1 Introduction

Purpose of Kafka: Event Streaming.
Capture data in real-time from event sources (e.g. Databases) → (1) Storing these event streams durably for later retrieval or (2) Manipulating or reacting to event streams in real-time (3) Routing event streams to different destinations.

2 Topics and Partitions

Messages/Events are categorised into topics. Topics are synonymous with "database table" or "folder" in a file system. Each topic is broken down into Partitions. Partition is synonymous with "Bucket" or "Single Log", in which messages are "committed to the log" in a (1) append-only fashion and (2) read in the order from beginning to end.

Additionally, messages with the same key are stored in the same partition.

Lastly, to make data fault-tolerant, every topic can be replicated.

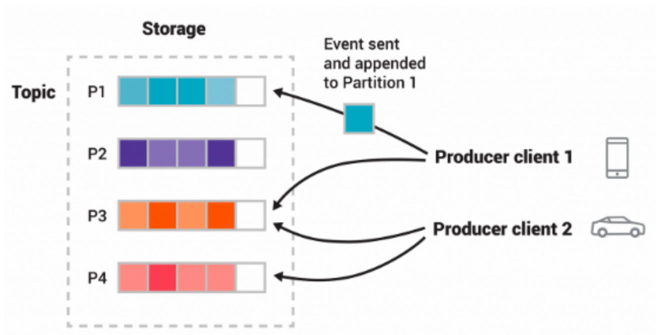


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

3 Brief Introduction to Producers and Consumers

The core of Kafka is broken down into two basic units: Producers and Consumers. Both units follow a typical Publish-and-Subscribe pattern.

Producers create new messages/events and are equivalent to “Publisher”. A message will be produced on a specific “Topic”.

Consumers read messages/events and are equivalent to “Subscriber”. A consumer can subscribe to one or more topics and read the messages in the order in which they are produced. Additionally, consumers have one key feature of maintaining the next possible “offset” metadata. The next possible “offset” helps consumer(s) to be fault-tolerant, allowing it to crash and restart without losing the position of the data to be processed.

4 Brokers and Clusters

A single Kafka server is called a broker. A broker receives messages from producers, assigns offsets to them, and writes them to storage on disk.

Kafka brokers operate within a cluster. Within the cluster of brokers, one of the brokers will be elected as the cluster controller or leader (leader election).

All producers must connect to the leader to publish the message. However, consumers can fetch from brokers (leader or follower).

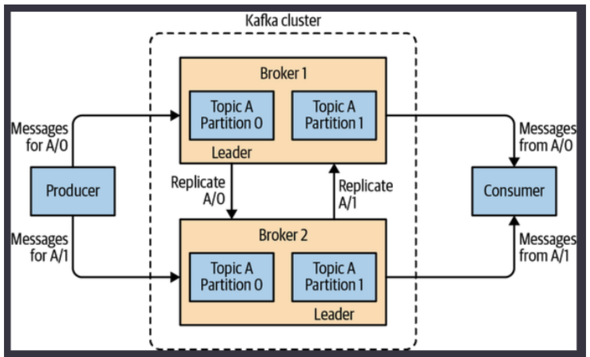


Figure 1-7. Replication of partitions in a cluster

5 Core APIS

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 more Kafka topic.
- The Consumer API to subscribe to (read) one or more topics.
- The Kafka Streams API to implement stream processing applications and microservices. One example might be performing aggregations using the word count algorithm.
- The Kafka Connect API allows implementing connectors that continuously pull from the source data system into Kafka or push from Kafka into the sink data system.

6 Kafka Producer

[More on Kafka Producer in part 2]

7 Kafka Consumer

7.1 Consumer and Consumer Group

Kafka consumers are typically part of a consumer group. If we are limited to a single consumer reading and processing data, it is possible that the single consumer will fail due to its inability to keep up with the rate of incoming message.

As such, there's a need to scale the consumption from topics, by increasing the number of consumers.

Here are four possible scenarios:

Scenario 1: One consumer with four partitions. All messages will be sent to consumer 1.

