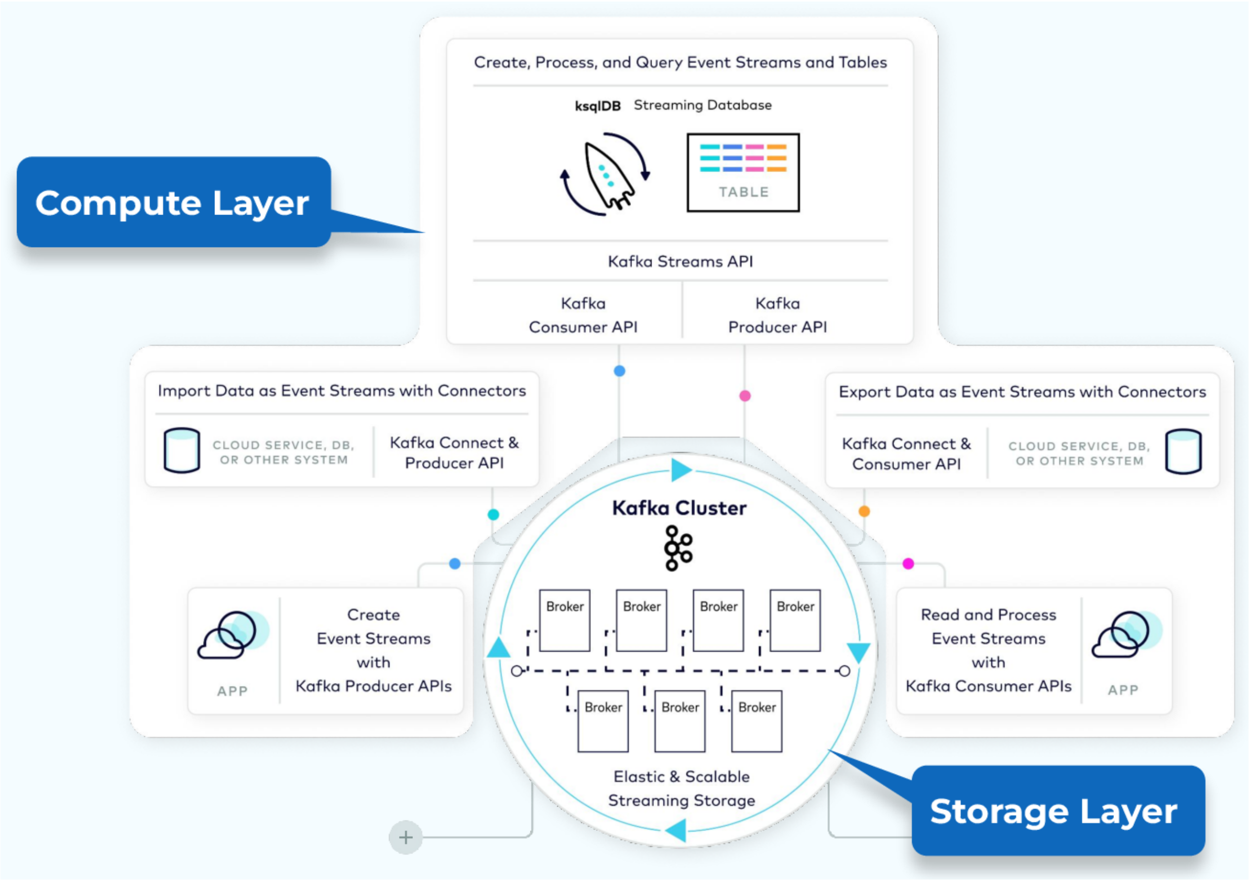
Fundamentals

Overview of Kafka Architecture


Compute Layer
Create, Process, and Query Event Streams and Tables
ksqlDB Streaming Database
TABLE
Kafka Streams API
Kafka Consumer API
Kafka Producer API
Import Data as Event Streams with Connectors
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Connect & Producer API
Export Data as Event Streams with Connectors
Kafka Connect & Consumer API
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Cluster
Broker
Broker
Broker
Broker
Broker
Broker
Broker
Elastic & Scalable Streaming Storage
APP
Create Event Streams with Kafka Producer APIs
Read and Process Event Streams with Kafka Consumer APIs
APP
Storage Layer

- Kafka is a data streaming system that allows developers to react to new events as they occur in real time
- Kafka architecture consists of a storage layer and a compute layer.
- The storage layer is designed to store data efficiently and is a distributed system such that if your storage needs grow over time you can easily scale out the system to accommodate the growth
- The compute layer consists of four core components—the producer, consumer, streams, and connector APIs, which allow Kafka to scale applications across distributed systems.

