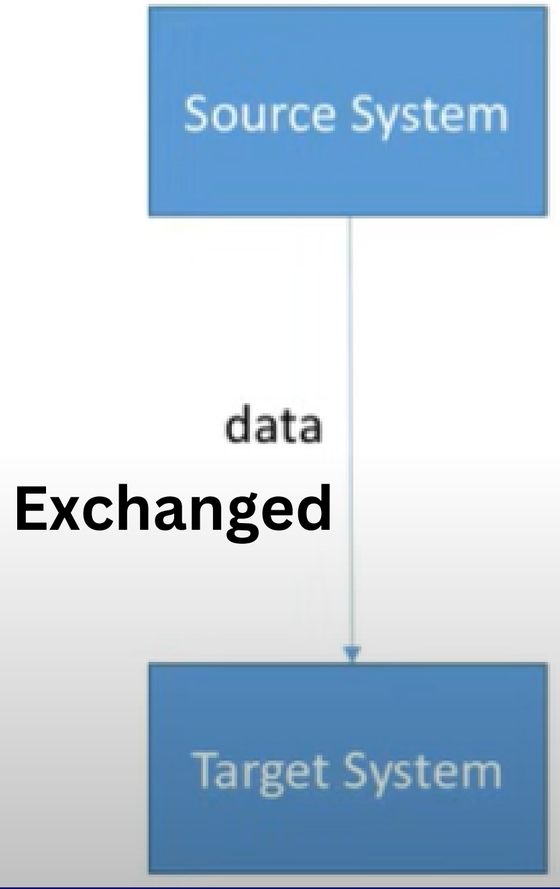
UNIT- IV

EVENT PROCESSING WITH APACHE KAFKA

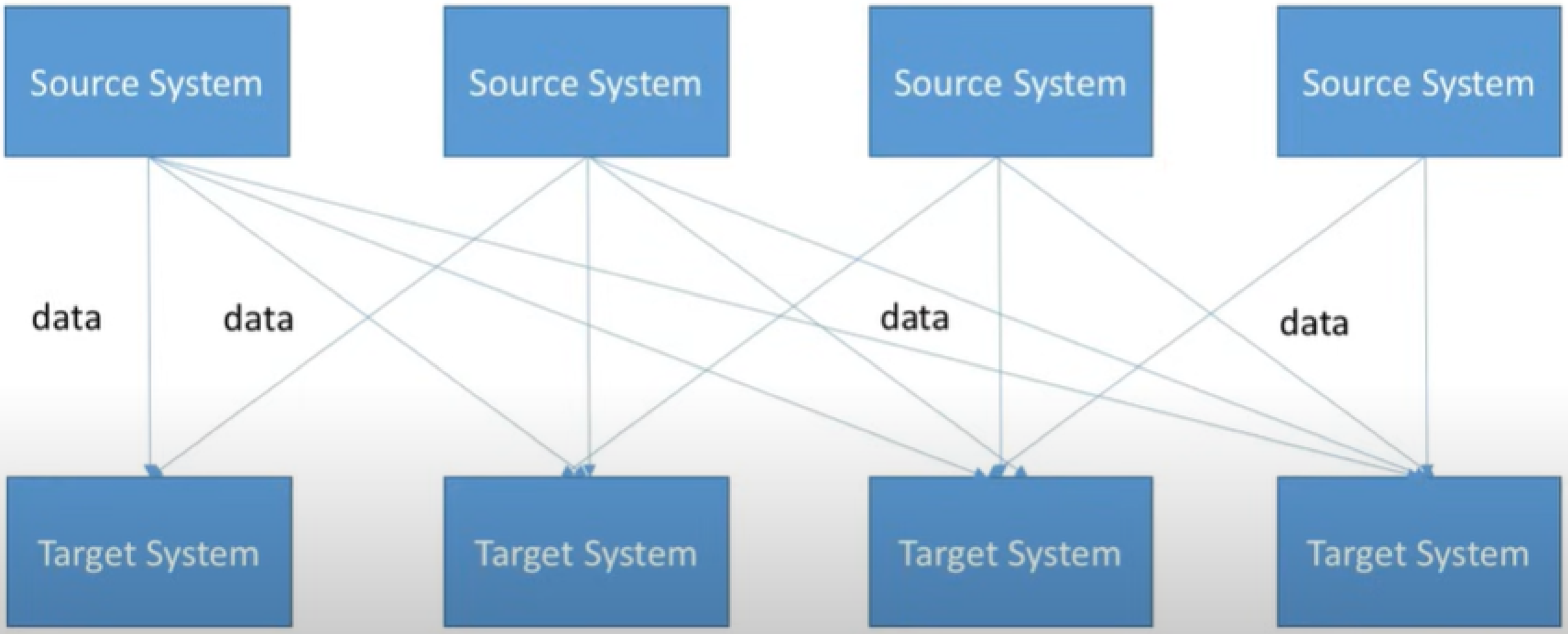
UNIT – IV EVENT PROCESSING WITH APACHE KAFKA 6

Apache Kafka, Kafka as Event Streaming platform, Events, Producers, Consumers, Topics, Partitions, Brokers, Kafka APIs, Admin API, Producer API, Consumer API, Kafka Streams API, Kafka Connect API

How companies start



Simple at first!



Very complicated!

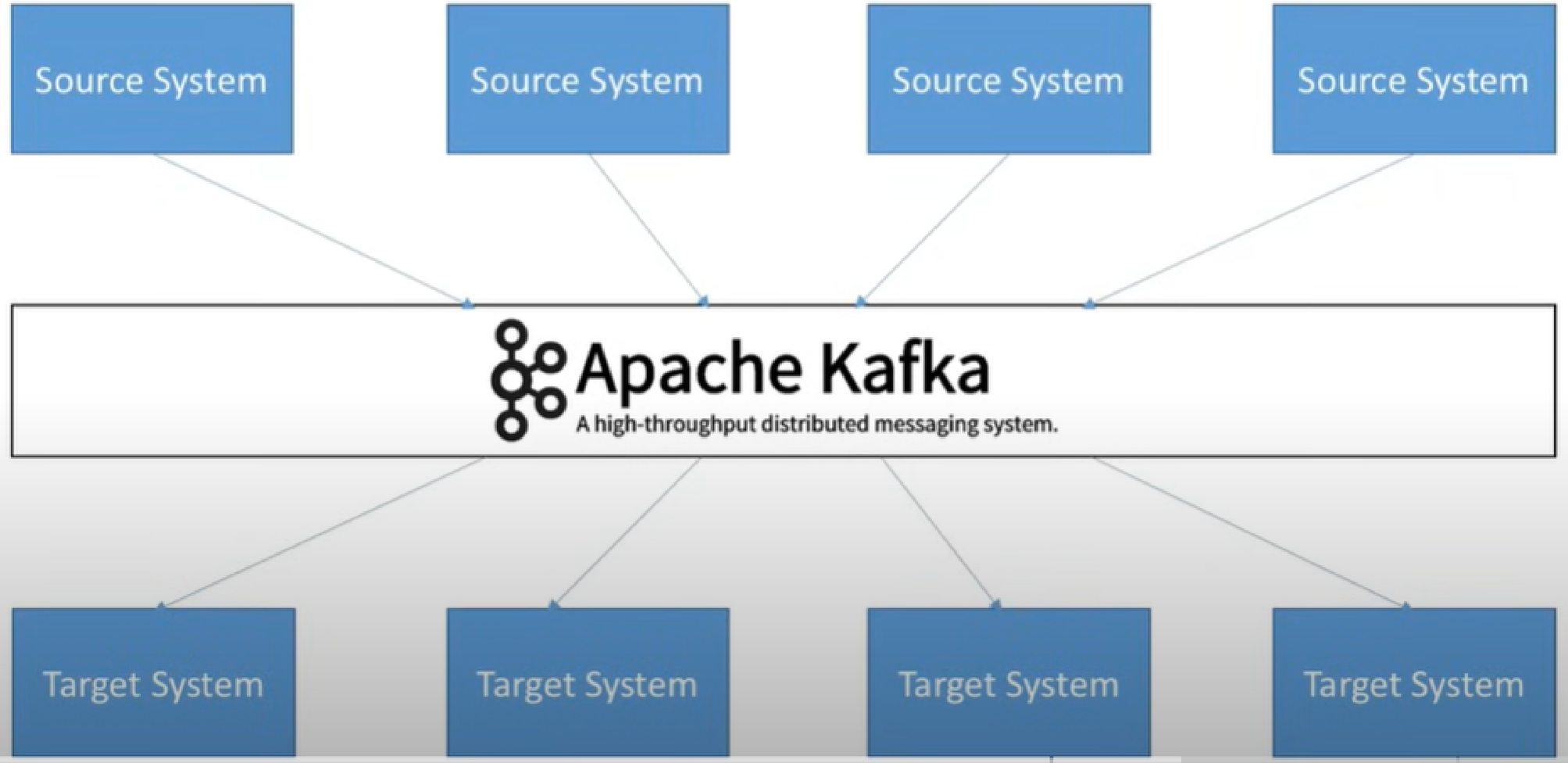
Problems organisations are facing with the previous architecture

- If you have 4 source systems, and 6 target systems, you need to write 24 integrations!

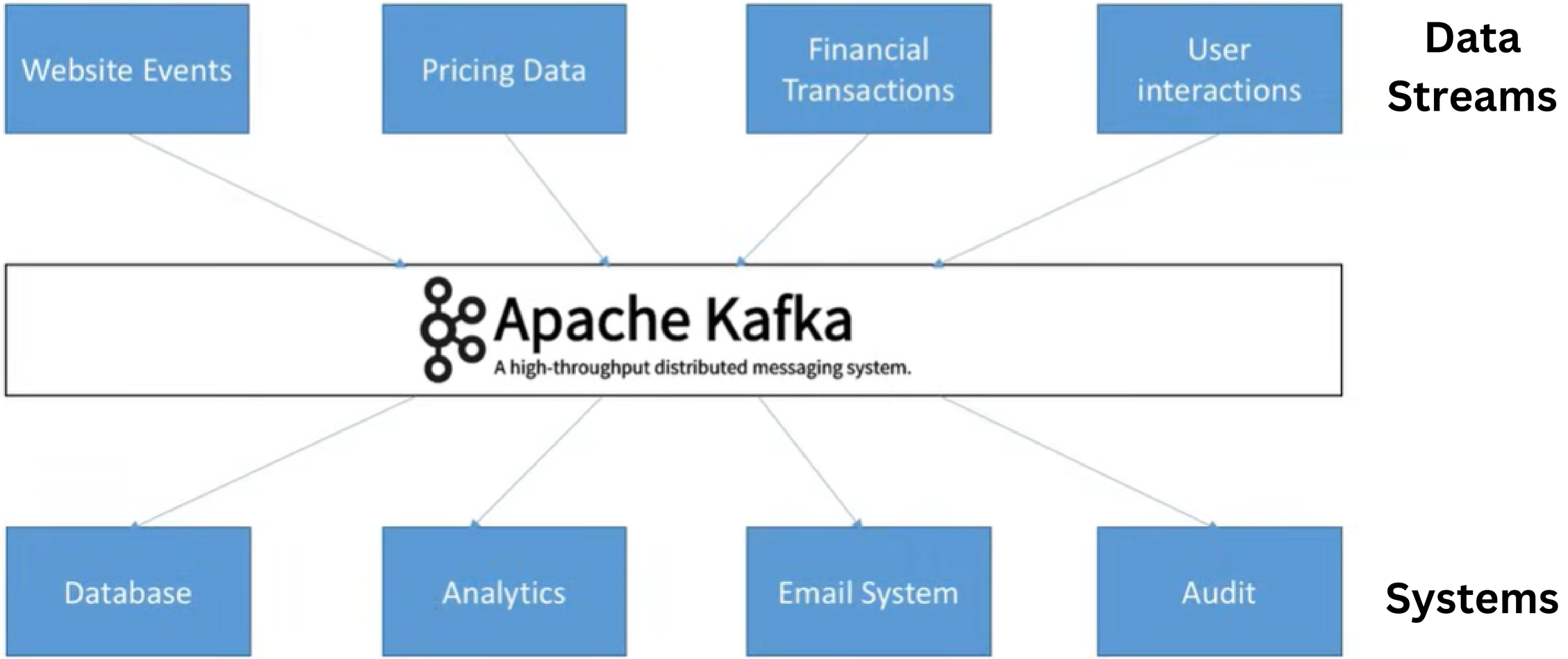
- Each integration comes with difficulties around
  - Protocol – how the data is transported *(TCP, HTTP, REST, FTP, JDBC…)*
  - Data format – how the data is parsed *(Binary, CSV, JSON, Avro…)*
  - Data schema & evolution – how the data is shaped and may change

- Each source system will have an increased load from the connections

Why Apache Kafka:
Decoupling of data streams & systems



Why Apache Kafka:
Decoupling of data streams & systems



Apache Kafka

- Apache Kafka is a **distributed streaming platform** for building real-time data pipelines and streaming applications.
- Apache Kafka originally created by **LinkedIn**.
- Kafka was created to Ingest **high volumes of event data with low latency**.