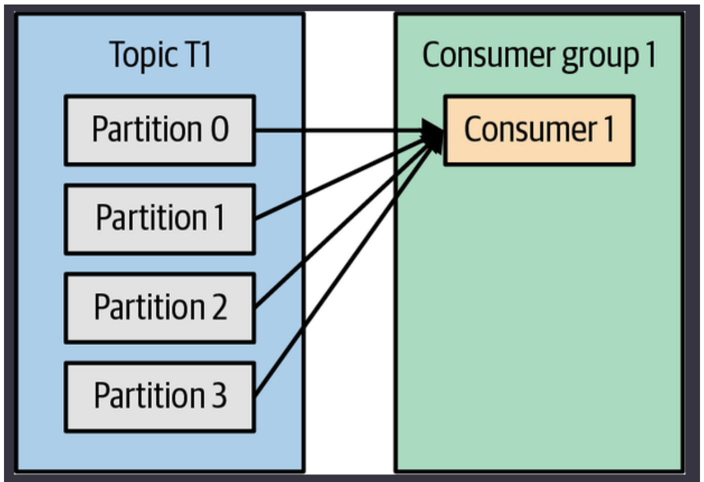


Figure 4-1. One consumer group with four partitions

Scenario 2: It is possible that, Consumer 1 will retrieve message from partition 0 and partition 2; and Consumer 2 will retrieve message from partition 1 and 3.

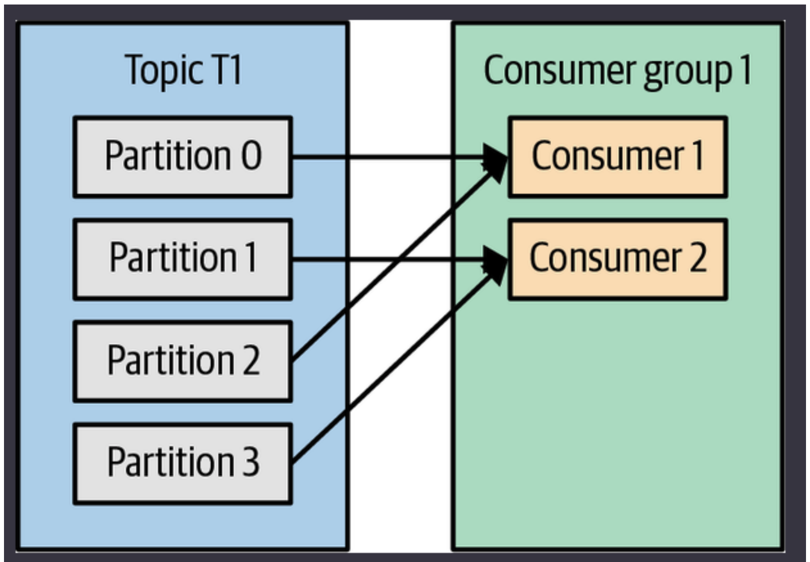


Figure 4-2. Four partitions split to two consumers in a group

Scenario 3: Each consumer will read a message from single partition.

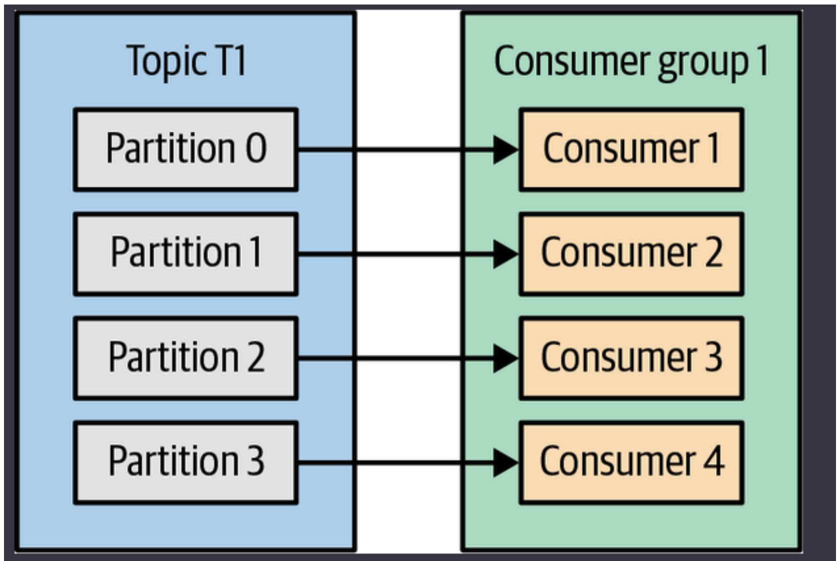


Figure 4-3. Four consumers in a group with one partition each

Scenario 4: If we have more consumers than partition, then the excessive consumer will be left to idle. Therefore, the rule of thumb, to scale the consumers to as many as partitions available.