Producer and Consumer APIs

The foundation of Kafka's powerful application layer is two primitive APIs for accessing the storage.

- the producer API for writing events
- the consumer API for reading them.

Kafka Connect

Kafka Connect, which is built on top of the producer and consumer APIs, provides a simple way to integrate data across Kafka and external systems.

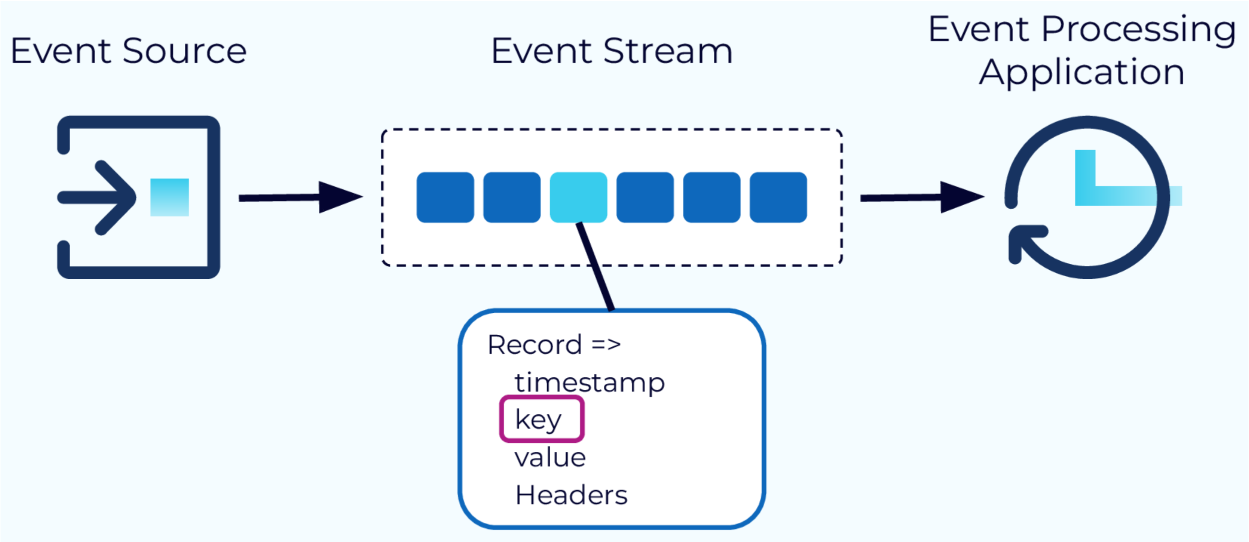
- Source connectors bring data from external systems and produce it to Kafka topics.
- Sink connectors take data from Kafka topics and write it to external systems.

Kafka Streams

For processing events as they arrive, we have Kafka Streams, a Java library that is built on top of the producer and consumer APIs.

Kafka Streams allows you to perform real-time stream processing, powerful transformations, and aggregations of event data.

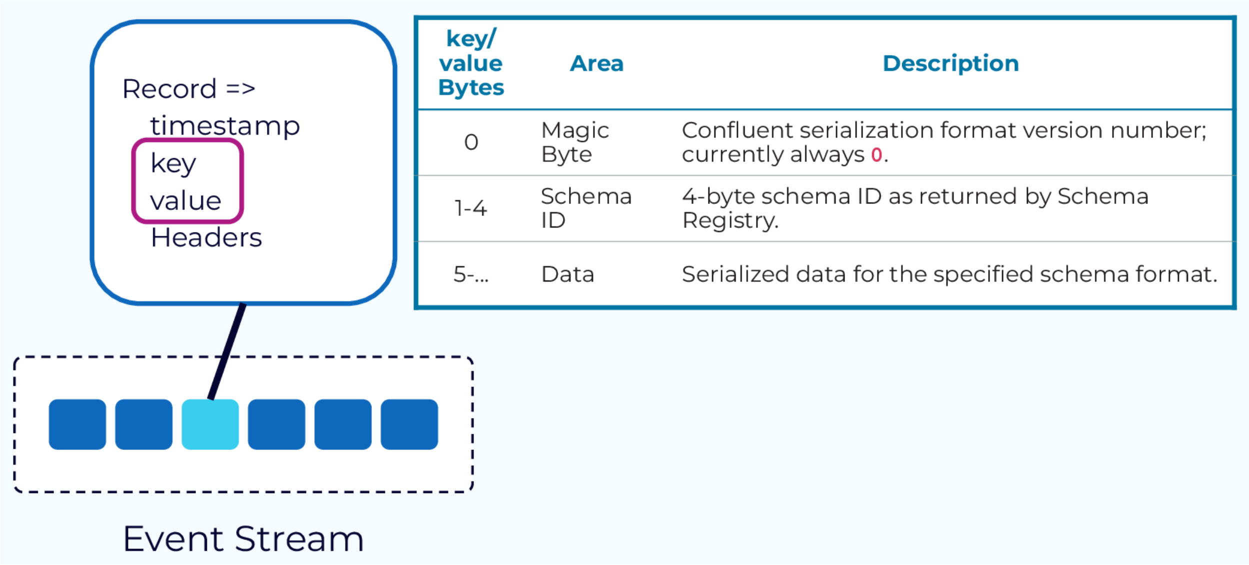
What Is an Event in Stream Processing?



