Apache Kafka: Distributed Event Streaming

Big Data Analytics - Lecture 11

Dr. Tariq Mahmood

November 10, 2025

Contents

1 Introduction to Apache Kafka

1.1 Definition and Origin

Apache Kafka is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

- **Origin:** Originally developed at LinkedIn to handle massive amounts of real-time data feeds.
- **Evolution:** It was open-sourced and donated to the Apache Software Foundation.
- **Core Function:** It acts as a distributed commit log, allowing applications to publish (write) and subscribe to (read) streams of records in real-time.

1.2 Core Capabilities

Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1. **Publish and Subscribe:** Similar to a message queue or enterprise messaging system, it allows the reading and writing of streams of events.
2. **Storage:** It stores streams of events durably and reliably for as long as you want (retention).
3. **Processing:** It processes streams of events as they occur or retrospectively (using Kafka Streams or ksqlDB).

2 System Architecture

Kafka's architecture relies on a decoupled design involving Producers, Consumers, and the Kafka Cluster itself.

2.1 Fundamental Concepts

2.1.1 Topics

A **Topic** is a category or feed name to which records are stored.

- **Analogy:** It is similar to a table in a database or a folder in a filesystem.
- **Examples:** `user_logs`, `clickstream`, `orders`.
- **Nature:** Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

2.1.2 Partitions

Topics are split into **Partitions**. This is the primary mechanism for scalability.

- **Structure:** Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log.
- **Indexing (Offsets):** The records in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each record within the partition.
- **Parallelism:** Partitions allow a topic to be parallelized by splitting the data across multiple brokers.

2.1.3 Brokers

A Kafka cluster consists of one or more servers known as **Brokers**.

- **Role:** Brokers receive messages from producers, assign offsets, and commit messages to storage on disk. They serve fetch requests from consumers.
- **Cluster:** A cluster is a group of brokers working together.

2.2 The Ecosystem Components

- **Producers:** Applications that publish (write) data to topics. They choose which record to assign to which partition within the topic (often based on a key hash).
- **Consumers:** Applications that subscribe to (read) topics and process the published messages.
- **ZooKeeper (Legacy) / KRaft (Modern):** Used for managing cluster metadata, controller election, and configuration. Modern Kafka is moving toward KRaft (Kafka Raft Metadata mode) to remove the ZooKeeper dependency.
- **Kafka Connect:** A tool for scalably and reliably streaming data between Kafka and other systems (e.g., databases, S3).
- **Schema Registry:** A serving layer for your metadata. It stores a versioned history of schemas (Avro, JSON, Protobuf) to ensure producer/consumer compatibility.

3 Data Flow and Partitioning Logic

3.1 Producer Behavior

When a producer sends a message, it includes a Key and a Value.

- **Key-based Partitioning:** If a key is provided, Kafka hashes the key to determine the target partition. This guarantees that all messages with the same key (e.g., `user_id=101`) will go to the same partition. This is crucial for ordering guarantees.
- **Round-robin:** If no key is provided, the producer cycles through partitions to balance the load.

Example Logic:

$$\text{Target Partition} = \text{hash}(\text{Key}) \pmod{\text{Number of Partitions}}$$

3.2 Consumer Groups

To facilitate scalable message consumption, Kafka uses the concept of **Consumer Groups**.

- **Load Balancing:** All consumers in a group share the same `group_id`. Kafka divides the topic partitions among the members of the group.
- **Rule of Thumb:** A partition is consumed by exactly one consumer within a specific group.
- **Broadcast:** If you want multiple applications to read the same data independently, they must be in *different* consumer groups.

4 Reliability and Fault Tolerance

Kafka provides robustness through replication.

4.1 Replication

Each partition has a configured **replication factor** (e.g., 3).

- **Leader:** One broker is the designated "Leader" for a partition. It handles all reads and writes.
- **Followers:** Other brokers hold "Follower" replicas. They passively replicate the leader's log.
- **Failover:** If the leader broker crashes, one of the in-sync followers is automatically elected as the new leader.

4.2 ISR (In-Sync Replicas)

An ISR is a replica that has fully caught up with the leader.

- A message is considered "committed" only when all ISRs have written it to their transaction log.
- This ensures that data is not lost if the leader fails immediately after an acknowledgement.

4.3 Delivery Guarantees

1. **At-most-once:** Messages may be lost but are never redelivered. (Lowest overhead).
2. **At-least-once:** Messages are never lost but may be redelivered. (Standard default; requires idempotency handling in consumer).
3. **Exactly-once:** Each message is delivered once and only once. (Achieved via Kafka Transactions and Idempotent Producers).