Apache Kafka

- Apache Kafka was open-sourced in **2011** through the **Apache Software Foundation** and mainly maintained by **Confluent**.
- It has since become one of the most popular event streaming platforms.
- Kafka is an event processing system.
- Kafka - Message Distributing System
- It is **highly scalable,** it can scale up to **millions of messages per second.**
- It has **high performance**, the latency to exchange data from one system to another is less than 10 millisecond.

Linked in

NETFLIX

Apache Kafka: Use cases

- Messaging System
- Activity Tracking
- Gather metrics from many different locations
- Application Logs gathering
- Stream processing (with the Kafka Streams API or Spark for example)
- De-coupling of system dependencies
- Integration with Spark, Flink, Storm, Hadoop, and many other Big Data technologies

Kafka as Event Streaming platform

Kafka as Event Streaming platform

- Event - An event is **something that happens** at a **specific point in time** signaling that **something has happened** as well as the information about **exactly what happened**.
- An event could be: when the user of a website adds a product to a cart or pays an invoice.
- The geographical coordinates of a pedestrian could be an event.
- Events will not be an interesting one if they just happen once or infrequently.
- Events will be a **repeating and evolving nature**.

Kafka as Event Streaming platform

- Events form into **event streams**, which reflect the changing behavior of a system.
- The true power of event streaming lies in the combination of many streams together which we call **event stream processing**.
- Here one stream reads from the output of another, making calculations that feed into third.

What is an Event Streaming?

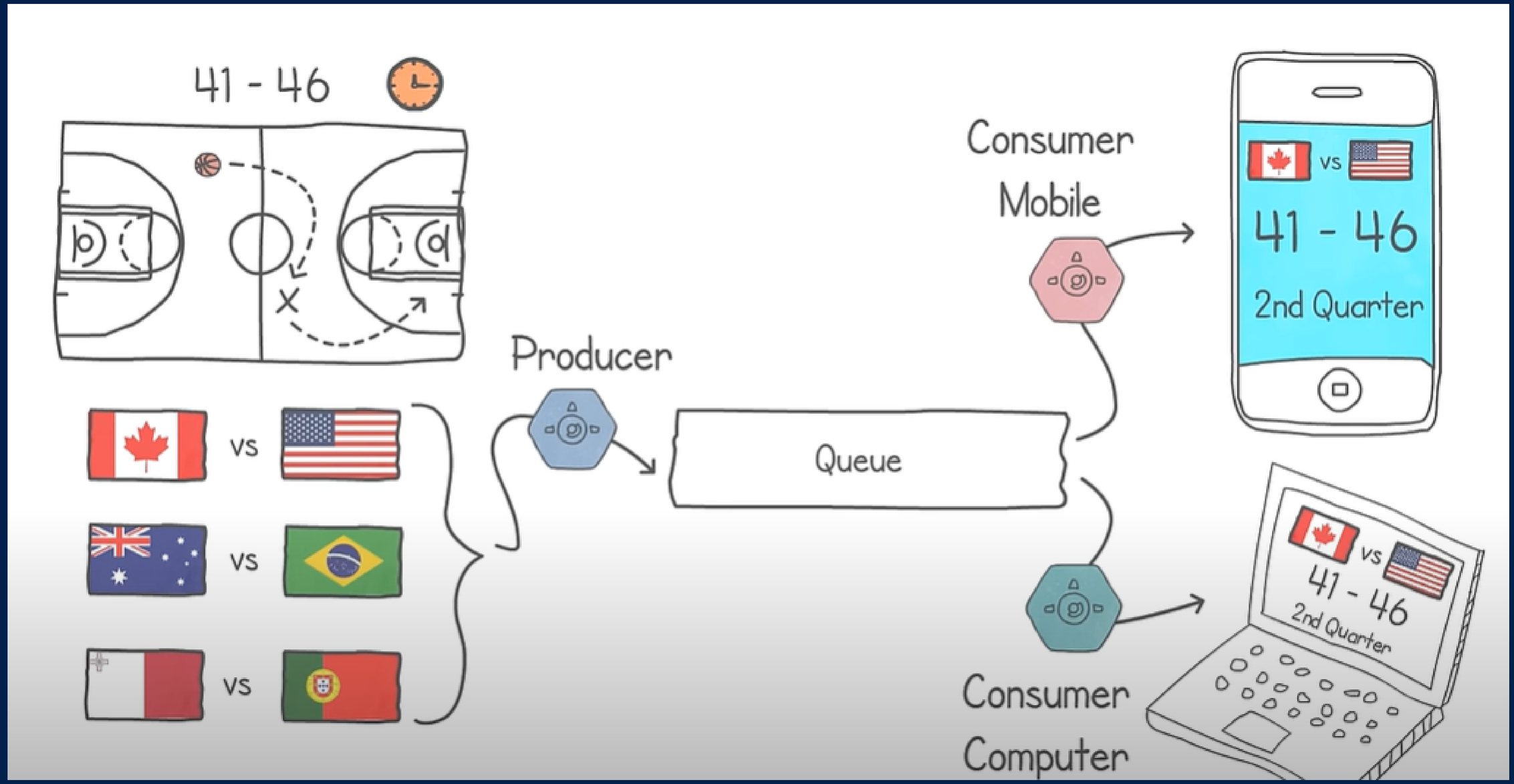
- An Event Stream is an **ordered sequence of events** representing important actions in a software domain.
- This can be something simple, like **clicking on a link**, or it might be something more complex, like **transferring funds between two banks**.
- **Event Stream Processing (ESP)** takes a continuous stream of events and processes them as soon as a change happens.
- By processing single points of data rather than an entire batch, event streaming platforms provide an architecture that enables software to understand, react to, and operate as events occur.

How Event Streaming Works?

- In software, any significant action can be recorded as an event.
- For example, someone clicking a link or viewing a webpage, or something more involved like paying for an order, withdrawing money, or even communicating with numerous, distributed IoT devices at once.
- These events can be organized into streams, essentially a series of events ordered by time.
- From there, events can be shared with other systems where they can be processed in real-time.
- Events are pushed and handled one at a time, as they happen.

How Event Streaming Works?

- This allows the system to react in real-time, rather than waiting for batches to accumulate.
- For example, each time someone clicks a link or views a webpage, we might push an event into a system such as **Apache Kafka**.
- Downstream, a **Flink** job could consume those events to **develop analytics** about how many views and clicks our website is receiving.

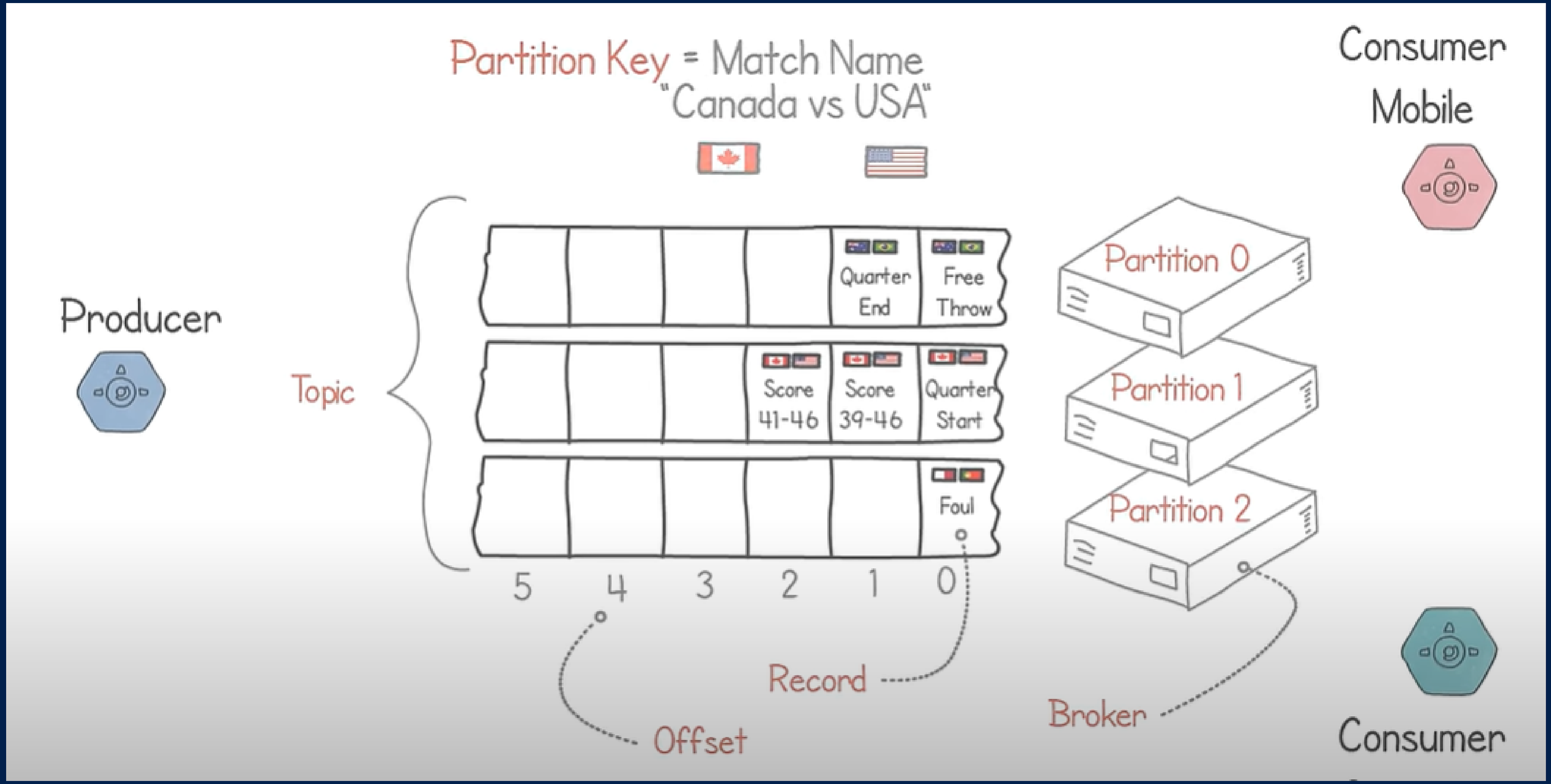


- Producer reads the update and write them in a queue.
- From the queue, many consumers are consuming the infomration.