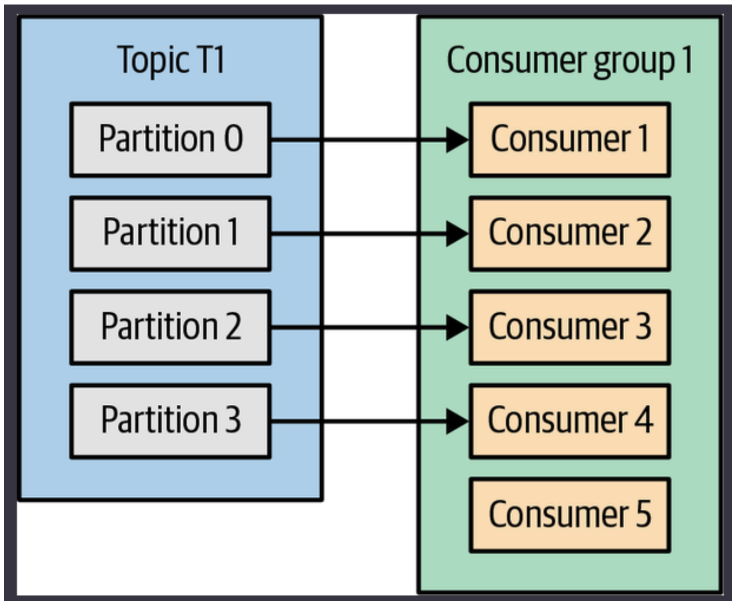


Figure 4-4. More consumers in a group than partitions means idle consumers

7.2 Consumer Group and Partition Rebalance

Moving partition ownership from one consumer to another is called rebalance. Rebalance helps in ensuring that the consumer group remains highly available and scalable.

There are two types of rebalance strategy:

- **Eager Rebalances**
  All consumer stop consuming, give up their ownership of all partition, rejon the consumer group, and get a brand new partition assignment.
  The tradeoff is that, there will be a short window of unavailability of the entire consumer group.

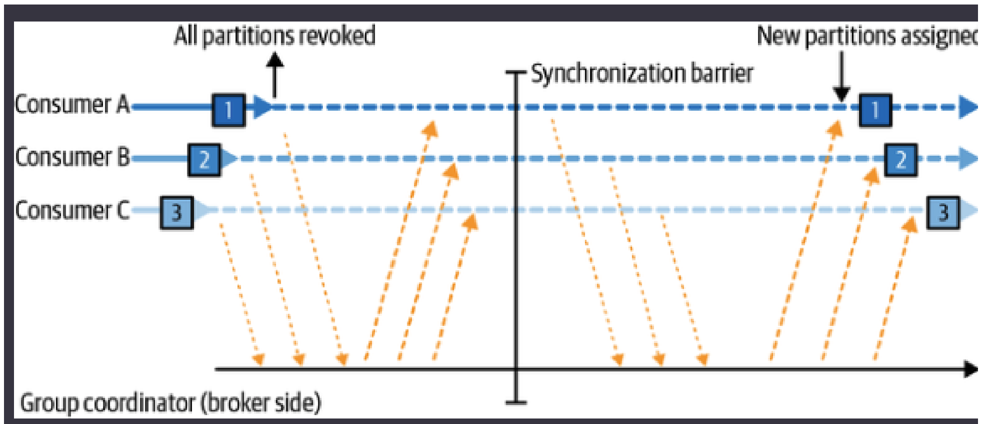


Figure 4-6. Eager rebalance revokes all partitions, pauses consumption, and reassigns them

- **Cooperate Rebalances or Incremental Rebalances**
  Cooperative Rebalances involve assigning only a small subset of the partitions from one consumer to another, and allowing consumers to continue processing records from all the partitions that are not reassigned.
  There are two phases to achieve this.
  - In the first phase, the consumer group leaderinforms all the consumers that they will lose ownership of a subset of their partition, then the consumers stop consuming from these partitions and give up their ownership in them.
  - In the second phase, the consumer group leader assigns these now orphaned partitions to their new owners.

