Kafka is a distributed *event streaming platform* that lets you read, write, store, and process *events* (also called *records* or *messages* in the documentation) across many machines.

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol.

Servers: Kafka is run as a cluster of one or more servers that can span multiple data centers or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters.

Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded.

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The Admin API to manage and inspect topics, brokers, and other Kafka objects.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and process the stream of events produced to them.
- The Kafka Streams API to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams into output streams.
- The Kafka Connect API to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka.

Distributed Message broker
Abstraction over distributed commit log

Features:
    High throughput
    Low latency

Topic: Order collection of events stored in a durable way(replication)

A topic is similar to a folder in a filesystem, and the events are the files in that folder.

A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever.

Events are durably stored in Kafka, they can be read as many times and by as many consumers as you want.

Once your data is stored in Kafka as events, you can process the data with the Kafka Streams client library for Java/Scala.

The library supports exactly-once processing, stateful operations, and aggregations, windowing, joins, processing based on event-time, and much more.

event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Kafka Connect

Grouping, filtering, aggregating

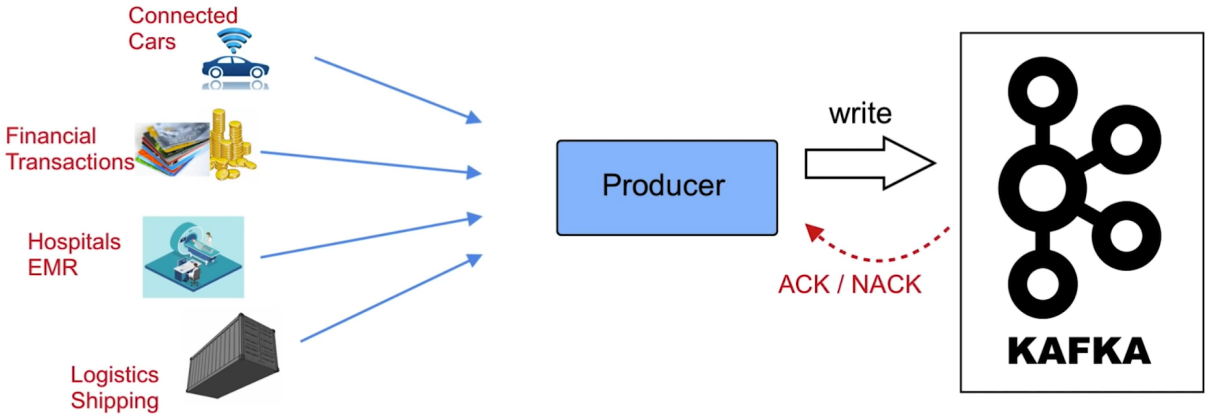
Kafka No Longer Requires ZooKeeper

`KRaft mode 2.8.0`

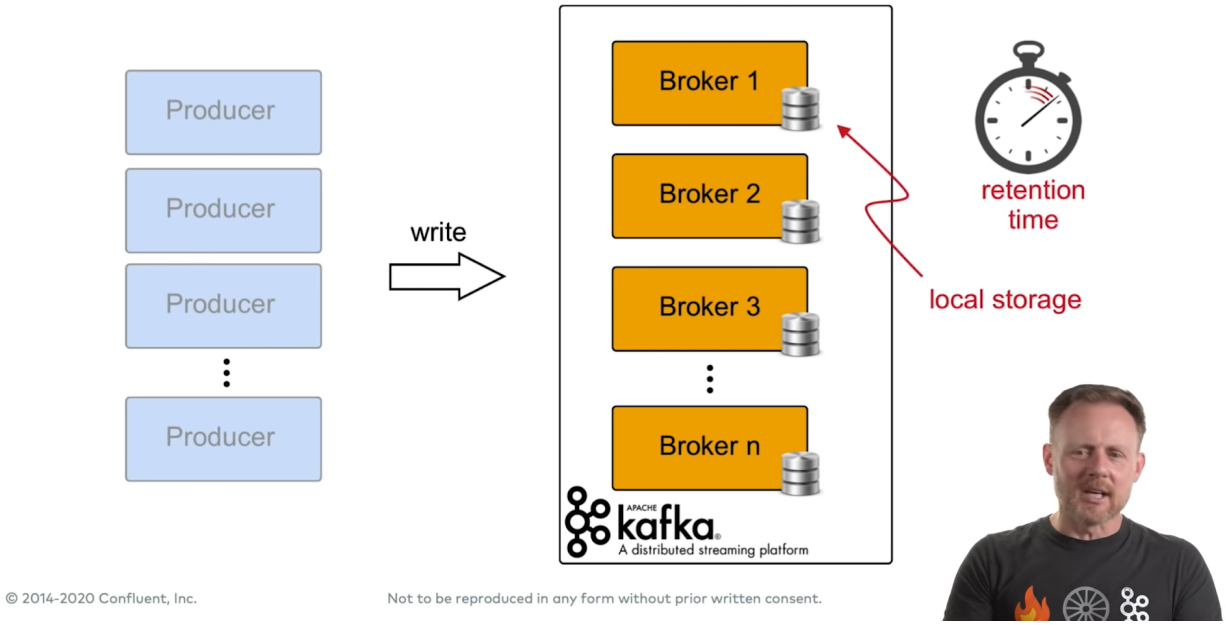
Apache ZooKeeper was used by Kafka as a metadata store. Metadata for partitions and brokers were stored to the ZooKeeper quorum that was also responsible for Kafka Controller election.

Producers send a message to a broker, after receiving broker stores them on disk keyed by unique offset. Moreover, by topic, partition and offset a broker allows consumers to fetch messages.