- Over a period of time, accumulation of data leads to the issue that is Queue is running out of memory.
- Hence, extra queues are added.
- The information gets distributed across the queues using the match name.
- Updates coming from the same match will be on the same queue.
- This is the difference between Kafka and other messaging systems.
- Items sent and receivevd through kafka requires a distribution strategy

Terminologies of Apache Kafka

- In Kafka, each one of the queues is called a partition.
- The total of partition is called partition count.
- Each server holding one or more partitions is called a broker.
- Each item in a partition is called a record.
- Grouping of partitions handling the same type of data is called topic.
- To identify each record uniquely, kafka provides a sequential number to each record called an offset.
- A record in a topic is identified by a partition number and an offset.


Partition Key = Match Name
"Canada vs USA"
Consumer
Mobile
Producer
Topic
Quarter End
Free Throw
Score 41-46
Score 39-46
Quarter Start
Foul
5
4
3
2
1
0
Partition 0
Partition 1
Partition 2
Record
Offset
Broker
Consumer

What are events?

Events

- Events are “things that happen,” or sometimes, they are otherwise defined as representations of facts.
- An event in kafka refers to a piece of data that represents a specific action in a system. It is also called messages or records.
- Transaction in a bank - Event
- An event is represented by something that looks like this:

{key: "nerve_signal" value: "beta: 12Hz, gamma: 8Hz" timestamp: "1979 8:52:32 AM GMT-07:00"}

- These events usually have header, partition keys and values and a timestamp, and the whole thing looks suspiciously like an object.

Characteristics :

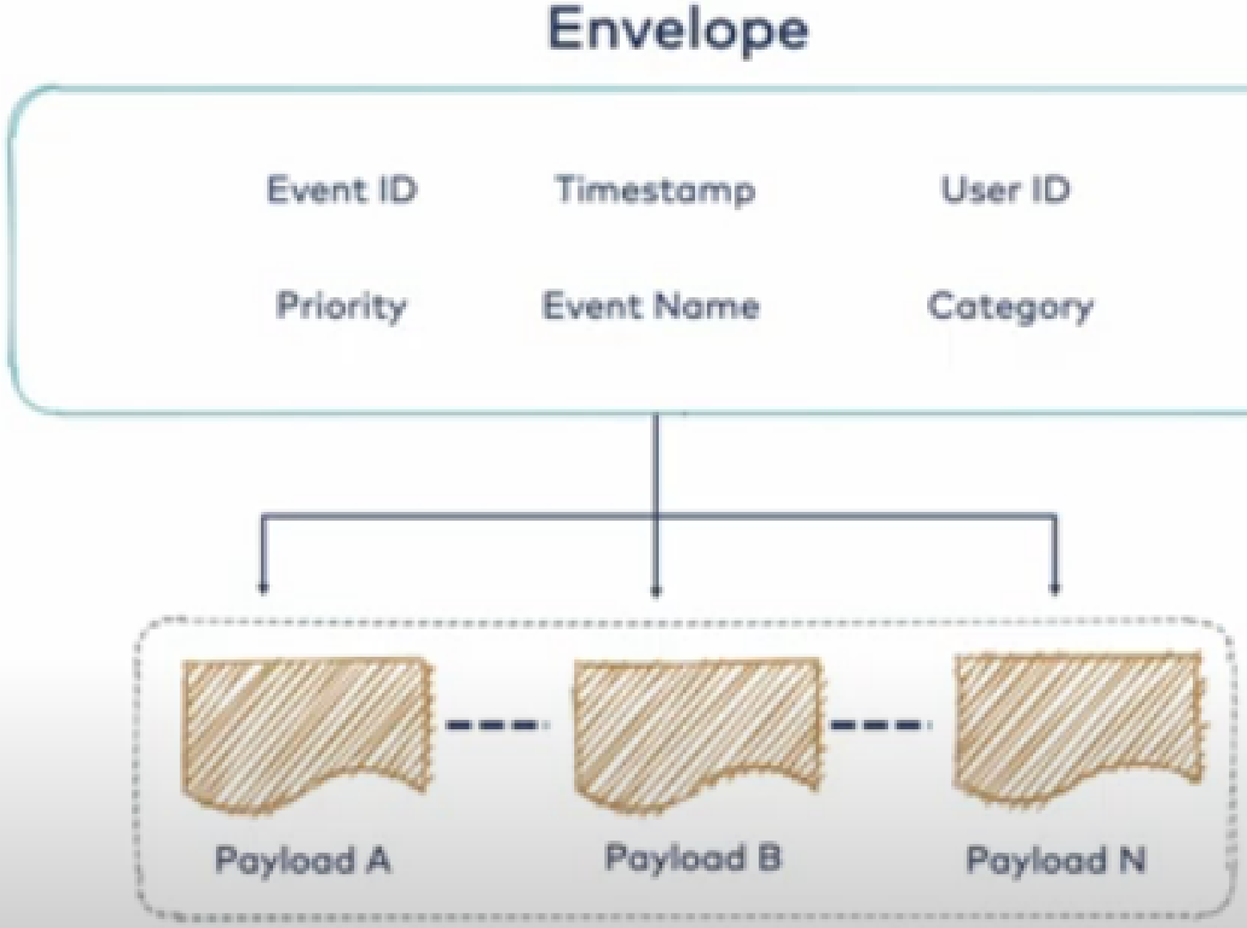
1. Events are IMMUTABLE & can include any type of data, such as a log entry, sensor reading, or user interaction.

2. SMALL in size and are produced and consumed asynchronously.

3. Ordered: Events within a SINGLE PARTITION are strictly ORDERED based on their offset.

4. Retained: Kafka events are RETAINED for a configurable period of time.

Event Payload vs Event Metadata



Metadata	"offset": 23, "partition": 0, "topic": "Shopping.Cart.v2" "timestamp": 1660501516
Key	3125123
Value	"cart_id": 3125123, "item_map": [(521,1), (923,3)], "shipping": "express"

Payload - How much amount transferred?

To whom the amount was transferred?

Metadata - At what time, this transaction happened?

Describing the payload

Event ID

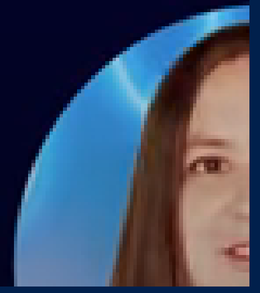
Events in the Kafka Ecosystem

event-driven design

event sourcing

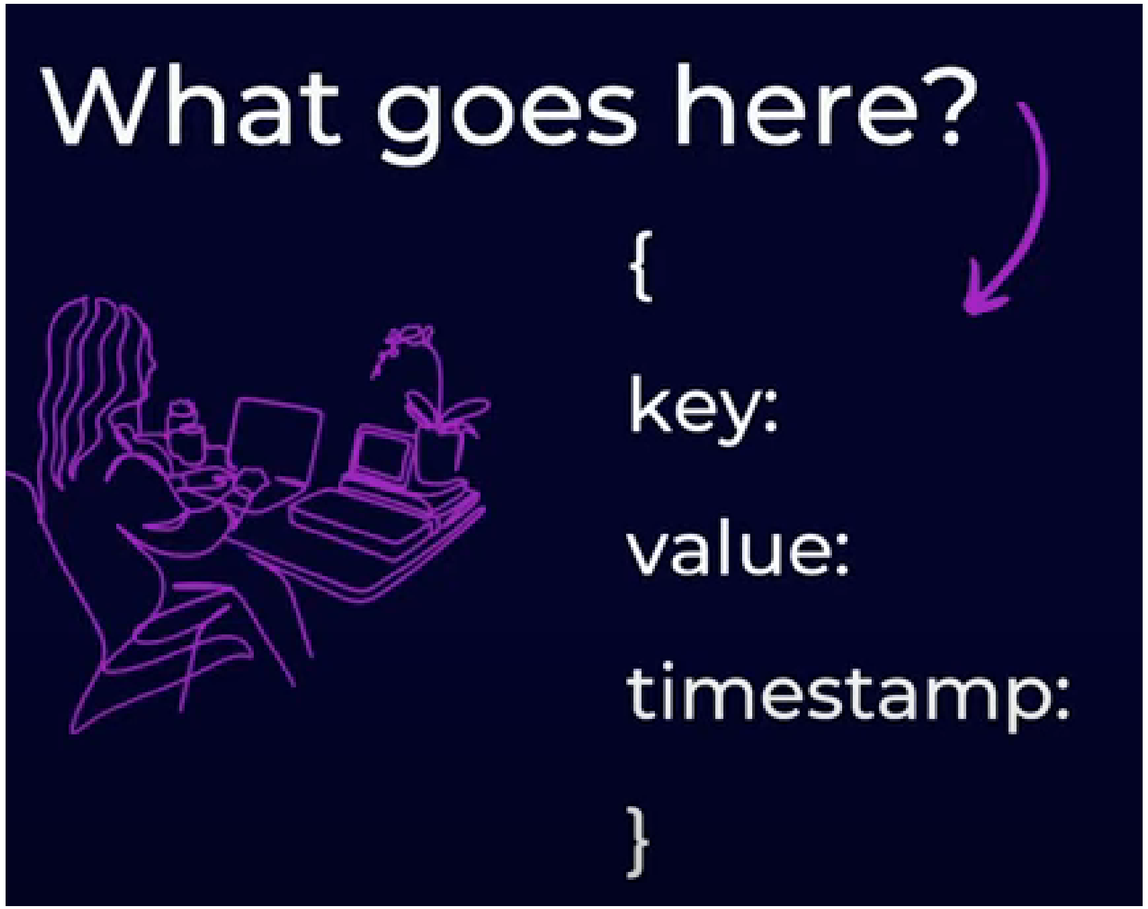
designing events

event streaming



Event Design

- **Event design is about answering the question, what values go here? and that is affected by things like whether you want your events to be read by external versus internal systems, or how tightly coupled you want the relationship between parts of your structure to be.**



