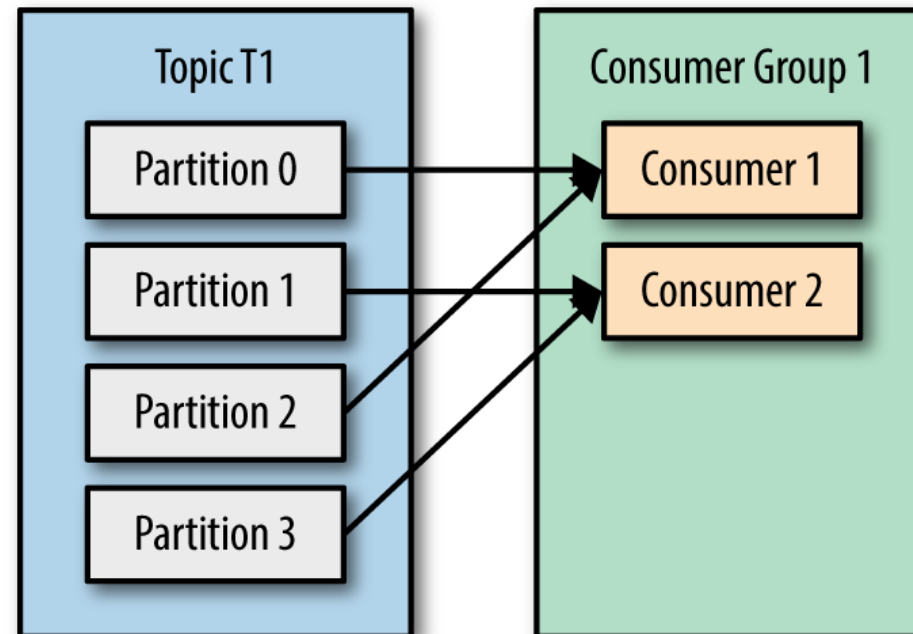
Consumer Groups

- ❑ consumers read data in consumer groups
- ❑ each consumer within a group reads from exclusive partitions
- ❑ if you have more consumers than partitions, some consumers will be inactive