Producers

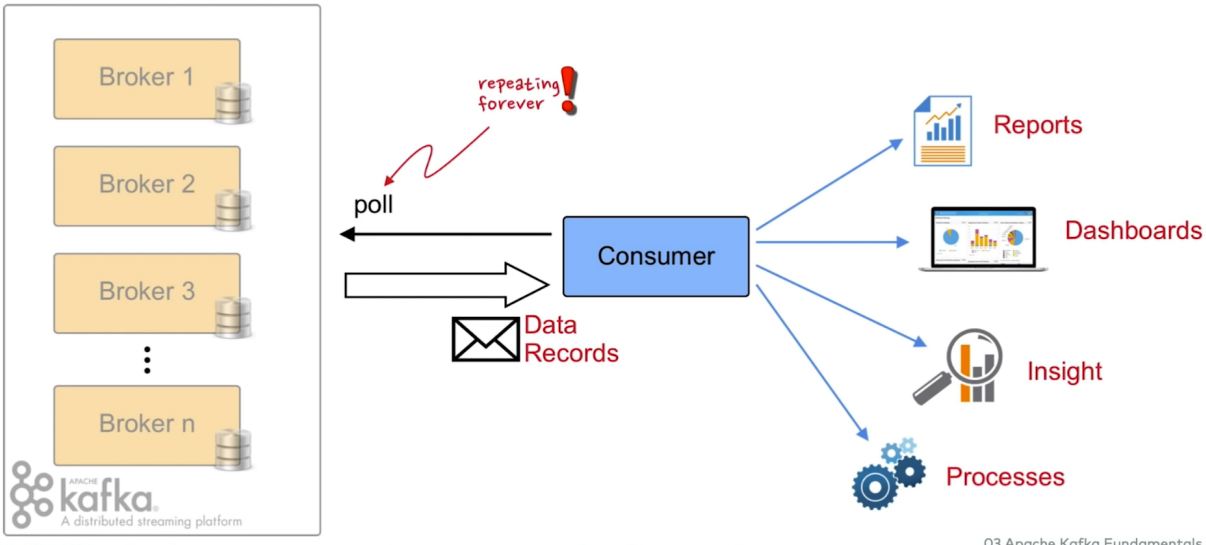


Kafka Brokers

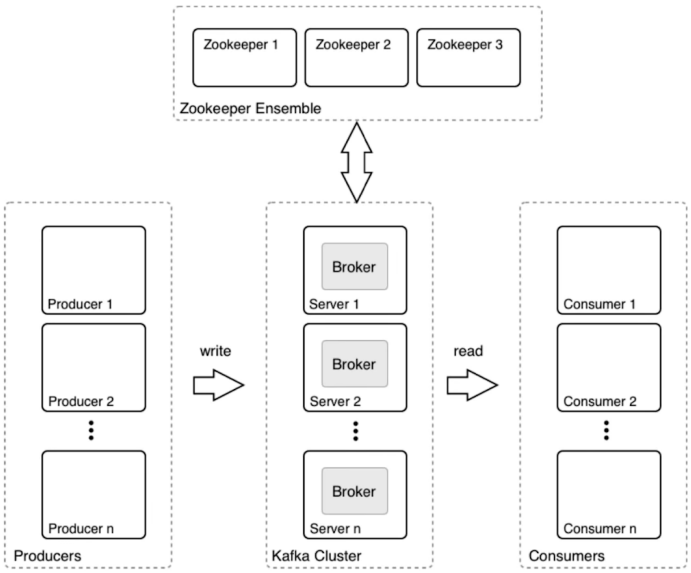


Kafka cluster consists of brokers(machine/VMS/containers/etc).