- An event is a record of something that happened that also provides information about what happened.
- Examples of events are customer orders, payments, clicks on a website, or sensor readings.
- An event shouldn't be too large.
- A 10GB video is not a good event. A reference to the location of that video in an object store is.

- An event record consists of a timestamp, a key, a value, and optional headers.
- The event payload is usually stored in the value.
- The key is also optional, but very helpful for event ordering, colocating events across topics, and key-based storage or compaction.

Record Schema

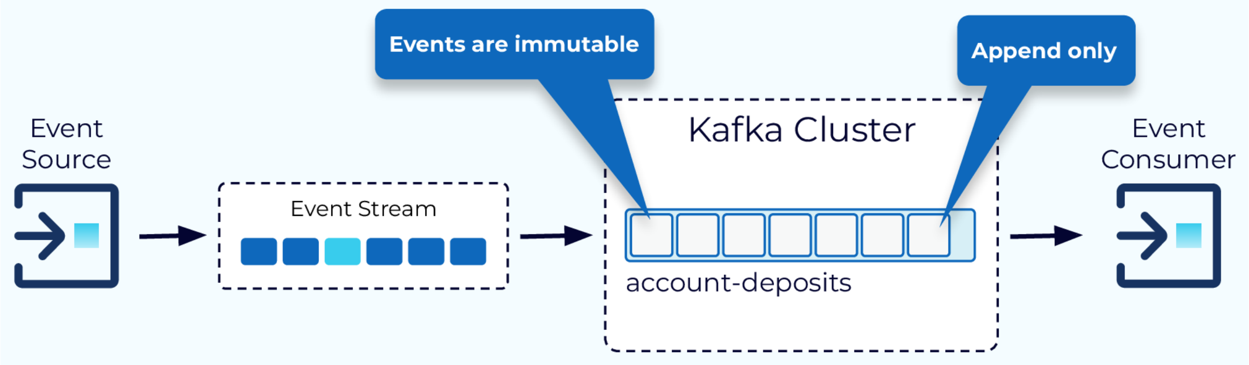


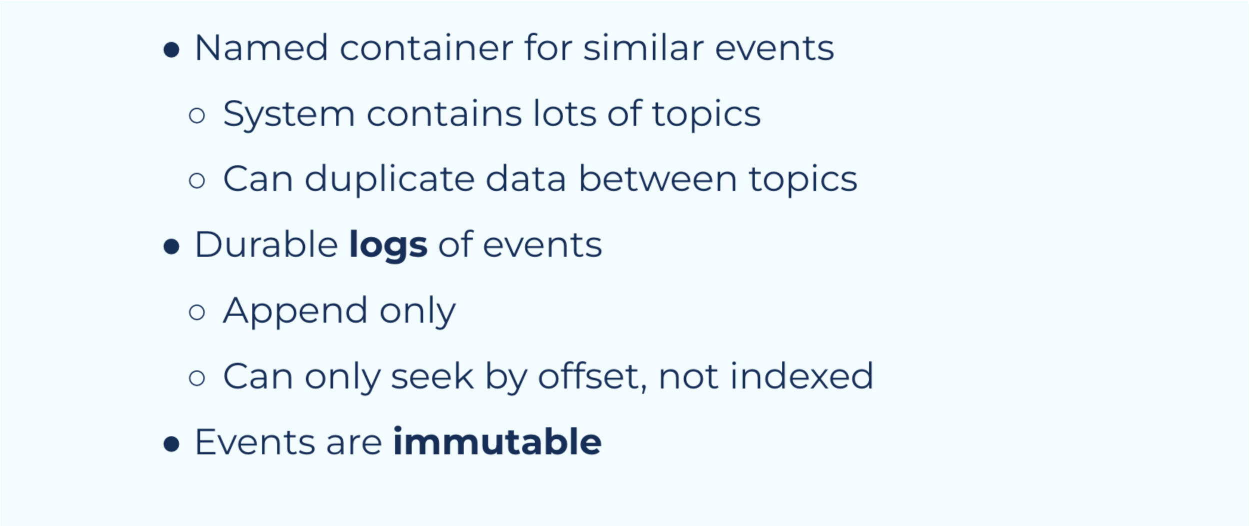
key/ value Bytes	Area	Description
0	Magic Byte	Confluent serialization format version number; currently always 0.
1-4	Schema ID	4-byte schema ID as returned by Schema Registry.
5-...	Data	Serialized data for the specified schema format.