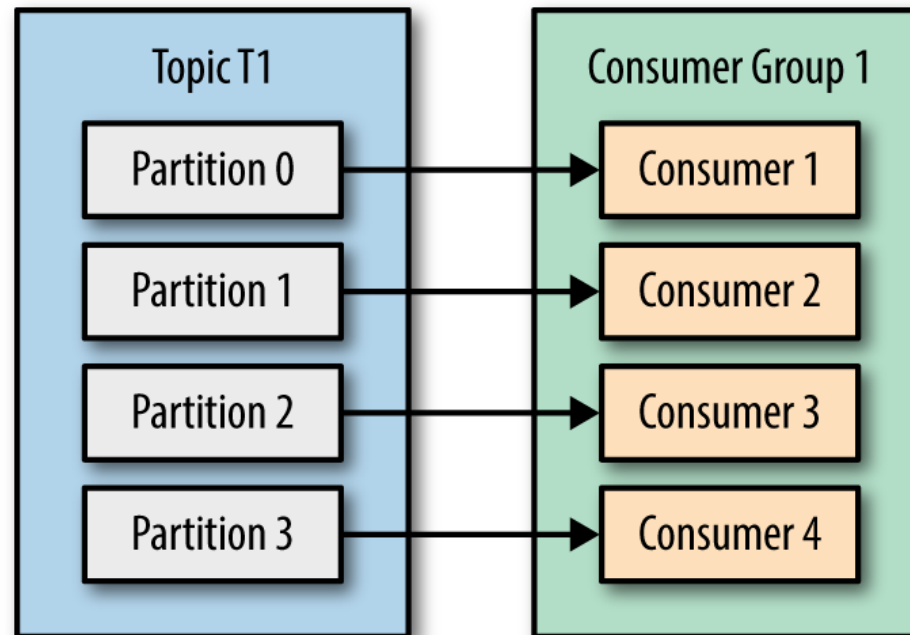
One Consumer group with four partitions



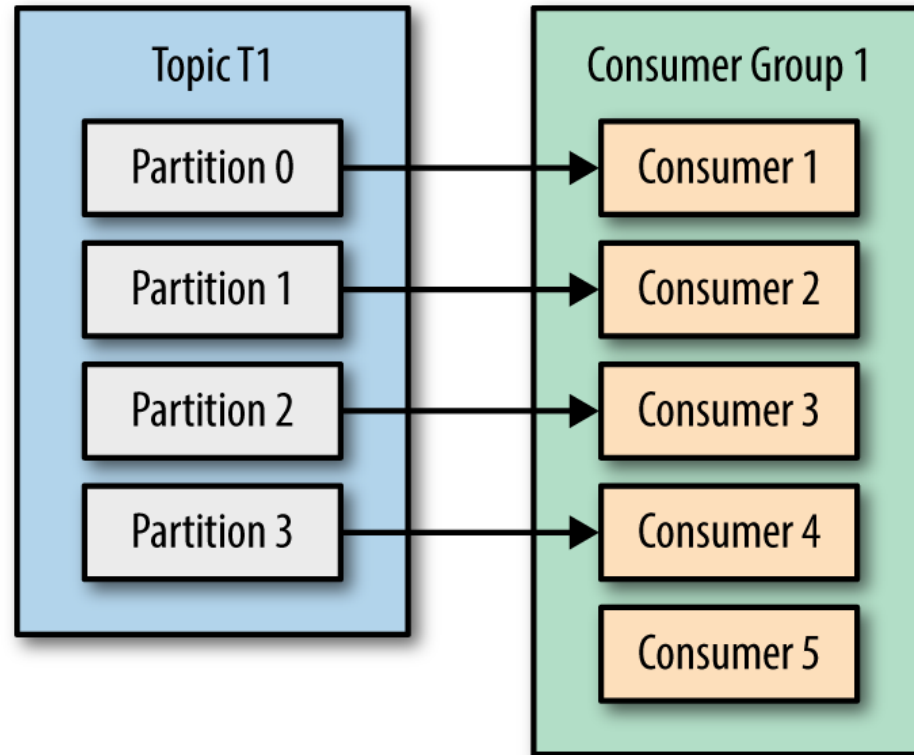
Four partitions split to two consumer groups



Four consumer groups to one partition each

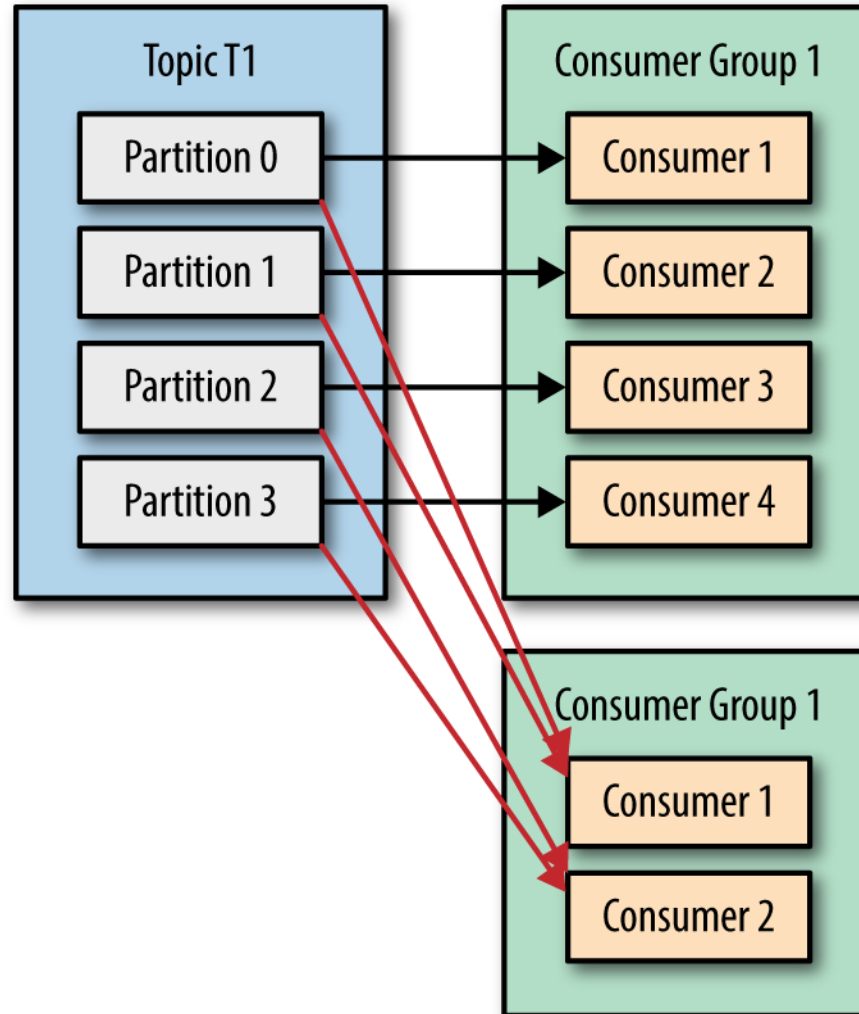


More consumer groups than partitions means missed messages



If we add more consumers to a single group with a single topic than we have partitions, some of the consumers will be idle and get no messages at all.