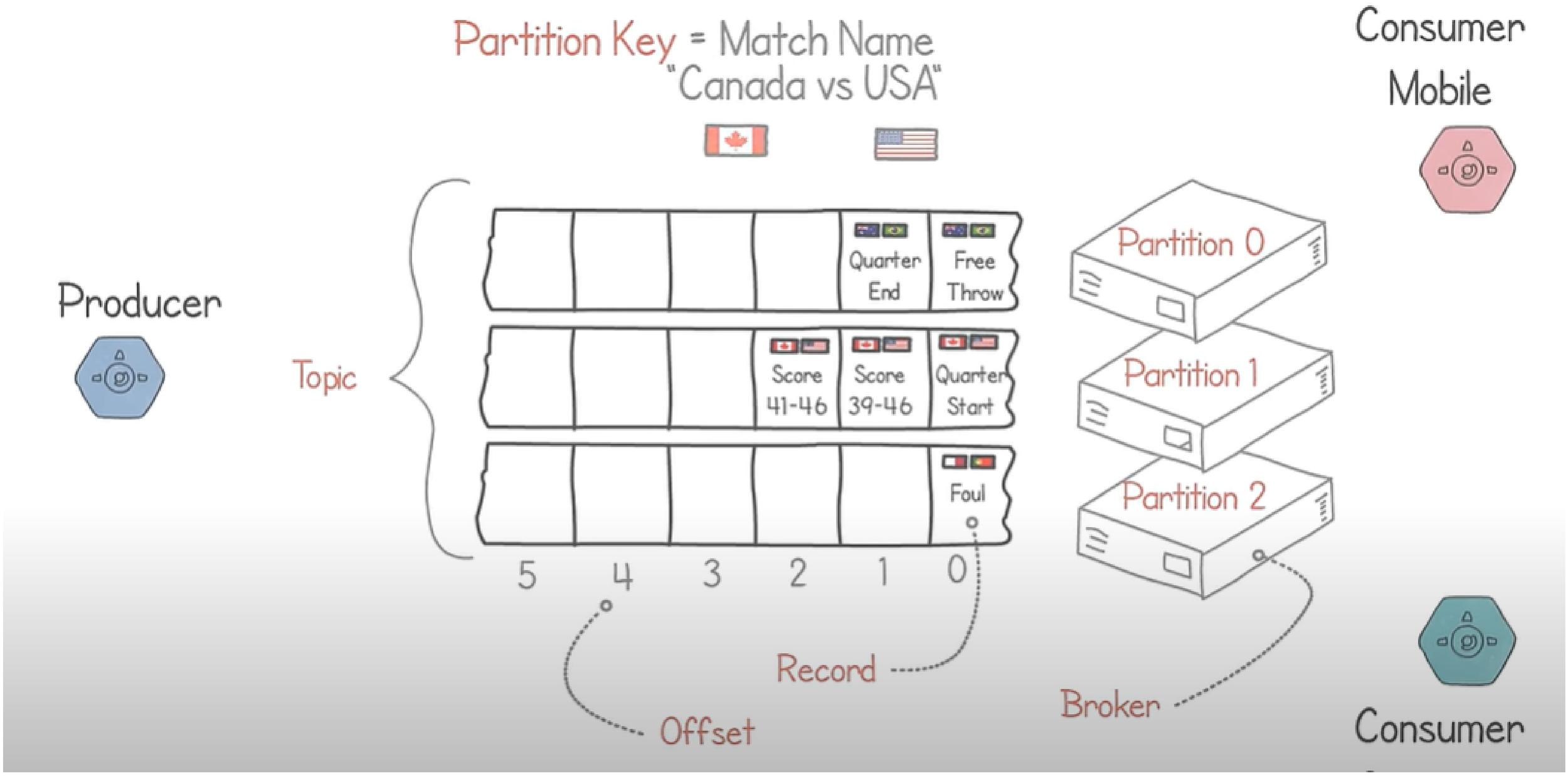
Event Streaming

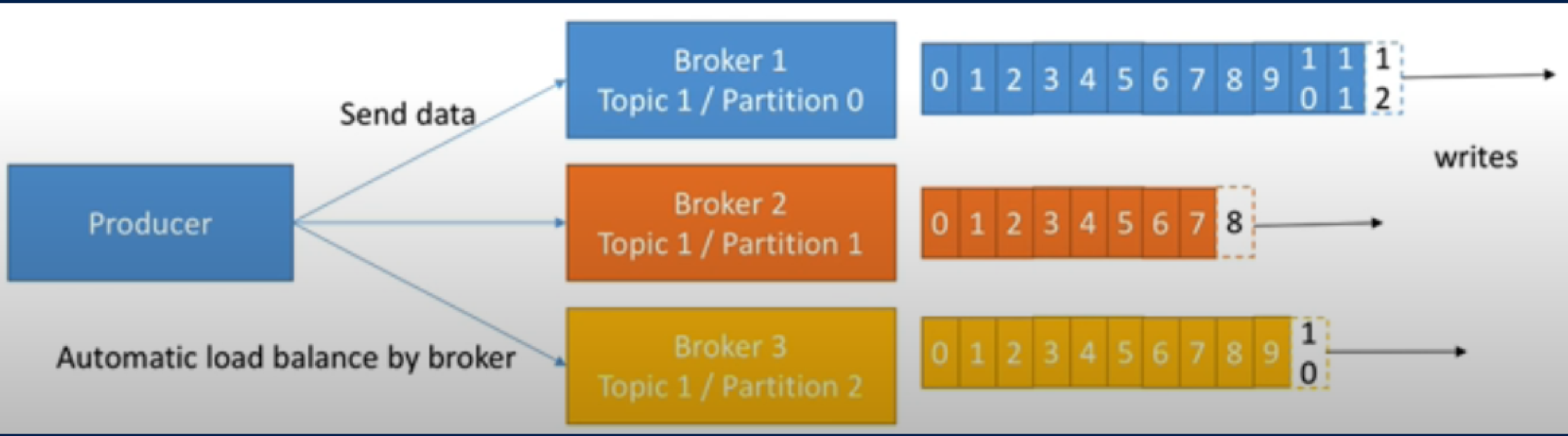
- **Event Streaming – A repeating phenomenon, we want to measure like changes in temperature for an Internet of Things App.**
- **We can record that is database or we can record a real time stream and doing operations like querying, Joining, Filtering and Aggregation.**

Producer

Producer

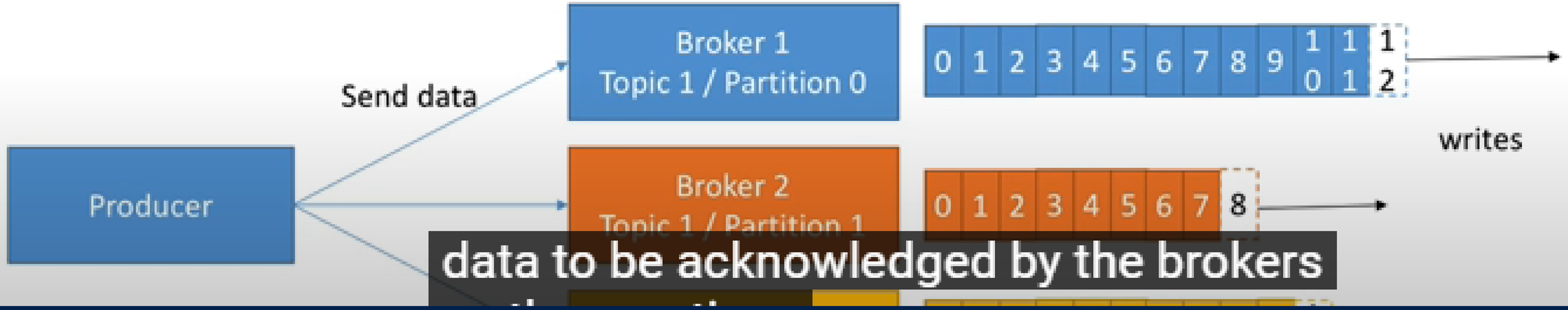
- In Apache Kafka, a producer is a component responsible for publishing records to Kafka topics.
- A Kafka topic is a category name to which records are published.
- Producers push records (also known as messages or events) into Kafka topics, and these records are then made available to consumers.
- They only have to specify the topic name and one broker to connect to, and Kafka will automatically take care of routing the data to the right brokers.


Partition Key = Match Name
"Canada vs USA"
Consumer
Mobile
Producer
Topic
Quarter End
Free Throw
Score 41-46
Score 39-46
Quarter Start
Foul
5
4
3
2
1
0
Partition 0
Partition 1
Partition 2
Record
Offset
Broker
Consumer


Broker 1
Topic 1 / Partition 0
Send data
Producer
Broker 2
Topic 1 / Partition 1
Broker 3
Topic 1 / Partition 2
Automatic load balance by broker
0 1 2 3 4 5 6 7 8 9 10 11 12
writes
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9 10

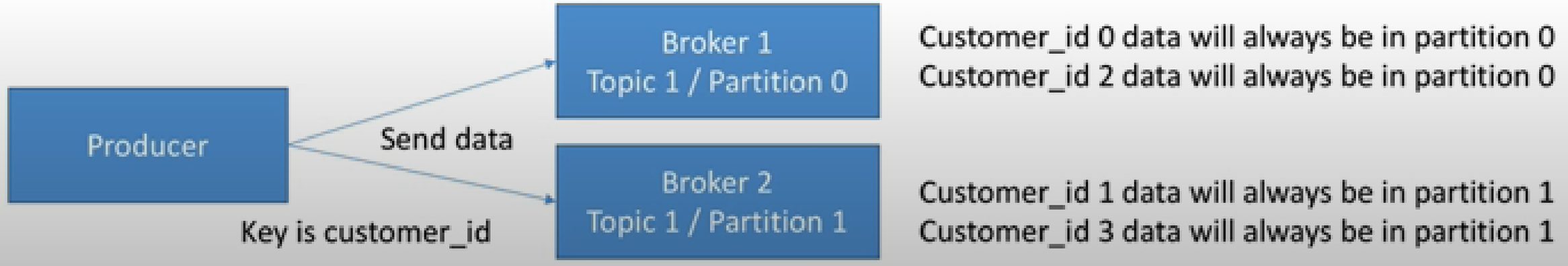
Producers

- Producers can choose to receive acknowledgment of data writes:
  - Acks=0: Producer won't wait for acknowledgment (possible data loss)
  - 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 the message
- If a key is sent, then the producer has the guarantee that all messages for that key will always go to the same partition
- This enables to guarantee ordering for a specific key



Key Aspects of Producer

1. Record:

- A producer sends records to Kafka topics.
- A record consists of a key, a value, and a timestamp.
- The key and value are both byte arrays, allowing for flexibility in the type of data that can be transmitted.

Key Aspects of Producer

2. Partitioning:

- Kafka topics are divided into partitions, and each partition can be thought of as a linearly ordered sequence of records.
- Producers decide to which partition a record should be sent, based on configurable partitioning strategies.
- The default strategy is round-robin, but custom partitioners can also be implemented to achieve specific behavior.

Key Aspects of Producer

3. Acknowledgment:

- **Producers can choose to receive acknowledgments from the Kafka brokers upon successful receipt of records.**
- **This acknowledgment mechanism ensures that producers can confirm whether their records were successfully delivered to Kafka.**

Key Aspects of Producer

4. Fault Tolerance:

- Producers are designed to handle failures gracefully.
- They can automatically recover from certain types of errors, such as leader changes or broker failures, by retrying failed operations.

Key Aspects of Producer

5. Configuration:

- Producers can be configured with various settings, including buffer size, batch size, compression type, and acknowledgment mode.
- These configurations allow developers to tune the performance and reliability of the producer to suit their specific requirements.

Key Aspects of Producer

6. Serialization:

- Before sending records to Kafka, producers typically serialize the key and value objects into byte arrays.
- Kafka provides support for different serialization formats, such as Avro, or JSON, through serializers and deserializers.

Consumer

CONSUMER

- In Apache Kafka, a consumer is a component responsible for subscribing to Kafka topics and retrieving records (messages or events) from those topics.
- Consumers form the other end of the messaging system compared to producers, as they are the entities that retrieve data from Kafka topics for processing.

Key Aspects of Consumer

1. Subscription:

- Consumers subscribe to one or more Kafka topics to receive records published to those topics.
- They can specify the topics they want to subscribe to either explicitly or using pattern-based subscriptions (e.g., subscribing to all topics matching a certain pattern).

Key Aspects of Consumer

2. Partition Assignment:

- Kafka topics are divided into partitions, and each partition is assigned to a specific consumer within a consumer group.
- Kafka ensures that each partition is consumed by only one consumer within the same consumer group, but multiple partitions can be assigned to different consumers within the group.

Key Aspects of Consumer

3. Offset Management:

- Kafka maintains an offset for each consumer within a partition, indicating the position of the last record that the consumer has processed.
- Consumers can manage these offsets manually or rely on Kafka's built-in offset management mechanisms, which track the offsets in a durable and fault-tolerant manner.