- In Kafka, the key and value are stored as byte arrays which means that clients can work with any type of data that can be serialized to bytes.
- A popular format among Kafka users is Avro, which is also supported by Confluent Schema Registry.
- When integrated with Schema Registry, the first byte of an event will be a magic byte which signifies that this event is using a schema in the Schema Registry.

- The next four bytes make up the schema ID that can be used to retrieve the schema from the registry, and the rest of the bytes contain the event itself.
- Schema Registry also supports Protobuf and JSON schema formats.

Kafka Topics

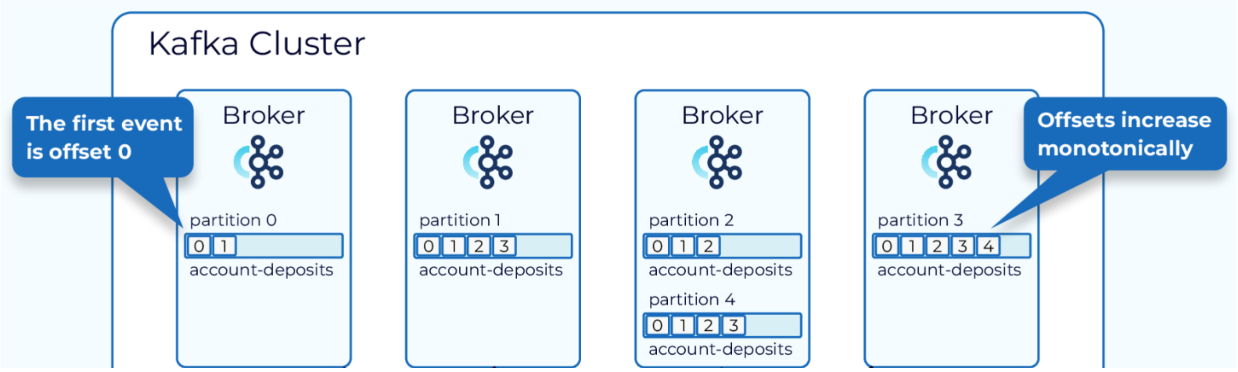




- Named container for similar events
  - System contains lots of topics
  - Can duplicate data between topics
- Durable **logs** of events
  - Append only
  - Can only seek by offset, not indexed
- Events are **immutable**

- A key concept in Kafka is the topic.
- Topics are append-only, immutable logs of events.
- Typically, events of the same type, or events that are in some way related, would go into the same topic.
- Kafka producers write events to topics and Kafka consumers read from topics.