Adding a new consumer group ensures no messages are missed



Consumer Groups and Topic Subscriptions

Kafka uses the concept of consumer groups to allow a pool of processes to divide the work of consuming and processing records.

These processes can either be running on the same machine or they can be distributed over many machines to provide scalability and fault tolerance for processing.

All consumer instances sharing the same group.id will be part of the same consumer group.

It is also possible that the consumer could encounter a "livelock" situation where it is continuing to send heartbeats, but no progress is being made.

To prevent the consumer from holding onto its partitions indefinitely in this case, we provide a liveness detection mechanism using the `max.poll.interval.ms` setting.

The consumer provides two configuration settings to control the behavior of the poll loop:

max.poll.interval.ms:

By increasing the interval between expected polls, we can give the consumer more time to handle a batch of records returned from poll(long). The drawback is that increasing this value may delay a group rebalance since the consumer will only join the rebalance inside the call to poll. You can use this setting to bound the time to finish a rebalance, but you risk slower progress if the consumer cannot actually call poll often enough.

max.poll.records:

Use this setting to limit the total records returned from a single call to poll. This can make it easier to predict the maximum that must be handled within each poll interval. By tuning this value, you may be able to reduce the poll interval, which will reduce the impact of group rebalancing.

Manual Partition Assignment

In some cases we may need finer control over the specific partitions that are assigned. For example:

- If the process is maintaining some kind of local state associated with that partition (like a local on-disk key-value store), then it should only get records for the partition it is maintaining on disk.

- If the process itself is highly available and will be restarted if it fails (perhaps using a cluster management framework like YARN, Mesos, or AWS facilities, or as part of a stream processing framework).
- In this case there is no need for Kafka to detect the failure and reassign the partition since the consuming process will be restarted on another machine.

```
String topic = "foo";
```

```
TopicPartition partition0 = new TopicPartition(topic, 0);
```

```
TopicPartition partition1 = new TopicPartition(topic, 1);
```

```
consumer.assign(Arrays.asList(partition0, partition1));
```

Controlling The Consumer's Position

Kafka allows specifying the position using

`seek(TopicPartition, long)` to specify the new position.

Special methods for seeking to the earliest and latest offset the server maintains are also available (`seekToBeginning(Collection)` and `seekToEnd(Collection)` respectively).

Delivery semantics for consumers

consumers choose when to commit offsets

- At most once
- At least once (usually preferred)
- > Exactly once (transactional)

At most once:

offsets are committed as soon as the message is received
if the processing goes wrong, the message will be lost

At least once:

offsets are committed after the message is processed if the processing goes wrong, the message will be read again this can result in duplicate processing of messages. use idempotent consumers.

Exactly once:

can be achieved for Kafka => kafka workflows using kafka streams api

At-most-once Kafka Consumer (Zero or more deliveries)

At-most-once consumer is the default behavior of a KAFKA consumer.

To configure this type of consumer,

- ❑ Set 'enable.auto.commit' to true
- ❑ Set 'auto.commit.interval.ms' to a lower timeframe.
- ❑ And do not make call to `consumer.commitSync()`; from the consumer. With this configuration of consumer, Kafka would auto commit offset at the specified interval.

At-Least-Once Kafka Consumer (One or more message deliveries, duplicate possible)

To configure this type of consumer:

- ❑ Set 'enable.auto.commit' to false
- ❑ Set 'auto.commit.interval.ms' to a higher number.
- ❑ Consumer should then take control of the message offset commits to Kafka by making the following call `consumer.commitSync()`

Producer - Adding the custom headers

```
Headers headers = new RecordHeaders();  
headers.add(new RecordHeader("header-1", "header-  
value-1".getBytes()));  
headers.add(new RecordHeader("header-2", "header-  
value-2".getBytes()));
```

```
ProducerRecord<Long, String> record = new  
ProducerRecord<Long, String>(TOPIC_NAME, null,  
counter, "A sample message", headers);
```

Consumer – Custom Headers

```
ConsumerRecords<Long, String> records =  
kafkaConsumer.poll(Duration.ofMillis(3000));
```

```
System.out.println("Fetched " + records.count() + "  
records");  
for (ConsumerRecord<Long, String> record : records)  
{  
    System.out.println("Received: " + record.key() + ":" +  
record.value());  
    record.headers().forEach(header -> {  
        System.out.println("Header key: " + header.key() +  
", value:" + header.value());  
    });  
}
```

**LET'S
PRACTICE**

03_Kafka_Lab

Kafka __consumer_offsets topic

Since kafka 0.9, it's not zookeeper anymore that store the information about what were the offset consumed by each groupid on a topic by partition.