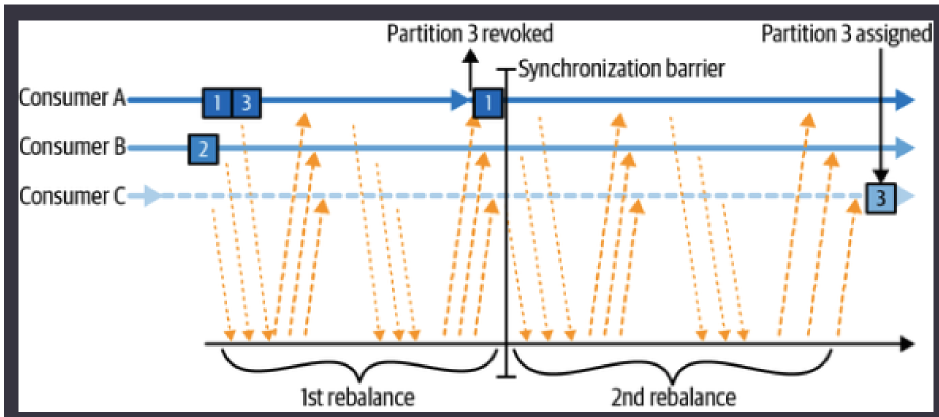


Figure 4-7. Cooperative rebalance only pauses consumption for the subset of partitions that will be reassigned

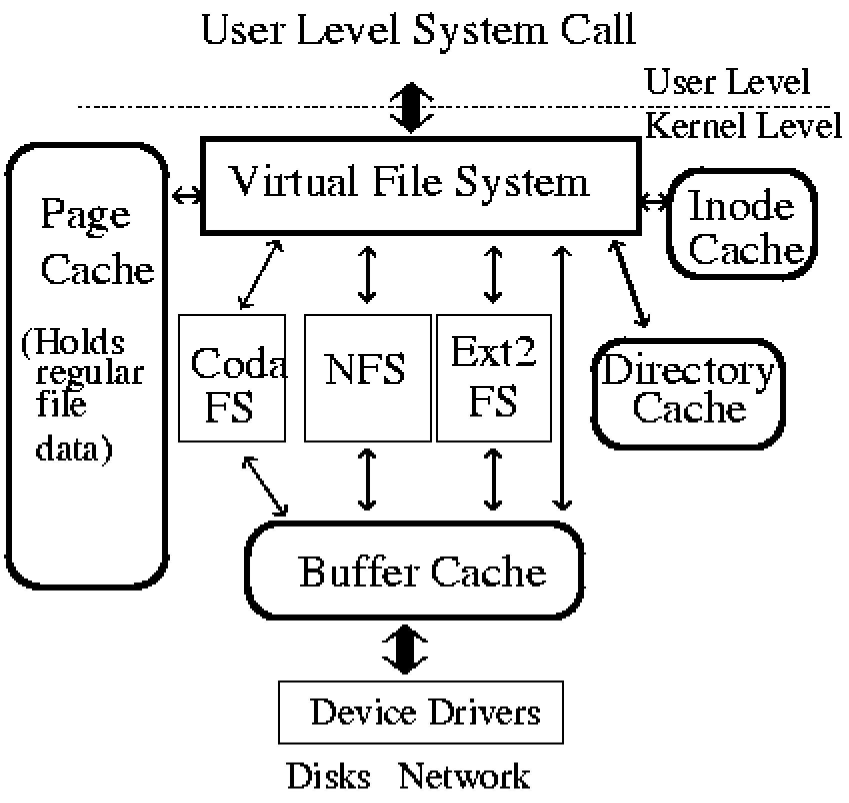
8 Kafka Design

8.1 Persistence via Filesystem

Kafka utilizes filesystem for storing and caching (page cache) messages. Contrary to typical beliefs, Kafka has optimally designed the disk structure to operate as fast the network.

A disk-based retention promotes durable message retention which promotes fault tolerance in consumers:

- If a consumer falls behind from the leader due to slow processing, there is not danger of losing data.
- The consumer can also be stopped or go offline, restart, and pick up the processing message where it left off with no data loss.



8.2 Exactly-Once Message Delivery Semantics

There are in total three types of semantics:

- At most once -- Messages may be lost but are never redelievered.
- At least once -- Messages are never lost but may be redelievered.
- Exactly once -- Each message is delivered once and only once.

Kafka uses the exactly-once semantics.

Kafka producers supports idempotent delivery, guaranteeing that resending will not result in duplicate entires in the log. This is achieved by assigning each producer an ID, and performs deduplication of messages.

Additionally, transaction-like semantics were applied to Kafka ever since version 0.11.0.0. Transaction-like semantics apply the Atomic principle, either all messages are successfully written or none.

Transaction-like semantics guarantee exactly-once processing. Exactly-once processing means that consuming, processing and producing are done atomically. Either the offset of the original message is committed and the result is successfully produced or neither of these things happen.

Using the example of stream processing. The consumer's position or offset is stored as a message in a topic, therefore, we can write the offset to Kafka in the same transaction as the output topic receiving the processed data. Therefore, if the transaction is aborted, the consumer's position will revert to its old value and the produced data on the output topic will not be visible to other consumers.