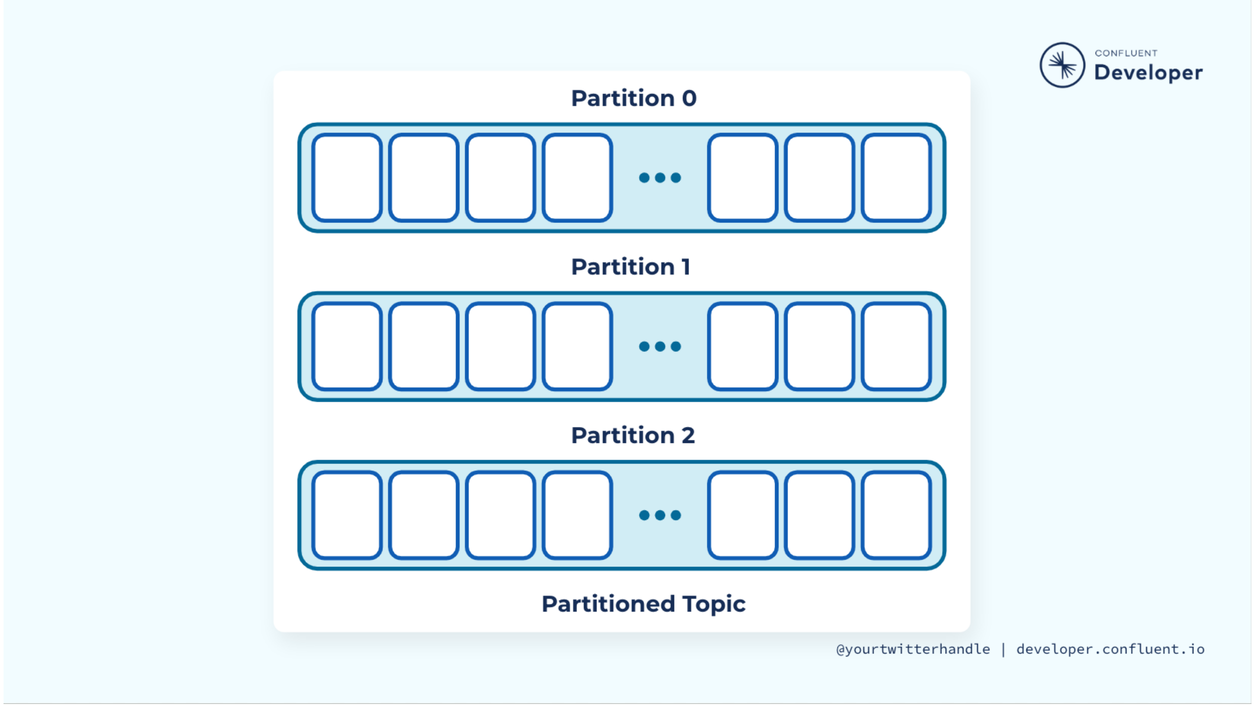
Kafka Topic Partitions

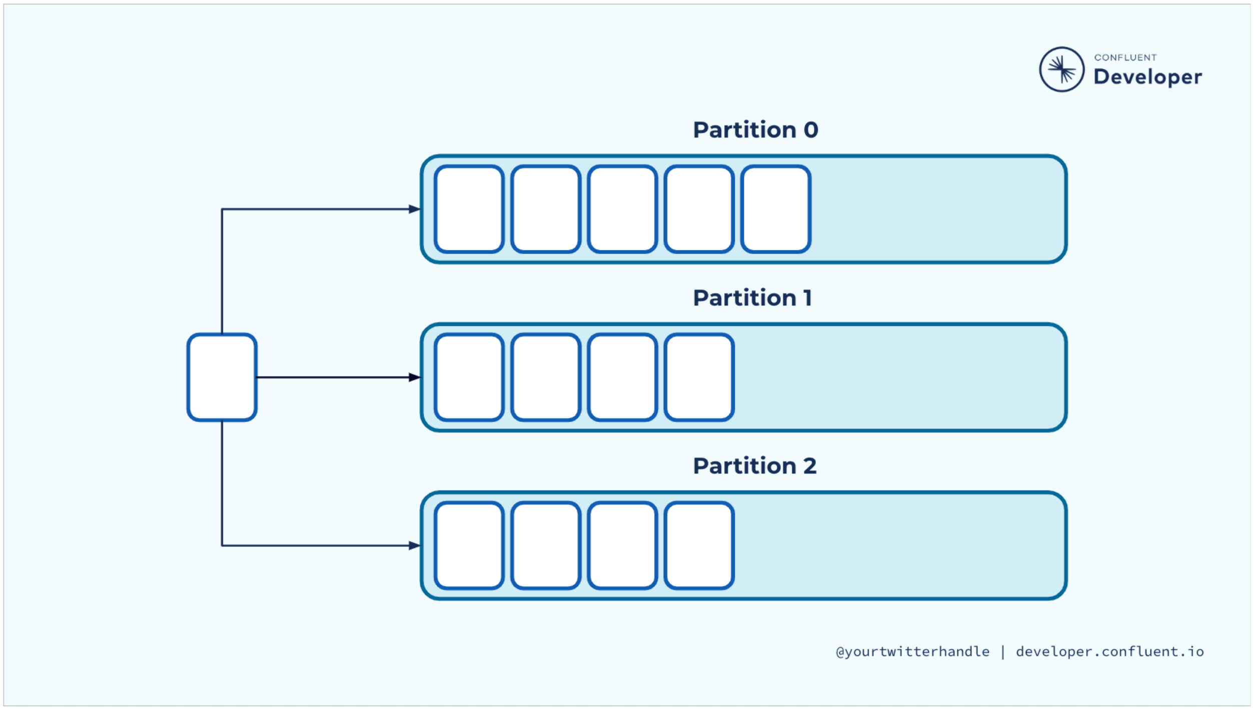


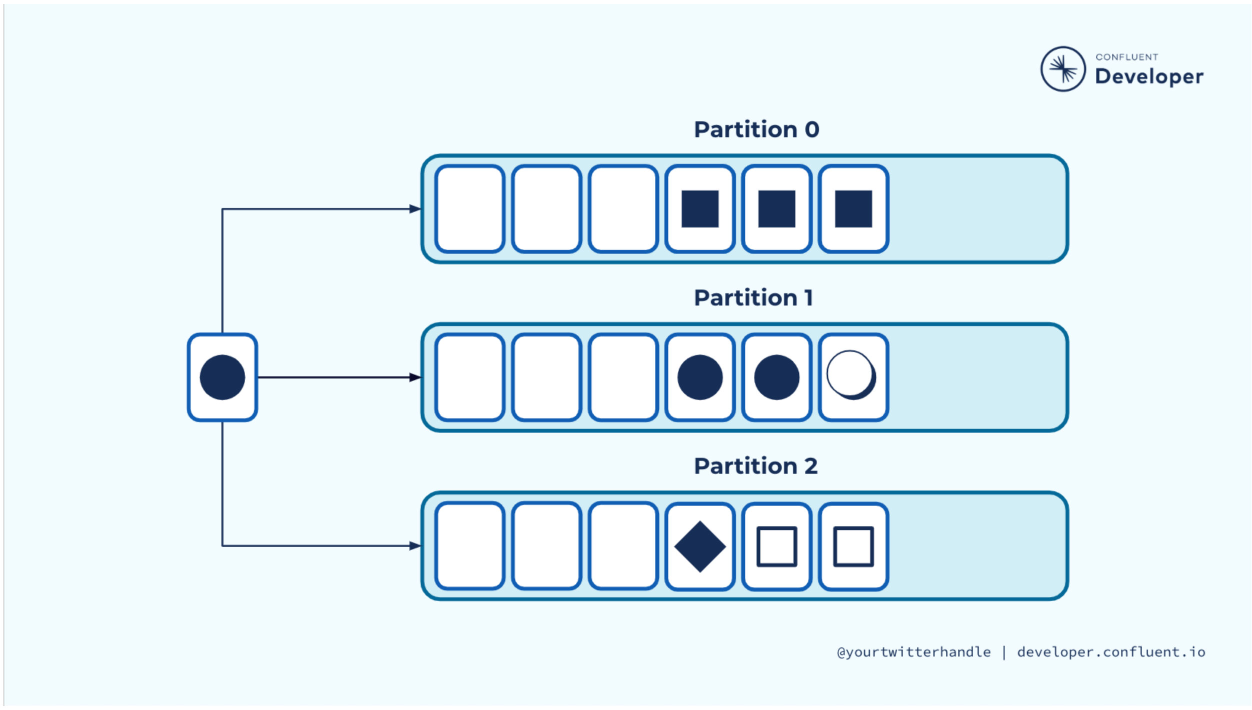
- In order to distribute the storage and processing of events in a topic, Kafka uses the concept of partitions.
- A topic is made up of one or more partitions and these partitions can reside on different nodes in the Kafka cluster.

- The partition is the main unit of parallelism.
- Events can be produced to a topic in parallel by writing to multiple partitions at the same time.
- Likewise, consumers can spread their workload by individual consumer instances reading from different partitions. If we only used one partition, we could only effectively use one consumer instance.

- Within the partition, each event is given a unique identifier called an offset.
- The offset for a given partition will continue to grow as events are added, and offsets are never reused.
- The offset has many uses in Kafka, among them are consumers keeping track of which events have been processed.




CONFLUENT
Developer
Partition 0
Partition 1
Partition 2
@yourtwitterhandle | developer.confluent.io