Kafka now store this information on a topic called
__consumer_offsets

Topic : __consumer_offsets (default values)

offsets.topic.replication.factor

Number of replicas for __consumer_offsets topic. The default value is 3.

offsets.topic.num.partitions

Number of partitions for __consumer_offsets topic. The default value is 50.

As it is binary data, to see what's inside this topic we will need to consume with the formatter
OffsetsMessageFormatter :

For kafka 0.11.0.0 and above

LET'S
PRACTICE

```
$KAFKA_HOME/bin/kafka-console-consumer.sh --formatter  
"kafka.coordinator.group.GroupMetadataManager$Offsets  
MessageFormatter" --bootstrap-server localhost:9092 --topic  
__consumer_offsets --from-beginning
```

The output will have the below format:

```
[groupId,topicName,partitionNumber]::[OffsetMetadata[Offset  
Number,NO_METADATA],CommitTime  
1520613132835,ExpirationTime 1520699532835]
```

```
[test-group,test-  
topic,0]::[OffsetMetadata[20,NO_METADATA],CommitTime  
1520613211176,ExpirationTime 1520699611176]
```

* for test-topic on test-group, consumer read to the offset 20,
on the partition 0.

But maybe I have 30 offsets to be read on my partition 0, and it should alert me, that the worker that should be reading the partition 0, is late or for some reason are not committing the offsets as read, so potentially an error should be fixed.

For that there are another command that can help us (you should know the name of the groupid you want to monitor):

LET'S
PRACTICE

```
$KAFKA_HOME/bin/kafka-run-class.sh  
kafka.admin.ConsumerGroupCommand --bootstrap-server kafka:9092  
--group my-group --describe
```

Topic	Partition	Current-offset	log-end-offset	Lag
test-topic	0	20	20	0
test-topic	1	-	1	-

if they are equal it's perfect, - means that we don't have yet the information on the `__consumer_offsets`.

Broker Configs

The essential configurations are the following:

broker.id

log.dirs

zookeeper.connect

zookeeper.connect

Specifies the ZooKeeper connection string in the form hostname:port

Specify multiple hosts in the form

hostname1:port1,hostname2:port2,hostname3:port3

advertised.listeners

Listeners to publish to ZooKeeper for clients to use,

auto.create.topics.enable (default: true)

Enable auto creation of topic on the server

auto.leader.rebalance.enable (default: true)

Enables auto leader balancing. A background thread checks and triggers leader balance if required at regular intervals

(OR)

kafka-preferred-replica-election.sh --zookeeper localhost:2181

compression.type (default: none)

The compression type for all data generated by the producer. Specify the final compression type for a given topic. This configuration accepts the standard compression codecs ('gzip', 'snappy', 'lz4', 'zstd')

delete.topic.enable (default: true)

leader.imbalance.check.interval.seconds (default : 300)
The frequency with which the partition rebalance check is triggered by the controller

listeners

Specify hostname as 0.0.0.0 to bind to all interfaces.

PLAINTEXT://myhost:9092,SSL://:9091

CLIENT://0.0.0.0:9092,REPLICATION://localhost:9093

log.flush.interval.messages (default : 9223372036854775807)

The number of messages accumulated on a log partition before messages are flushed to disk

Valid [1...]

log.flush.interval.ms

The maximum time in ms that a message in any topic is kept in memory before flushed to disk.

If not set, the value in log.flush.scheduler.interval.ms is used

log.flush.scheduler.interval.ms (default : 9223372036854775807)

The frequency in ms that the log flusher checks whether any log needs to be flushed to disk

log.flush.offset.checkpoint.interval.ms (default : 60000)

The frequency with which we update the persistent record of the last flush which acts as the log recovery point

log.retention.bytes

The maximum size of the log before deleting it

log.retention.hours (default : 168)

The number of hours to keep a log file before deleting it (in hours), tertiary to log.retention.ms property

log.retention.minutes

log.retention.ms

log.segment.bytes (default : 1073741824)

The maximum size of a single log file

log.segment.delete.delay.ms (default : 60000)

The amount of time to wait before deleting a file from the filesystem

message.max.bytes (default: 1000012)

The largest record batch size allowed by Kafka.

min.insync.replicas (default : 1)

When a producer sets acks to "all" (or "-1"), min.insync.replicas specifies the minimum number of replicas that must acknowledge

num.io.threads (default: 8)

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

num.network.threads (default :3)

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

num.recovery.threads.per.data.dir (default: 1)

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

zookeeper.connection.timeout.ms

The max time that the client waits to establish a connection to zookeeper. If not set, the value in zookeeper.session.timeout.ms is used

zookeeper.session.timeout.ms (6000)

Zookeeper session timeout

zookeeper.max.in.flight.requests (default : 10)

The maximum number of unacknowledged requests the client will send to Zookeeper before blocking.