Key Aspects of Consumer

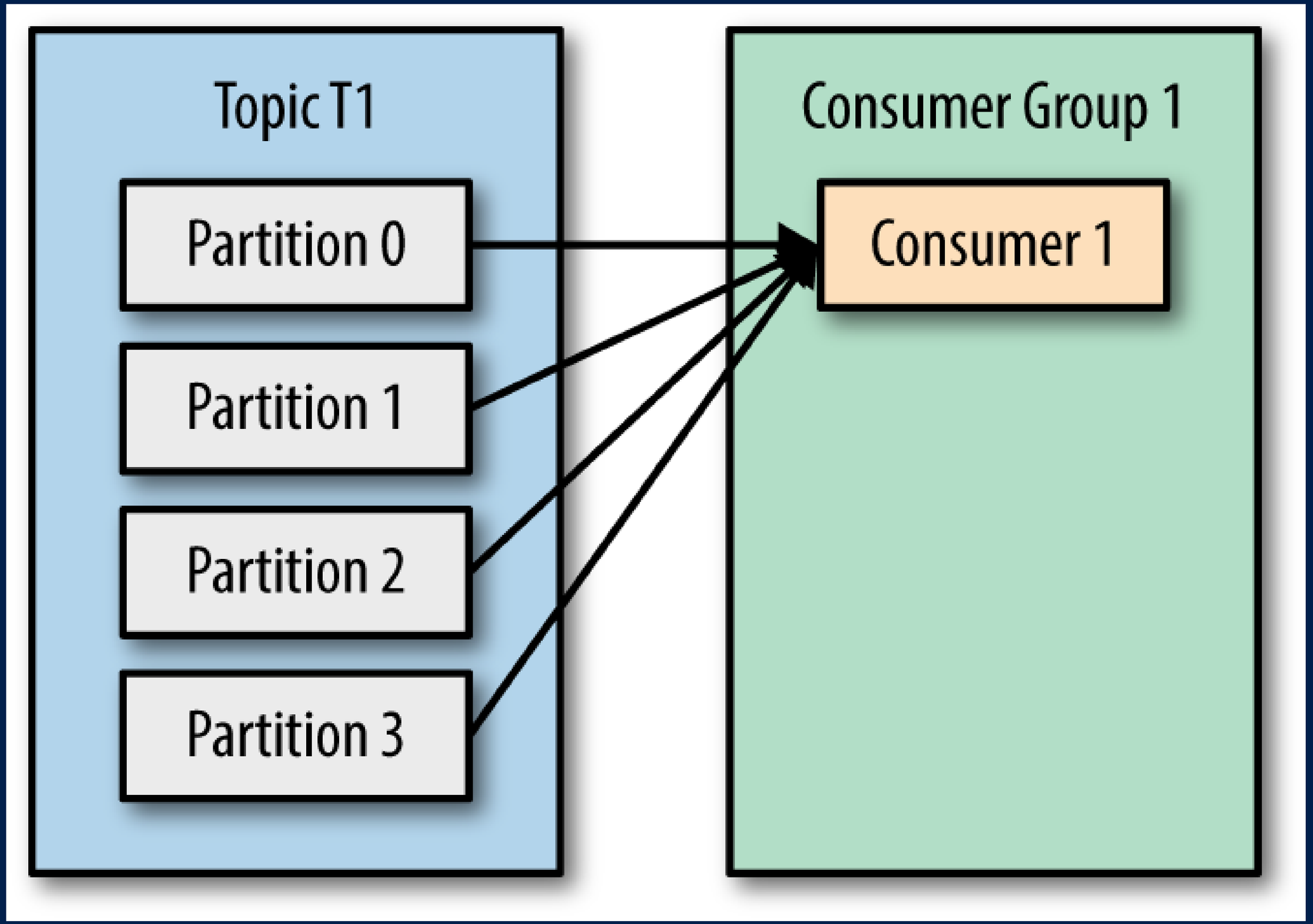
4. Polling Model:

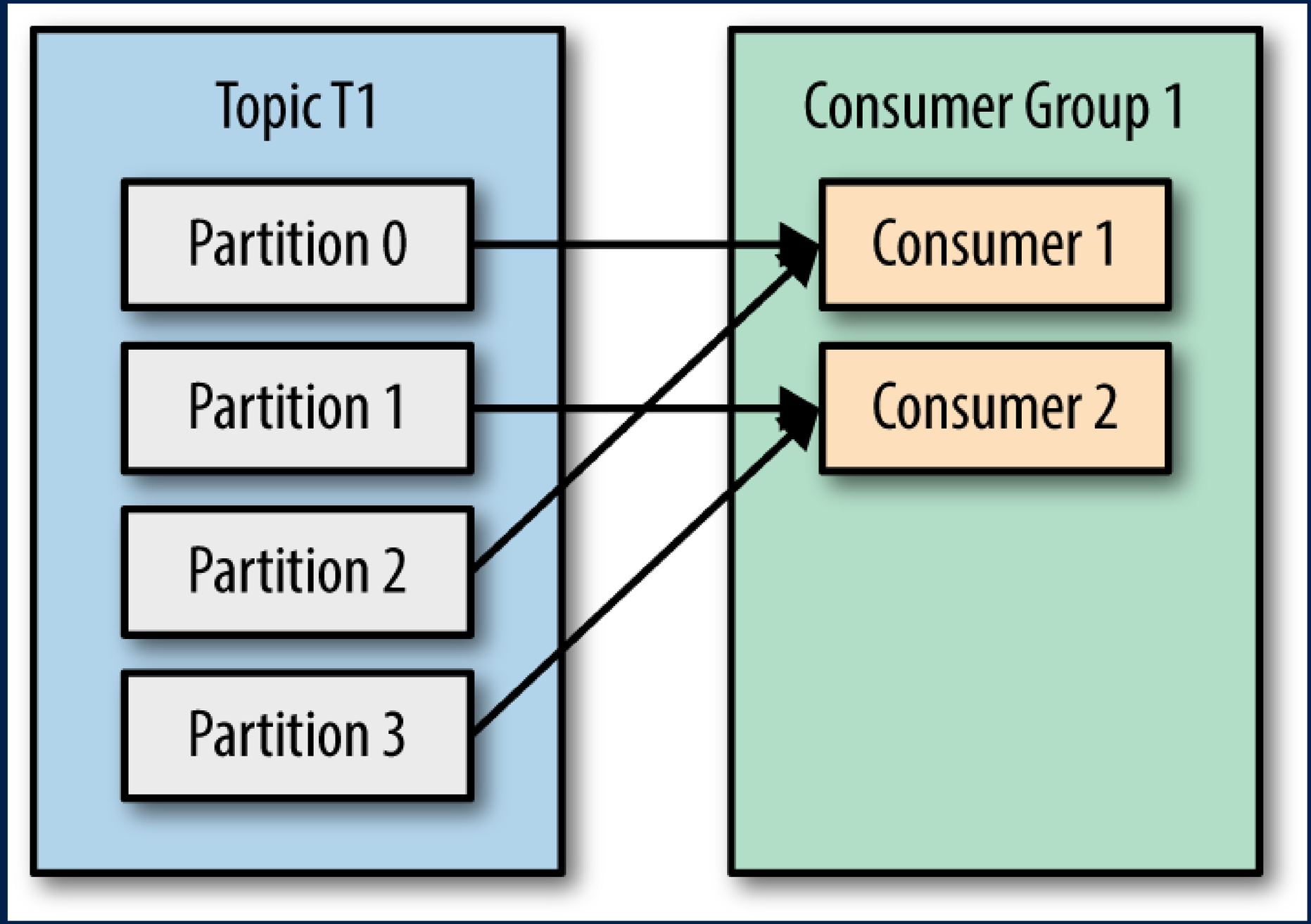
- Consumers typically use a polling model to retrieve records from Kafka.
- They periodically poll Kafka brokers for new records, specifying the maximum number of records to fetch in each poll request.
- This allows consumers to control their rate of consumption and handle records at their own pace.