5 Message Queuing Concepts

Kafka functions as a highly scalable message queue, offering distinct advantages over monolithic architectures.

5.1 Benefits of Message Queues

- **Asynchronous Processing:** The producer does not block waiting for the consumer to process data.
- **Decoupling:** Producers and Consumers act independently. A change in one does not break the other.
- **Load Leveling:** During traffic spikes, the queue buffers requests, allowing consumers to process at their own pace without crashing.
- **Fault Tolerance:** If a consumer crashes, the messages persist in the queue until the consumer recovers.

5.2 Comparison with Other Tools

- **RabbitMQ:** Ideal for complex routing logic using AMQP. Lower throughput than Kafka.
- **ActiveMQ:** Traditional JMS broker. Good for enterprise integration patterns.
- **Redis Streams:** Lightweight, in-memory, very fast but less durable for massive historical storage.
- **Kafka:** Optimized for high throughput, log storage, replayability, and massive scale.

6 Consumer Lag

Consumer lag is a critical metric in Kafka operations. It indicates performance issues in the processing pipeline.

6.1 Definition

Consumer Lag is the difference between the latest message written to a partition and the last message processed (committed) by a consumer.

$$\text{Consumer Lag} = \text{Latest Offset (Log End)} - \text{Last Committed Offset} \tag{1}$$

6.2 Example Scenario

Consider a topic `orders` with 1 partition.

- Producer writes messages M1 through M5 (Offsets 0 to 4).
- Consumer processes M1, M2, M3 (commits Offset 2).
- **Calculation:**

$$\text{Log End Offset} = 4 \quad \text{(Message M5)}$$

$$\text{Committed Offset} = 2 \quad \text{(Message M3)}$$

$$\text{Lag} = 4 - 2 = 2 \text{ messages waiting.}$$

6.3 Causes of High Lag

- **High Producer Rate:** Producers are generating data faster than consumers can process it.
- **Slow Consumer:** The processing logic (e.g., database writes, complex computations) takes too long.
- **Rebalancing:** When a consumer joins/leaves a group, the group "rebalances," pausing consumption temporarily.
- **Infrastructure Issues:** Network latency or slow disk I/O on the consumer side.

7 Practical Implementation

7.1 Docker Compose Setup

To run Kafka locally, a `docker-compose.yml` file is often used to spin up ZooKeeper and Kafka.

```yaml
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:8.1.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
  kafka:
    image: confluentinc/cp-kafka:8.1.0
    depends_on: [zookeeper]
    ports:
      - "9092:9092"
    environment:
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Listing 1: Docker Compose Configuration

7.2 CLI Operations

Common administrative tasks using the Kafka CLI:

```bash
# Create a topic named 'test'
kafka-topics --create --topic test --bootstrap-server localhost:9092 \
--partitions 3 --replication-factor 1

# Start a console producer
kafka-console-producer --topic test --bootstrap-server localhost:9092

# Start a console consumer (reading from the beginning)
kafka-console-consumer --topic test --bootstrap-server localhost:9092 \
--from-beginning
```

Listing 2: Kafka CLI Commands

7.3 Python Implementation (kafka-python)

7.3.1 Producer Code

```python
from kafka import KafkaProducer

# Initialize producer connected to local broker
producer = KafkaProducer(bootstrap_servers='localhost:9092')

# Send 10 messages
for i in range(10):
    # Encode message as bytes
    message_val = f'msg-{i}'.encode()
    producer.send('test', key=b'key', value=message_val)

# Ensure all messages are sent
producer.flush()
```

Listing 3: Python Producer

7.3.2 Consumer Code

```python
from kafka import KafkaConsumer

# Initialize consumer for topic 'test'
consumer = KafkaConsumer(
    'test',
    bootstrap_servers='localhost:9092',
    auto_offset_reset='earliest', # Start from beginning if no offset found
    group_id='g1'                 # Consumer Group ID
)

# Process messages
for msg in consumer:
    print(f"Offset: {msg.offset}, Key: {msg.key}, Value: {msg.value}")
```

Listing 4: Python Consumer

8 Monitoring Tools

To ensure the health of a Kafka cluster, several metrics must be tracked, including Broker health, Disk usage, and specifically Consumer Lag. Common tools include:

- **Prometheus + JMX Exporter:** For scraping metrics from Kafka JVMs.
- **Grafana:** For visualizing metrics.
- **Confluent Control Center:** A comprehensive GUI for managing and monitoring Kafka.
- **Burrow:** A tool developed by LinkedIn specifically to monitor consumer lag without requiring consumer cooperation.