Consumers



Architecture



Decoupling of producers and consumers

Zookeeper -

Access control and secrets

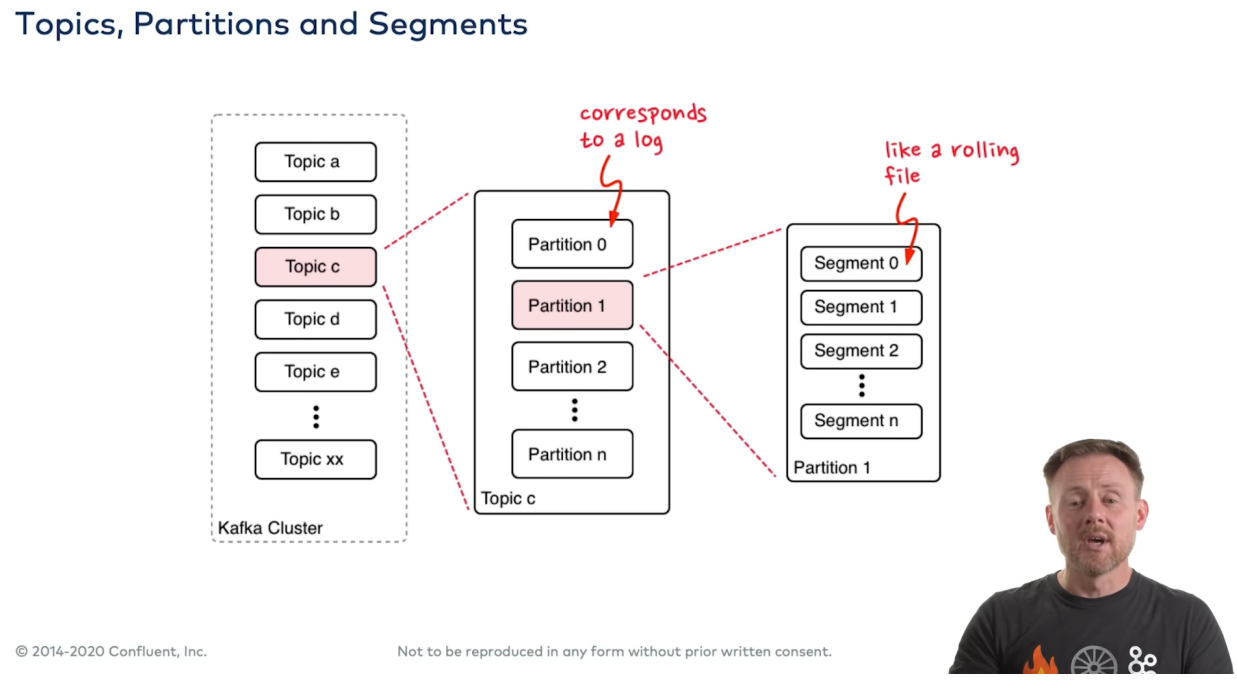
Management of cluster

Failure and recovery

Leader election

Topics: Logical Representation

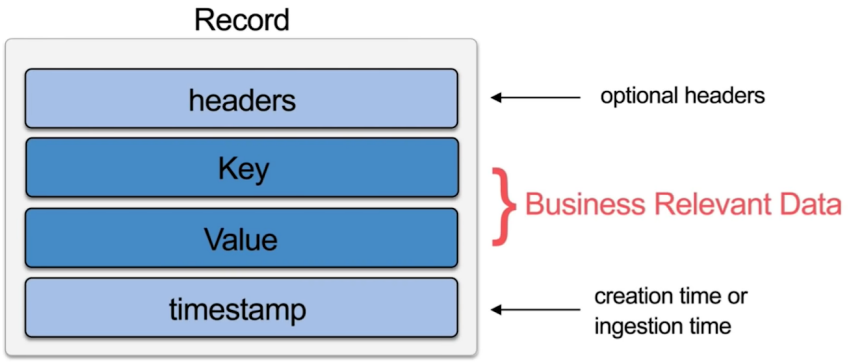
Topic, partitions and segments



Kafka cluster will not keep track of partition size, move them around when broker is overloaded. We need to do it ourself.

Offset

Data Elements



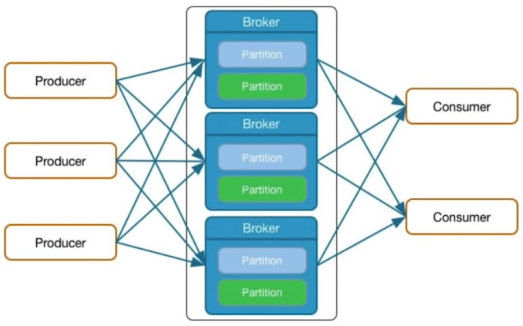
Key/Value can be complex objects as well