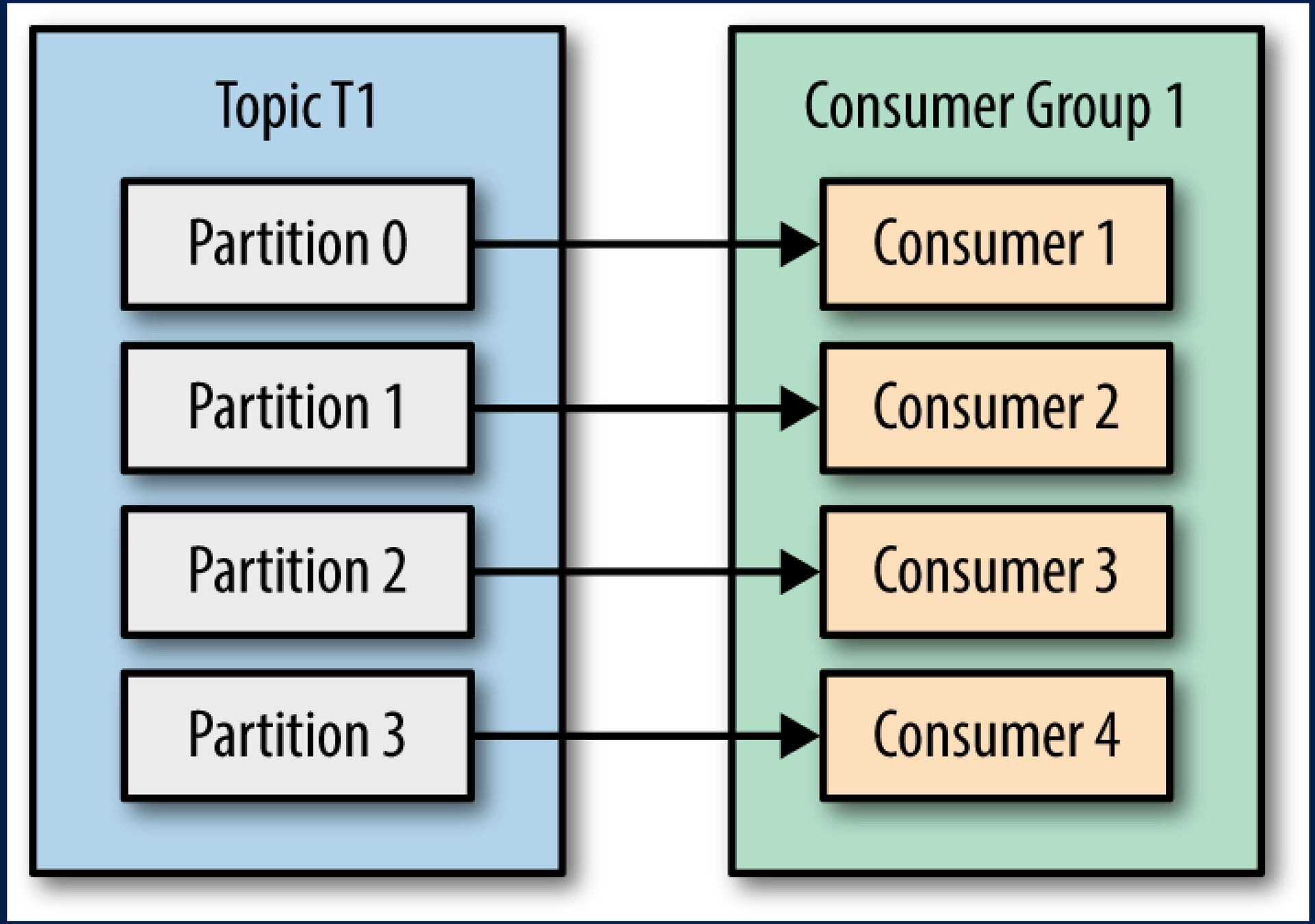
Key Aspects of Consumer

5. Parallelism:

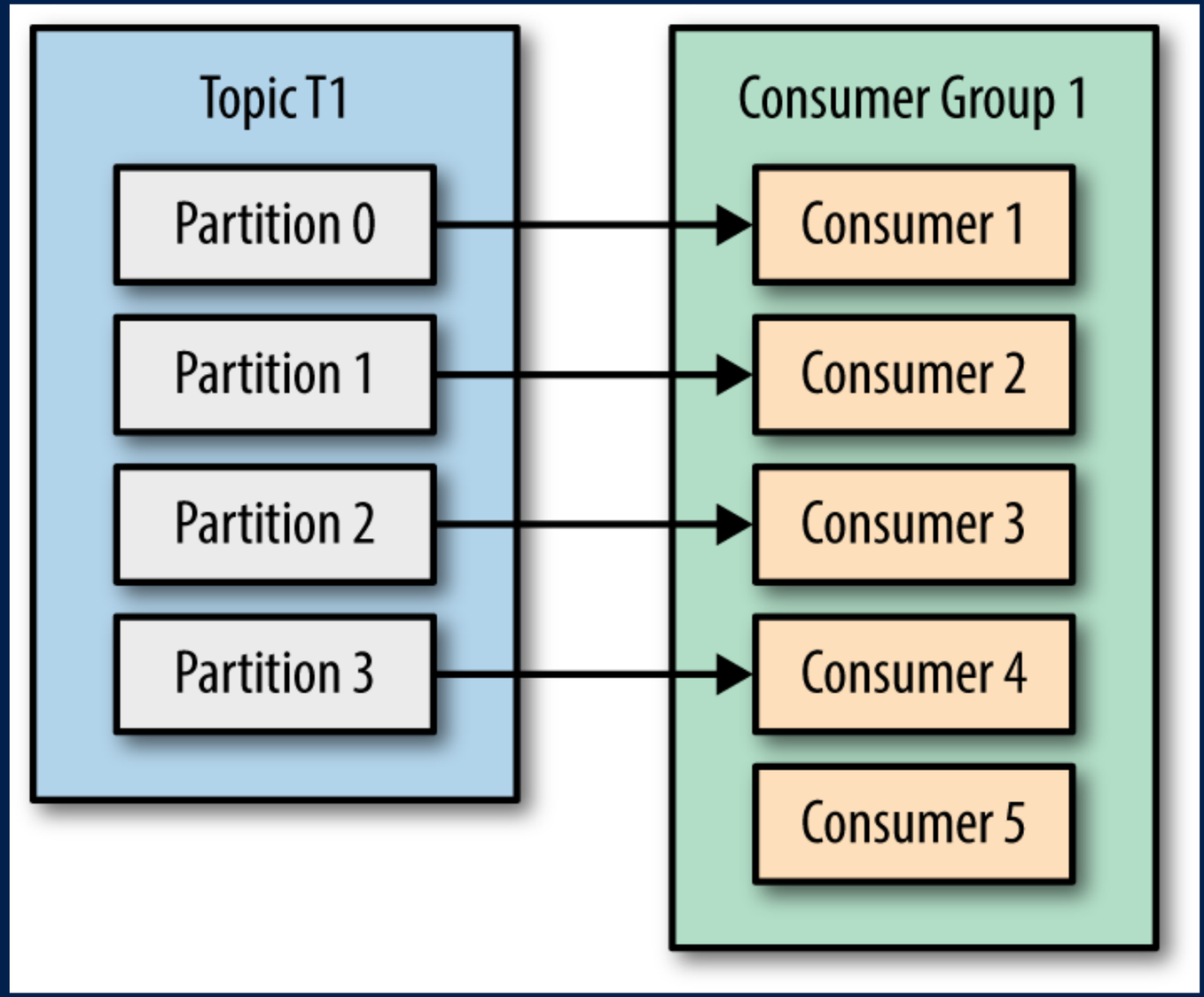
- Consumers within the same consumer group can operate in parallel, with each consumer processing records from one or more partitions simultaneously.
- This parallelism enables scalable and efficient processing of data across multiple consumers.


Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1

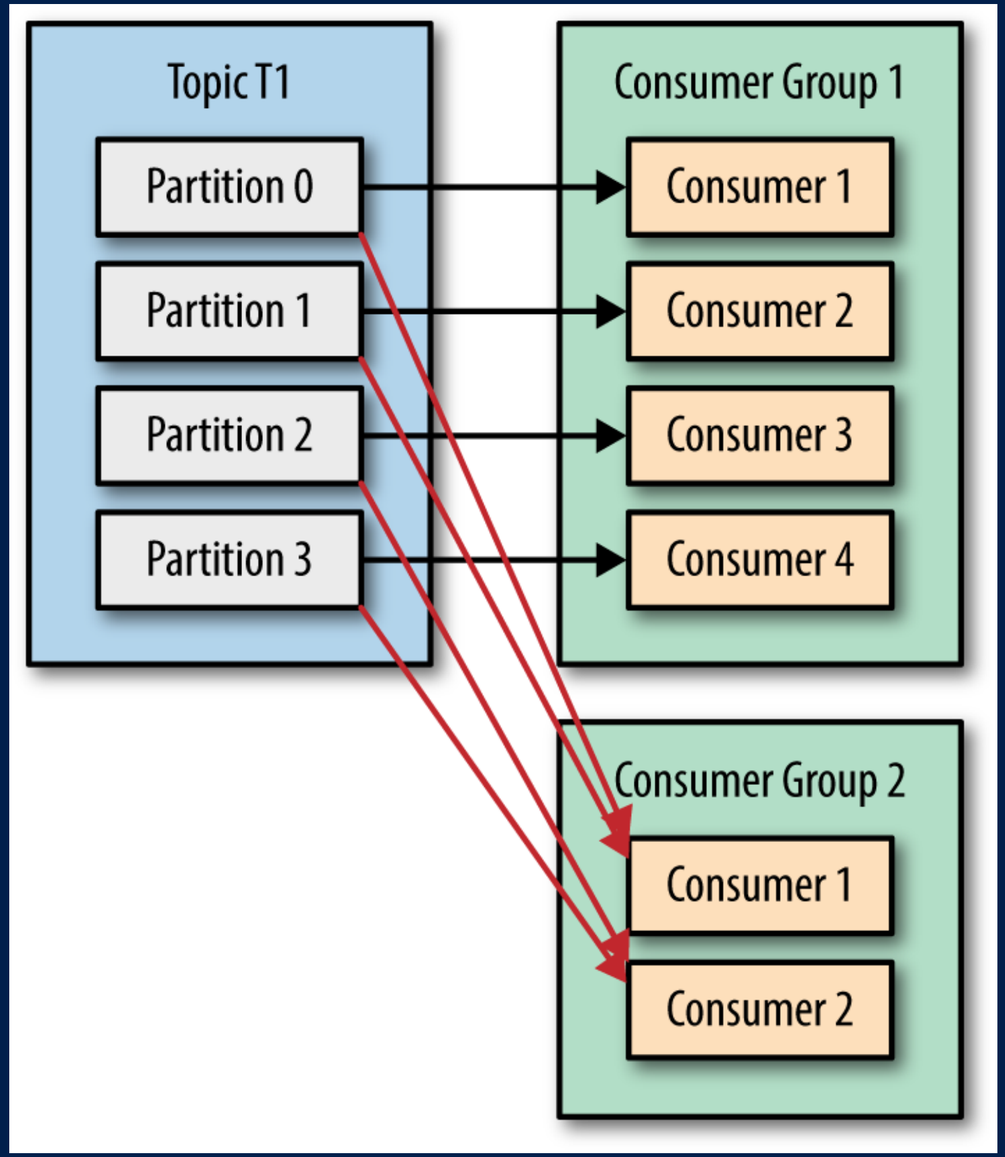

Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1
Consumer 2


Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1
Consumer 2
Consumer 3
Consumer 4

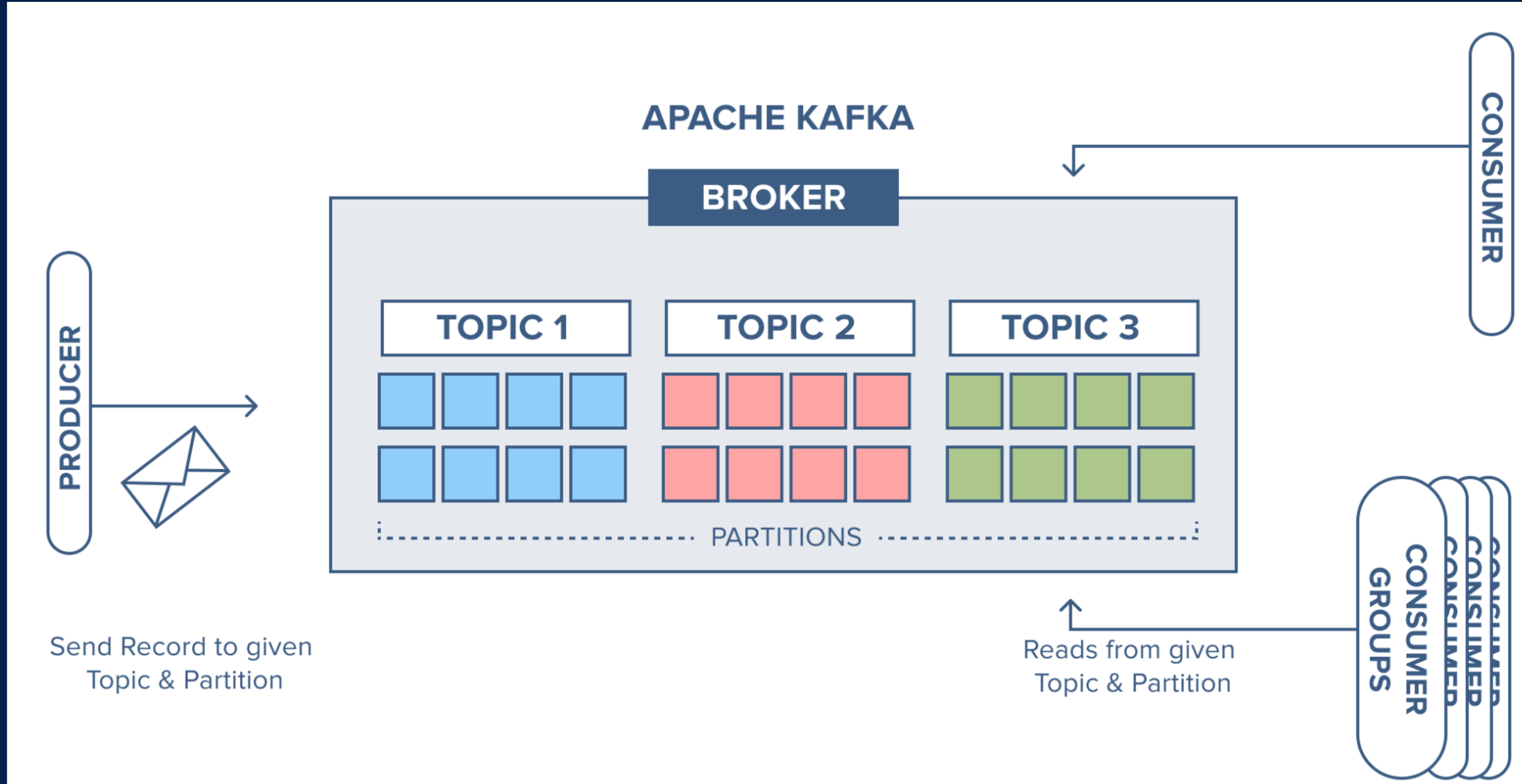
More consumers in a group than partitions means idle consumers



Adding a new consumer group, both groups receive all messages



Topics


APACHE KAFKA
BROKER
TOPIC 1
TOPIC 2
TOPIC 3
PARTITIONS
PRODUCER
CONSUMER
CONSUMER GROUPS
Send Record to given Topic & Partition
Reads from given Topic & Partition

Topics

- Topics are the fundamental units of data organization in Apache Kafka.
- They represent a stream of records, where each record is a key-value pair.
- Topics are similar to tables in databases or queues in messaging systems.
- Kafka topics are multi-subscriber. This means that a topic can have zero, one, or multiple consumers subscribing to that topic and the data written to it.