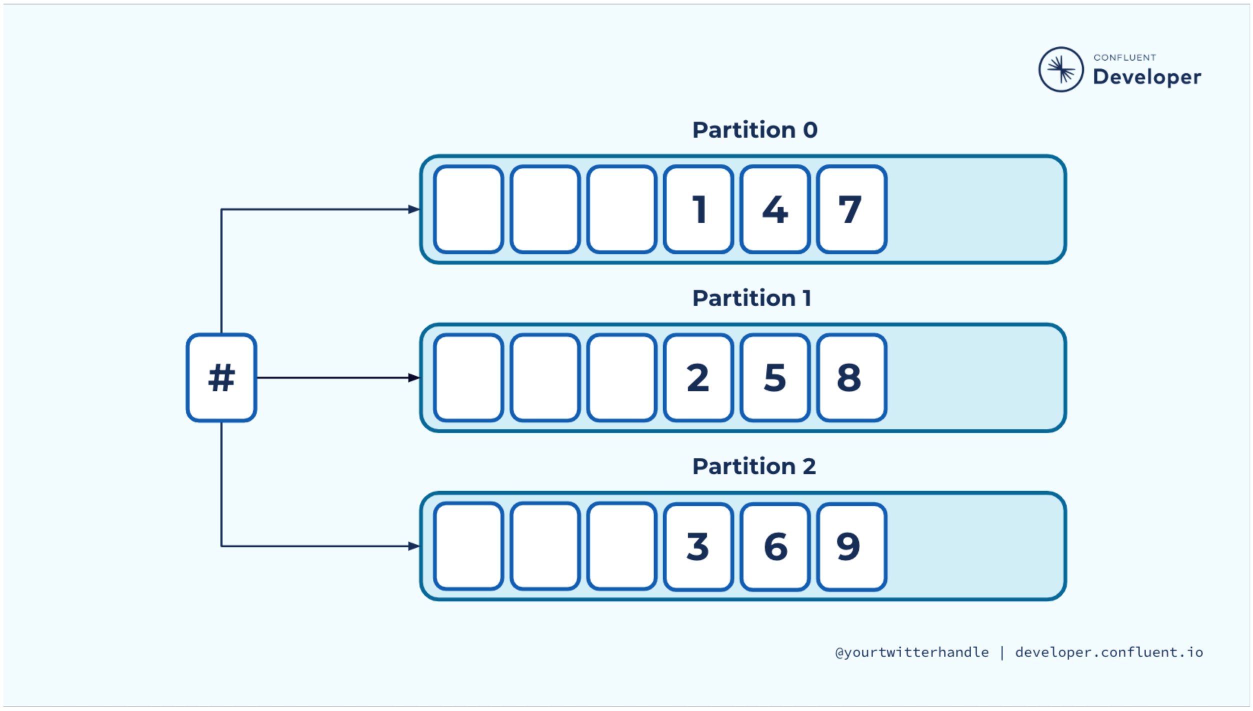
CONFLUENT Developer

Partition 0

Partition 1

Partition 2

@yourtwitterhandle | developer.confluent.io



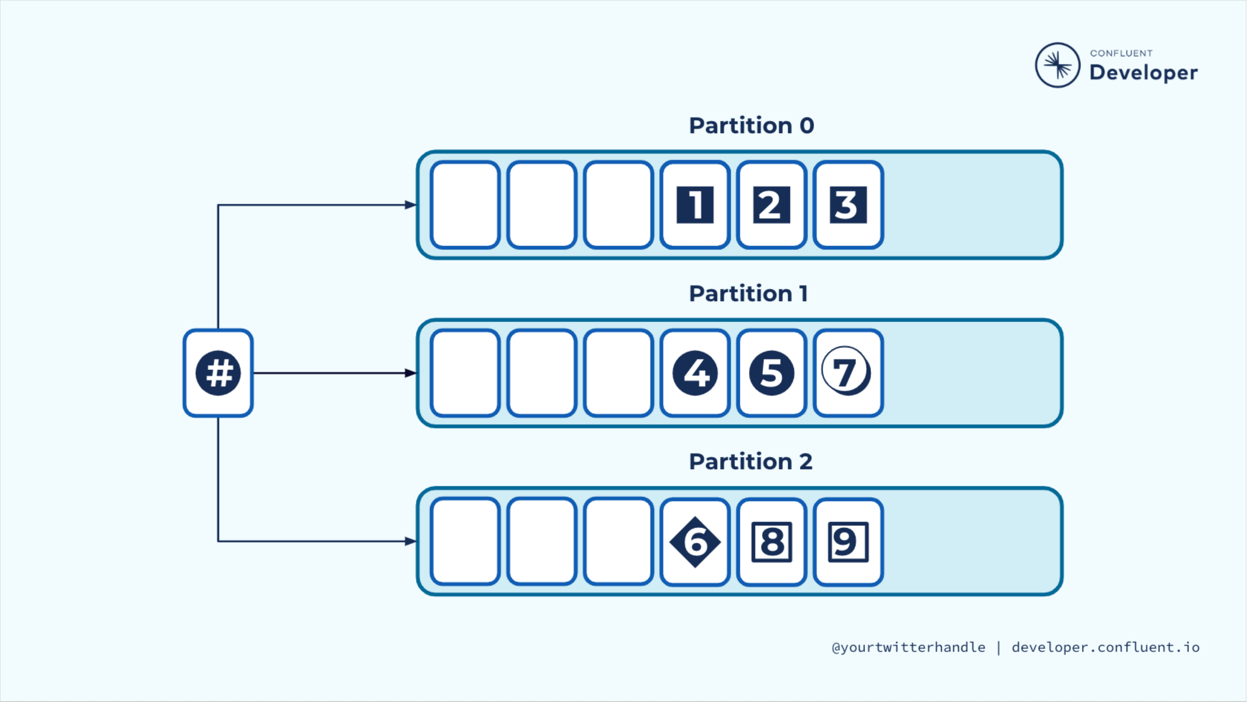
CONFLUENT Developer

Partition 0

Partition 1

Partition 2

@yourtwitterhandle | developer.confluent.io



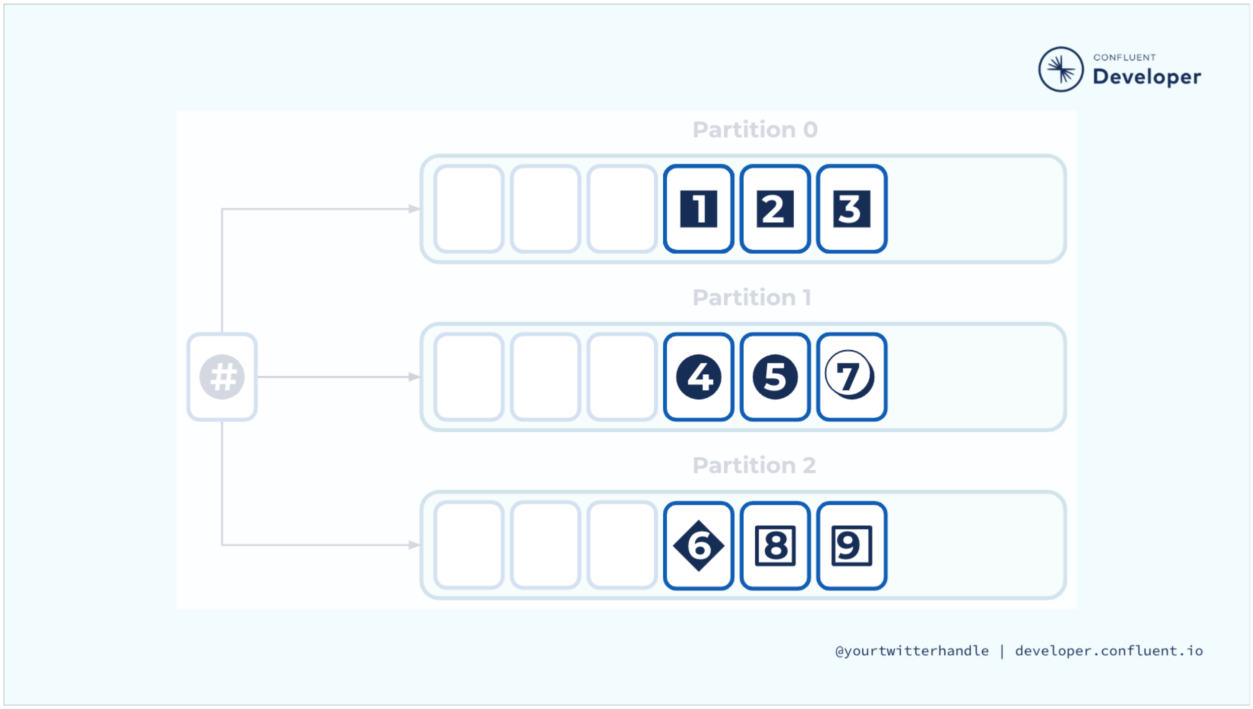
CONFLUENT Developer

Partition 0

Partition 1

Partition 2

@yourtwitterhandle | developer.confluent.io



Producer is to decide which partition to send the messages to.

Depending on various configuration and parameters, the producer decides the destination partition.

Lets check different scenario:-

1. **No key specified** :- producer will randomly decide partition and would try to balance the total number of messages on all partitions.
2. **Key Specified :-** the producer uses `Consistent Hashing` to map the key to a partition. consistent is a hashing mechanism where for the same key same hash is generated always, and it minimizes the redistribution of keys on a re-hashing scenario
3. **Partition Specified :-** You can hardcode the destination partition as well.

4. **Custom Partitioning logic:-** We can write some rules depending on which the partition can be decided.