Brokers Manage Partitions

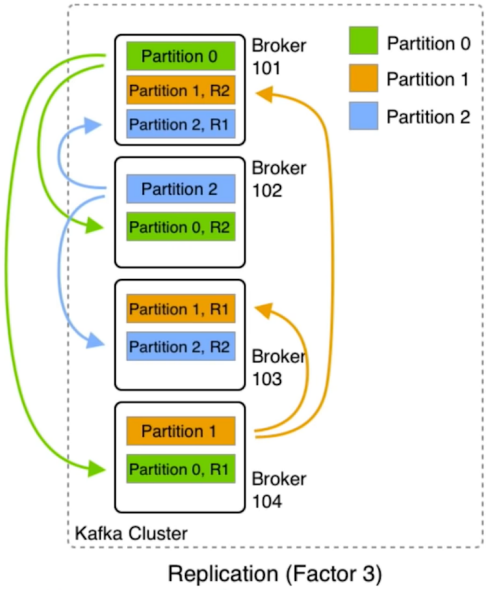
- Messages of Topic spread across Partitions
- Partitions spread across Brokers
- Each Broker handles many Partitions
- Each Partition stored on Broker's disk
- Partition: 1..n `log` files
- Each message in Log identified by *Offset*
- Configurable Retention Policy

Broker Basics

- Producer sends Messages to Brokers
- Brokers receive and store Messages
- A Kafka Cluster can have many Brokers
- Each Broker manages multiple Partitions



Broker Replication



Replication (Factor 3)