Anatomy of a topic

1. Name: Each topic is identified by a **unique name** within the Kafka cluster.

2. Partitions: Topics are divided into partitions, which are ordered, immutable sequences of records.

- Each partition is essentially a **commit log**.
- Partitions allow for **parallel processing and scalability.**

Anatomy of a topic

3. Replication: Kafka provides **fault tolerance** through **partition replication**.

- Each partition can have **multiple replicas** spread across different brokers to ensure **data durability and high availability**.

4. Retention Policy: Topics can have a retention policy that determines how long Kafka retains messages in the topic.

- Messages can be retained based on **time or size constraints.**

Importance of a topic

1. Data Organization: Topics provide a logical organization for data streams, allowing producers to publish records on specific topics and consumers to subscribe to topics of interest.

2. Decoupling: Topics decouple producers and consumers, enabling asynchronous communication between different components of distributed systems. Producers and consumers interact with topics independently, which promotes loose coupling and flexibility in system design.

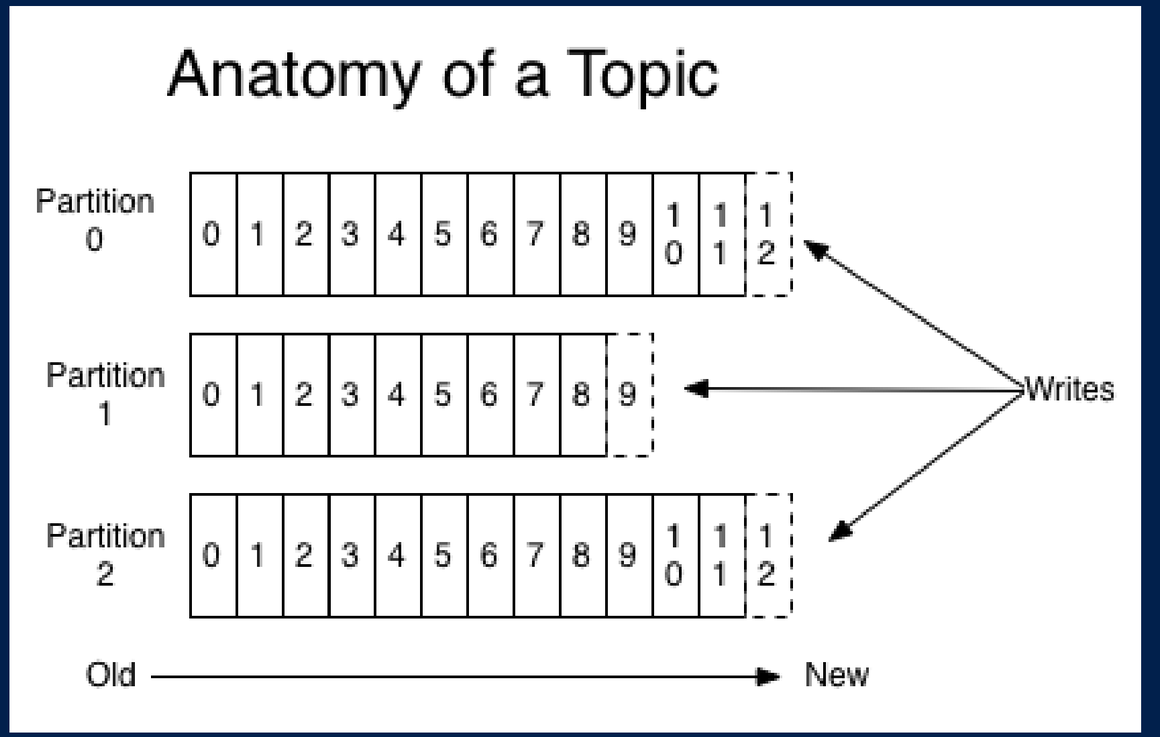
3. Event Sourcing: Kafka topics are often used as event logs or event streams, enabling event sourcing architectures where the entire history of changes to a system is captured as a sequence of events in Kafka topics.

Keyed Topics vs. Unkeyed Topics

- Topics can be either keyed or unkeyed.
- In keyed topics, records with the same key are always assigned to the same partition, ensuring order preservation for records with the same key.
- Unkeyed topics allow Kafka to distribute records across partitions randomly.

Partition Assignment

- In Kafka, topics are partitioned and replicated across brokers throughout the implementation.
- Brokers refer to each of the nodes in a Kafka cluster.
- The partitions are important because they enable parallelization of topics, enabling high message throughput.



Partitions

Partitions

- Partitions are a fundamental concept in Apache Kafka and play a crucial role in the scalability, reliability, and parallelism of data processing within Kafka.
- In Kafka, a topic is divided into one or more partitions.
- Each partition is an ordered, immutable sequence of records that are continually appended to by producers and read by consumers.
- Partitions are the basic unit of parallelism in Kafka and allow for horizontal scaling of both data storage and consumption.

Characteristics of Partitions

1. Ordering: Records within a partition are ordered by their offset, which represents the position of the record in the partition.
2. Immutable: Once written, records in a partition cannot be modified.
   - This immutability ensures that the data remains consistent and enables Kafka to efficiently support high-throughput data streams.

Characteristics of Partitions

3. Scalability: By partitioning a topic, Kafka can distribute the data across multiple brokers in a cluster, enabling horizontal scalability.

- Each partition can reside on a different broker, allowing Kafka to handle large volumes of data.

4. Replication: Partitions can be replicated across multiple brokers for fault tolerance and high availability.

- Kafka maintains multiple replicas of each partition to ensure that data remains accessible even in the event of broker failures.

Characteristics of Partitions

5. Consumer Parallelism: Consumers within a consumer group can parallelize the consumption of records by assigning different partitions to different consumer instances.

- This allows Kafka to achieve high throughput and low latency for data processing.

Partitioning Strategy

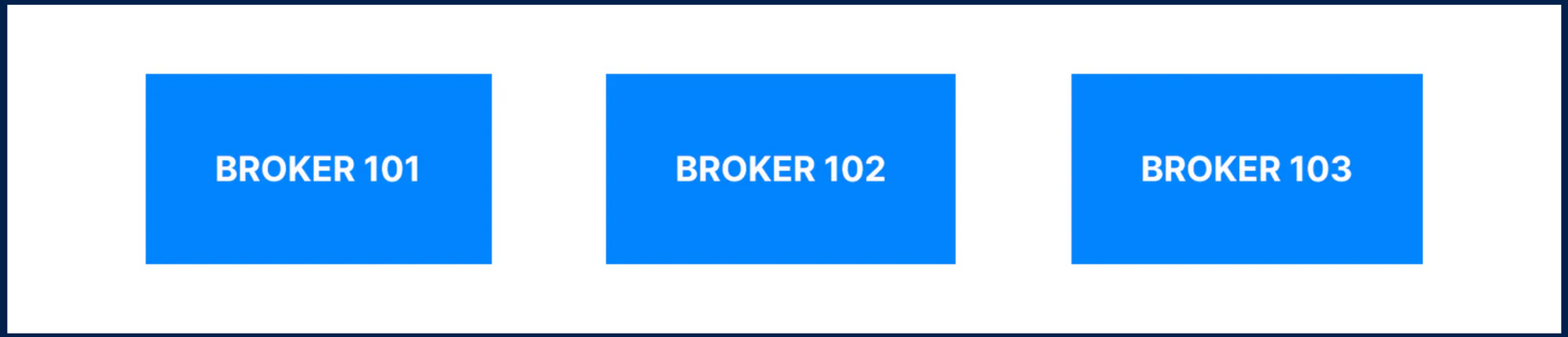
- When a producer publishes a record to a Kafka topic without specifying a partition, Kafka uses a partitioning strategy to determine which partition the record should be written to.
- The default partitioning strategy is based on the record's key.
- Kafka calculates a hash of the key and uses it to select the partition.
- This ensures that records with the same key are always written to the same partition, preserving the order of records with the same key.

Brokers

Broker

- A single Kafka server is called a Kafka Broker.
- A Kafka broker is a program that runs on the Java Virtual Machine (Java version 11+) and usually a server/a computer/instance/or container that is meant to run the Kafka process.
- An ensemble of Kafka brokers working together is called a Kafka cluster.
- Some clusters may contain just one broker or others may contain three or potentially hundreds of brokers.
- Companies like Netflix and Uber run hundreds or thousands of Kafka brokers to handle their data.

- A broker in a cluster is identified by a unique numeric ID.



Kafka Cluster

Role of Brokers

1. Data Storage: Brokers store the data published by producers to Kafka topics. Each broker hosts one or more partitions of one or more topics. The data is stored on disk in an append-only manner, ensuring high throughput and durability.

2. Data Replication: Brokers replicate data across multiple brokers to ensure fault tolerance and high availability. Kafka maintains multiple replicas of each partition, distributing them across different brokers in the cluster. This replication ensures that data remains accessible even if one or more brokers fail.

Role of Brokers

3. Data Ingestion: Brokers receive data from producers and append it to the appropriate partitions of the topics. They handle incoming requests efficiently, buffering and batching data to optimize throughput and reduce disk I/O.

4. Data Serving: Brokers serve data to consumers by responding to fetch requests. Consumers can read data from any replica of a partition, allowing Kafka to distribute read requests across multiple brokers and achieve high throughput.

5. Metadata Management: Brokers maintain metadata about topics, partitions, replicas, and consumer groups. This metadata is used by clients to discover the structure of the Kafka cluster and to route requests to the appropriate brokers.

Characteristics of Brokers

1. **Scalability:** Kafka clusters can scale horizontally by adding more brokers to distribute the data and client requests across a larger number of nodes. Each broker contributes to the overall throughput and storage capacity of the cluster.
2. **Fault Tolerance:** Brokers replicate data across multiple nodes to ensure fault tolerance. If a broker fails, Kafka can continue to serve data from replicas hosted on other brokers, ensuring uninterrupted data availability.

Characteristics of Brokers

3. High Throughput: Brokers are optimized for high-throughput data ingestion and serving. They leverage techniques such as batching, compression, and zero-copy networking to achieve low latency and high throughput.

4. Resource Management: Brokers manage system resources such as CPU, memory, and disk I/O to ensure optimal performance and stability. They use configurable settings for buffer sizes, disk quotas, and resource limits to prevent resource contention and overload.