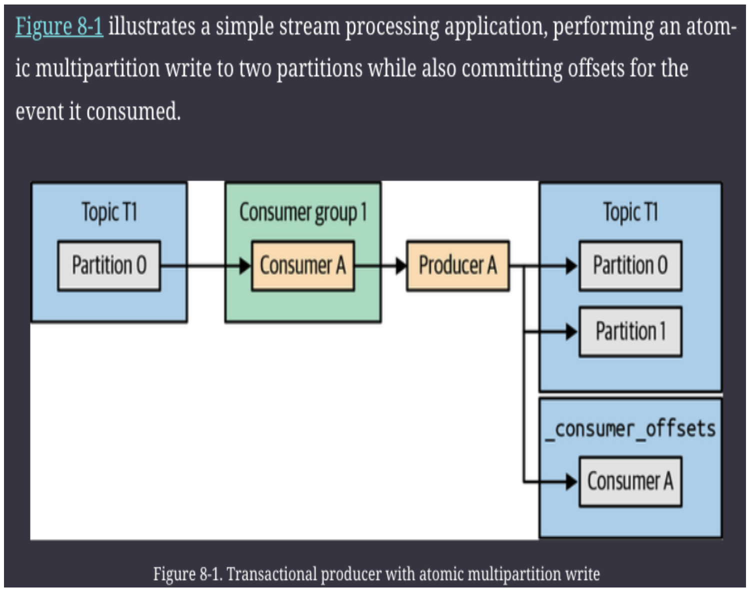


Figure 8-1 illustrates a simple stream processing application, performing an atomic multipartition write to two partitions while also committing offsets for the event it consumed.

Figure 8-1. Transactional producer with atomic multipartition write

8.3 Replication is the crux of Kafka's architecture

Replication is important to guarantee availability and ultimately, ensures fault-tolerance. Fault-tolerance guarantees automatic fail over when a server in the cluster fails.

To recap, Kafka is organized by topics. And in each topic, there can be many partitions. And each partition can have multiple replicas. Each partition has a single leader and zero or more followers.

In short, there are two types of replicas:

- Leader Replica
  Each partition has a single replica designated as the leader. All writes go to the leader to guarantee consistency. Consumer can consume from either the leader replica or follower replica.
- Follower Replica
  The main job of follower is to replicate messages from the leader and apply them to their own log, in short, to stay up-to-date with the most recent message the leader has. If the leader replica for a partition crashes, one of the follower replicas will be elected to be the leader.

As with every distributed system, there's a need to precisely define "liveness" of a broker.

A broker is "alive" only if it has fulfilled the two conditions:

- Broker must maintain an active session with the controller in order to receive regular metadata updates.
- Broker acting as follower must replicate the writes from leader and not fall "too far" behind.

Nodes that fulfil the two conditions of "alive" are rereferred to as "in-sync" replicas (ISR).

8.4 Quorum, a different approach via ISR

A Quorum approach is only applied, if you choose (1) the number of acknowledgements required and (2) the number of logs in order to elect a leader.

Kafka does not use a typical majority vote approach to elect a leader. The majority vote approach works because we have 2f + 1 replicas. If at least f+1 replicas have to receive an acknowledgement before declaring a leader, then we can only tolerate no more than f failures.

The downside of the majority vote concept is that it is too expensive.

As such, Kafka, takes a different approach in choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader, and the ISR set is persisted in the cluster metadata whenever it changes.

The maintenance of ISR set has several advantages:

- Can commit without the slowest server, while tolerating at most f failures.
- Does not need the crashed nodes to recover with all their data intact,
  The algorithm is applied in a manner that only allows replicas to re-join the ISR set only if the rebooted replica is fully re-synced with the leader replica.

8.5 What if all leaders die?

To handle this situation, one will have to decide between availability or consistency.

- Availability
  Select the first successful re-booted replica as the leader.
- Consistency
  Wait for all replicas to join the ISR set first before selecting the leader.

8.6 Log Compaction

Log compaction ensures that Kafka will always retain at least the last known value of each message key within the log of data for a single topic partition.

[More on how log compaction works in Kafka will be explained in part 2]

References

1. https://raft.github.io/ [Raft Consensus Algorithm]
2. Neha Narkhede, et al. Kafka: The Definitive Guide. “O’Reilly Media, Inc.,” 31 Aug. 2017.
3. https://kafka.apache.org/documentation/
4. Building Scalable Data Pipelines with Kafka from Educative.io