Leader and follower

Producer connects to leader to write the messages/events

Load Balancing and Semantic Partitioning

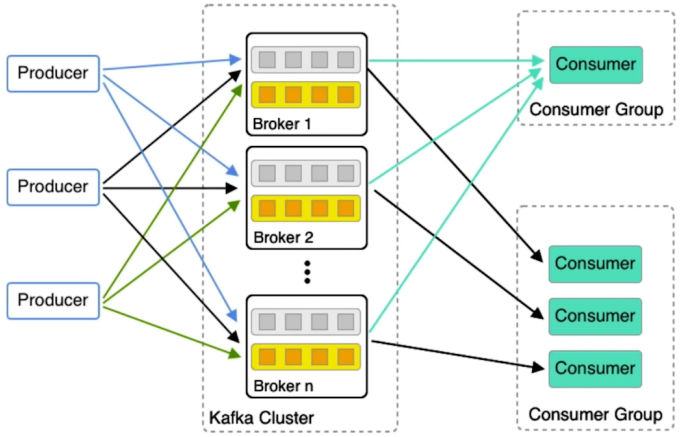
- Producers use a Partitioning Strategy to assign each Message to a Partition
- Two Purposes:
  - Load Balancing
  - Semantic Partitioning
- Partitioning Strategy specified by Producer
  - Default Strategy: `hash(key) % number_of_partitions`
  - No Key → Round-Robin
- Custom Partitioner possible

For the above, when the number_of_partitions change the strategy will change.

Consumer Basics

- Consumers **pull** messages from 1...n topics
- New inflowing messages are automatically retrieved
- Consumer offset
  - keeps track of the last message read
  - is stored in special topic
- CLI tools exist to read from cluster

Scalable Data Pipeline



Consumer group

Properties:

Bootstrap Server: Tells where the cluster(contains brokers) is

Producers and consumers need to interact with leaders. After 2.5 consumers can interact with followers.

Leader/follower at the partition level

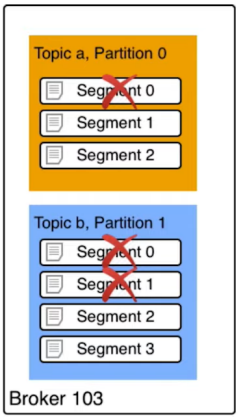
A partition can have a leader and any number of followers.

Data Retention Policy

How long do I want or can I store my data?

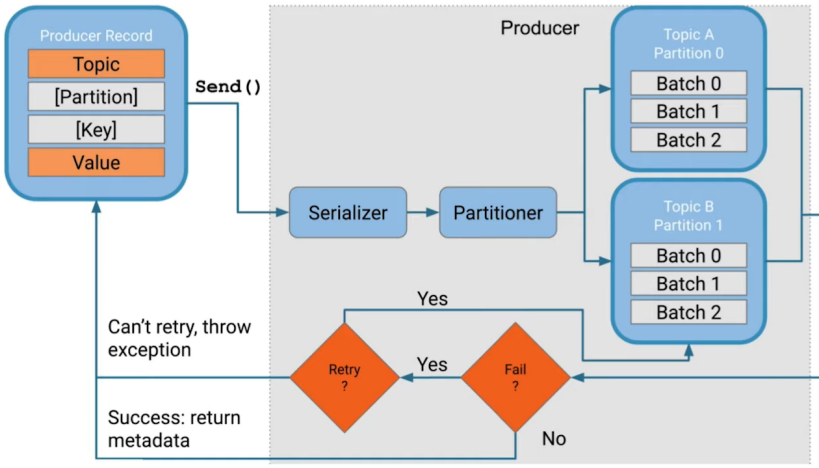
- How long (default: `1 week`)
- Set **globally** or **per topic**
- Business decision
- Cost factor
- Compliance factor → GDPR