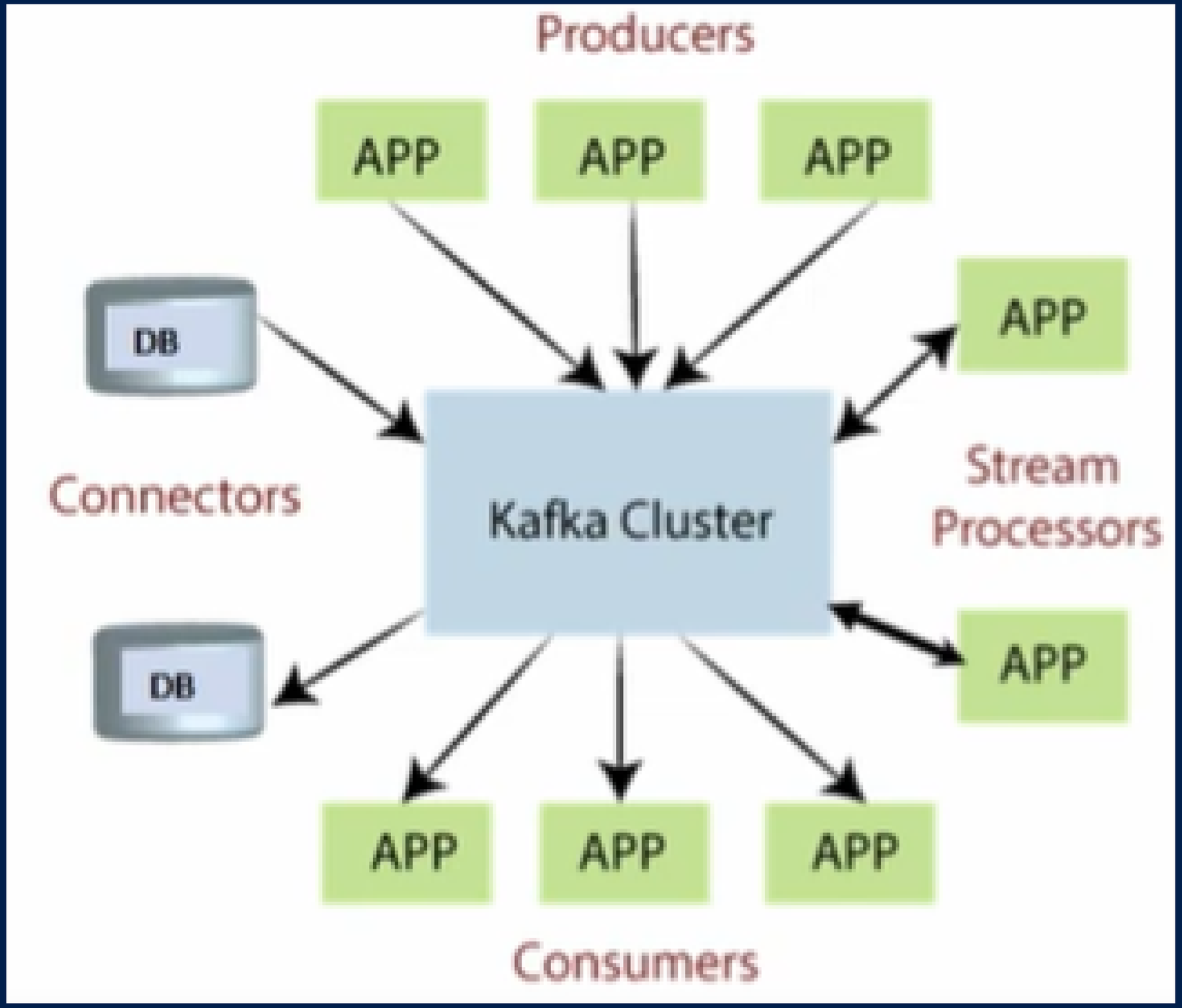
KAFKA API

KAFKA API

- "Kafka API" typically refers to the various interfaces (Application Programming Interfaces) provided by Apache Kafka, a distributed streaming platform.
- These APIs allow developers to interact with Kafka clusters programmatically, enabling them to perform various operations such as producing, consuming, and managing streams of data.

KAFKA API

1. Producer API
2. Consumer API
3. Streams API
4. Connector API
5. Admin API


Producers
APP
APP
APP
DB
APP
Connectors
Kafka Cluster
Stream
Processors
DB
APP
APP
APP
APP
Consumers

Producer API sends messages to Kafka.

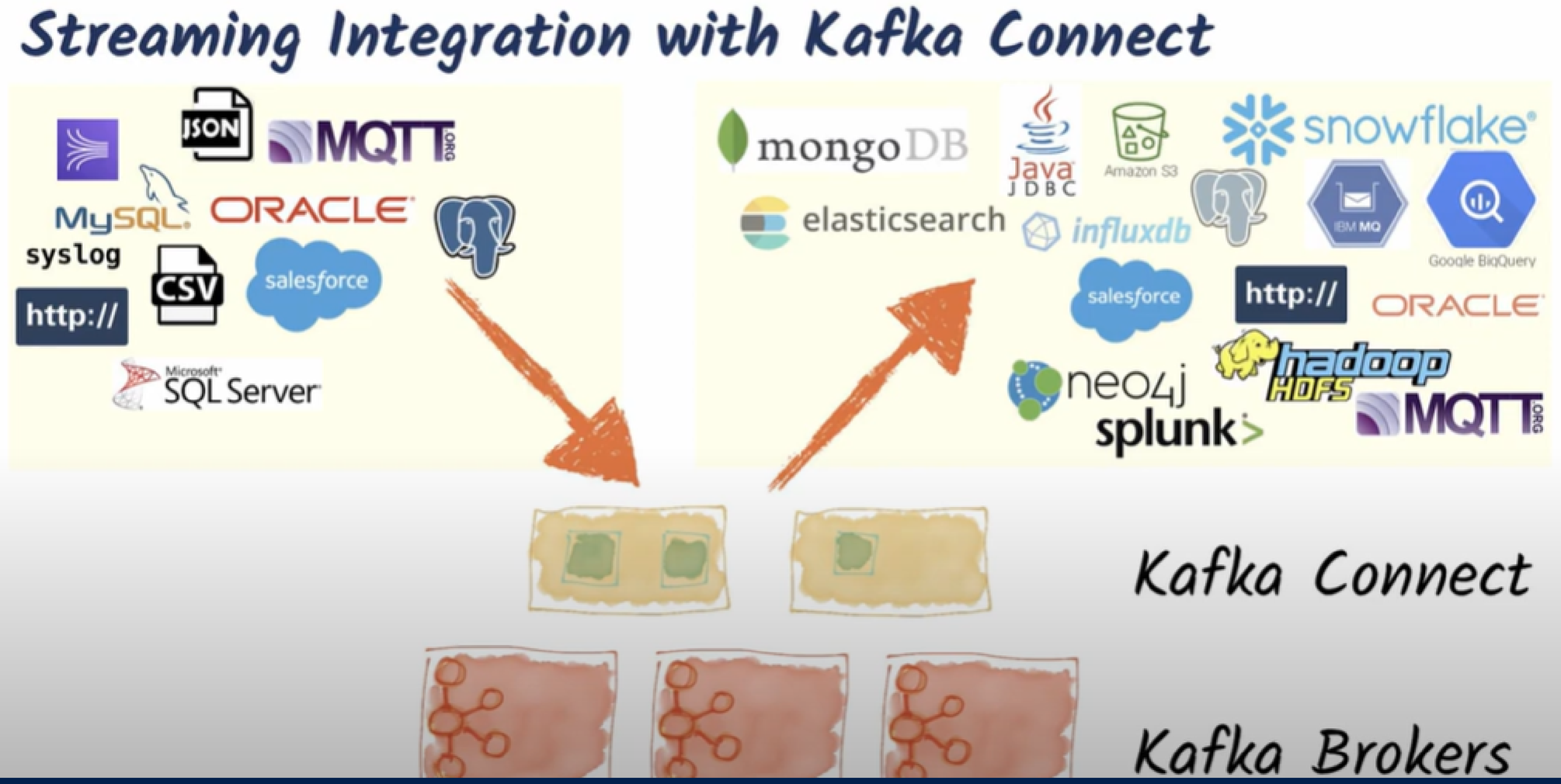
Consumer API reads messages from Kafka.

Streams API does magic with messages inside Kafka.

Manipulation

Connect API connects Kafka with other systems, like databases.

KAFKA CONNECT API



KAFKA API

- The Kafka API encompasses different types of APIs, each serving a specific purpose:

1. **Producer API:** This API allows applications to publish data (produce) to Kafka topics.
   - Producers send records to Kafka brokers, which then distribute them across partitions within topics.
2. **Consumer API:** The Consumer API enables applications to subscribe to topics and consume data produced by producers.
   - Consumers can read data from one or more partitions of a topic in parallel.

KAFKA API

3. Streams API: Kafka Streams API is used for building stream processing applications that transform input Kafka topics into output Kafka topics.

- It provides a high-level abstraction for building real-time processing applications.

4. Connector API: Kafka Connect **(External Systems)** is a framework for building and running reusable data import/export connectors between Kafka and other systems.

- The Connector API facilitates the development of connectors that move data in and out of Kafka.

KAFKA API

5. Admin API: The Admin API is used for managing and inspecting topics, brokers, and other Kafka objects.

- It allows administrators to create, delete, configure, and monitor Kafka resources programmatically.

ADMIN API

- The Admin API is used for managing and inspecting topics, brokers, and other Kafka objects.
- It allows administrators to create, delete, configure, and monitor Kafka resources programmatically.
- It provides a set of operations for creating, deleting, configuring, and monitoring Kafka resources such as topics, brokers, consumer groups, ACLs (Access Control Lists), and more.

Producer API

- **Producer API is used to send one or more messages to the same or multiple Kafka topics.**
- **There are three variants of the Producer API:**
    i. **GET /produce/$TOPIC/$MESSAGE?key=$KEY**
    ii. **[GET | POST] /produce/$TOPIC**
    iii. **[GET | POST] /produce**

1. GET /produce/$TOPIC/$MESSAGE?key=$KEY

- Sends a single message ($MESSAGE) to a topic ($TOPIC) using HTTP GET.
- Optionally message key can be appended with a query parameter?key=$KEY.

Without message key:

- curl https://tops-stingray-7863-eu1-rest-kafka.upstash.io/produce/greetings/hello_kafka -u myuser:mypass

1. GET /produce/$TOPIC/$MESSAGE?key=$KEY

With a message key:

- curl https://tops-stingray-7863-eu1-rest-kafka.upstash.io/produce/cities/Istanbul?key=city -u myuser:mypass

2. [GET | POST] /produce/$TOPIC

- Produces one or more messages to a single topic ($TOPIC).
- Messages are sent using the request body as JSON.
- Structure of the message JSON is:

```
Header {key: String, value: String}
Message{
  partition?: Int,
  timestamp?: Long,
  key?: String,
  value: String,
  headers?: Array<Header>
}
```

2. [GET | POST] /produce/$TOPIC

- It’s valid to send a single message or array of messages as JSON.

Single message with only value:

- curl https://tops-stingray-7863-eu1-rest-kafka.upstash.io/produce/greetings -u myuser:mypass \
  -d '{"value": "hello_kafka"}'

2. [GET | POST] /produce/$TOPIC

Single message with multiple attributes:

- curl https://tops-stingray-7863-eu1-rest-kafka.upstash.io/produce/cities -u myuser:mypass \
-d '{"partition": 1, "key": "city", "value": "Istanbul", "headers": [{"key": "expire", "value": "1637745834756"}] }'

2. [GET | POST] /produce/$TOPIC

- **Multiple messages with only values:**
- curl https://tops-stingray-7863-eu1-rest-kafka.upstash.io/produce/greetings -u myuser:mypass \
  -d '[
- {"value": "hello_world"},
- {"value": "hello_upstash"},
- {"value": "hello_kafka"}
-]'