Data purged per segment



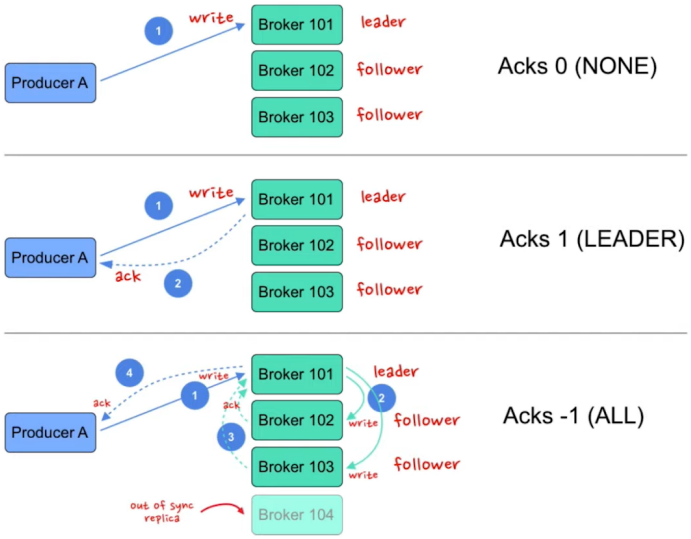
Retention Policy

Producer Design

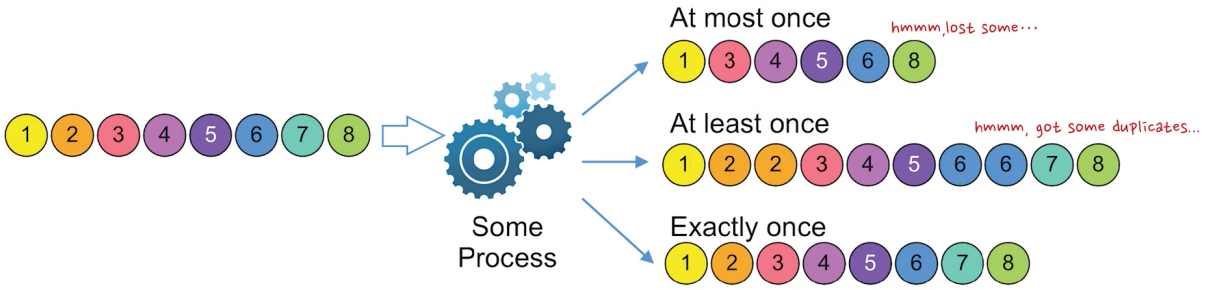


Batching exists

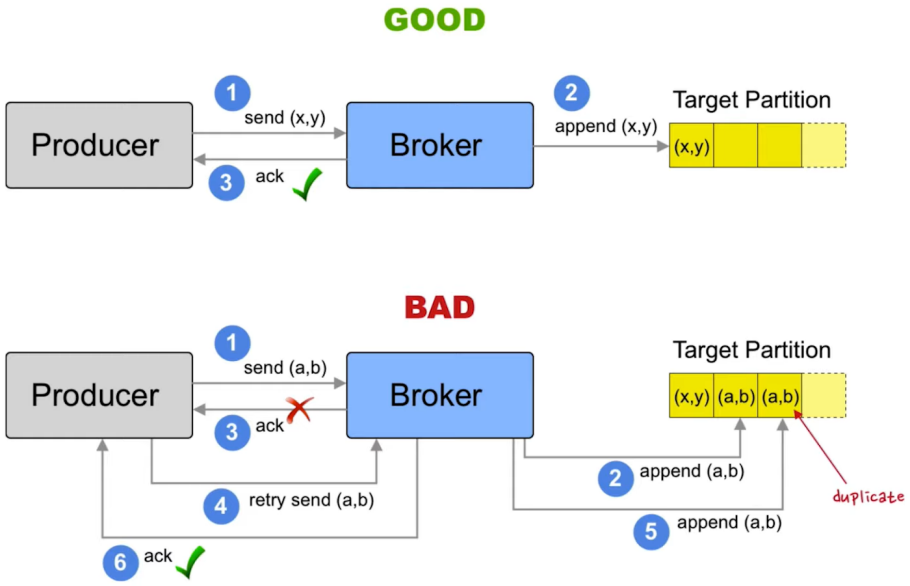
Producer Guarantees



Delivery Guarantees



Idempotent Producers



Exactly Once Semantics

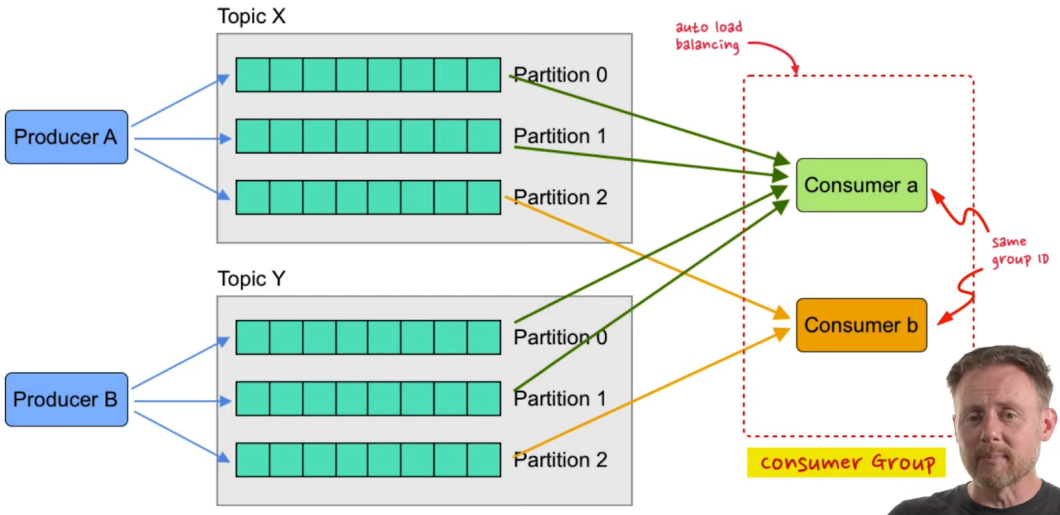
What?

- Strong **transactional guarantees** for Kafka
- Prevents clients from processing duplicate messages
- Handles failures gracefully

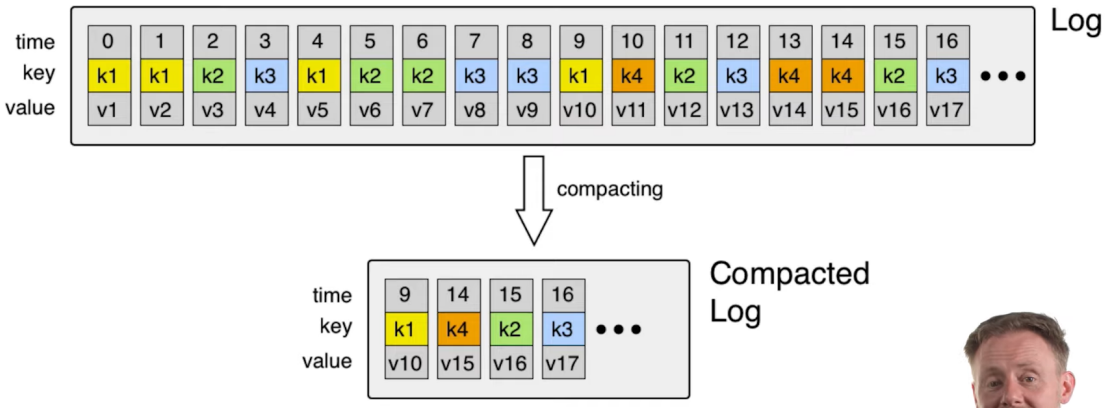
Use Cases

- Tracking ad views
- Processing financial transactions
- Stream processing

Consumer Groups



Compacted Topics



Security Overview

- Kafka supports Encryption in Transit
- Kafka supports Authentication and Authorization
- No Encryption at Rest out of the box!
- Clients can be mixed with & without Encryption & Authentication

⚠ Using Security is optional - BUT!

A **consumer group** is a set of consumers which cooperate to consume data from some topics. The partitions of all the topics are divided among the consumers in the group. As new group members arrive and old members leave, the partitions are re-assigned so

that each member receives a proportional share of the partitions. This is known as rebalancing the group