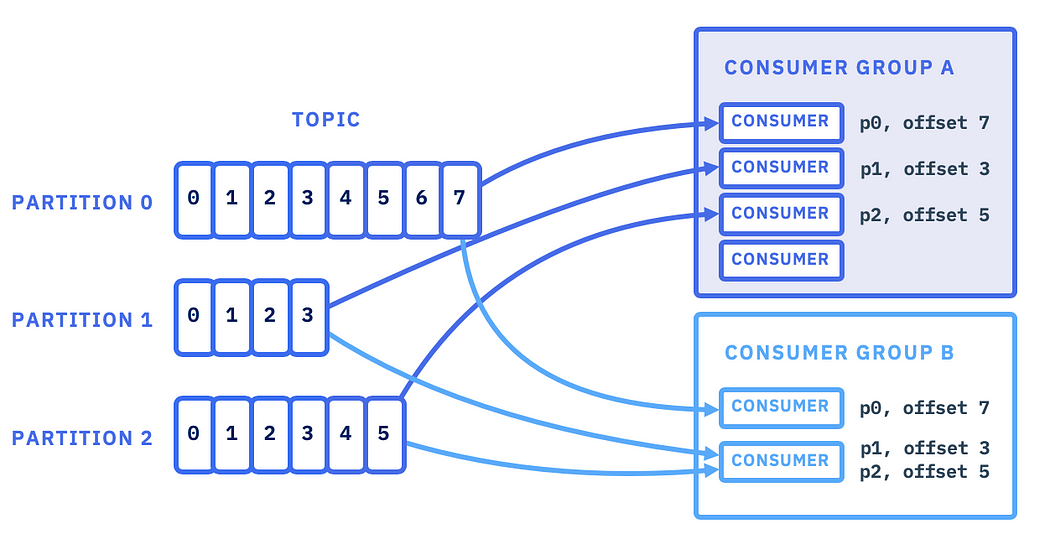
Another important concept for this command is the use of groups by the ***group*** parameter. In many cases, you might divide partitions amongst multiple **Consumers**.



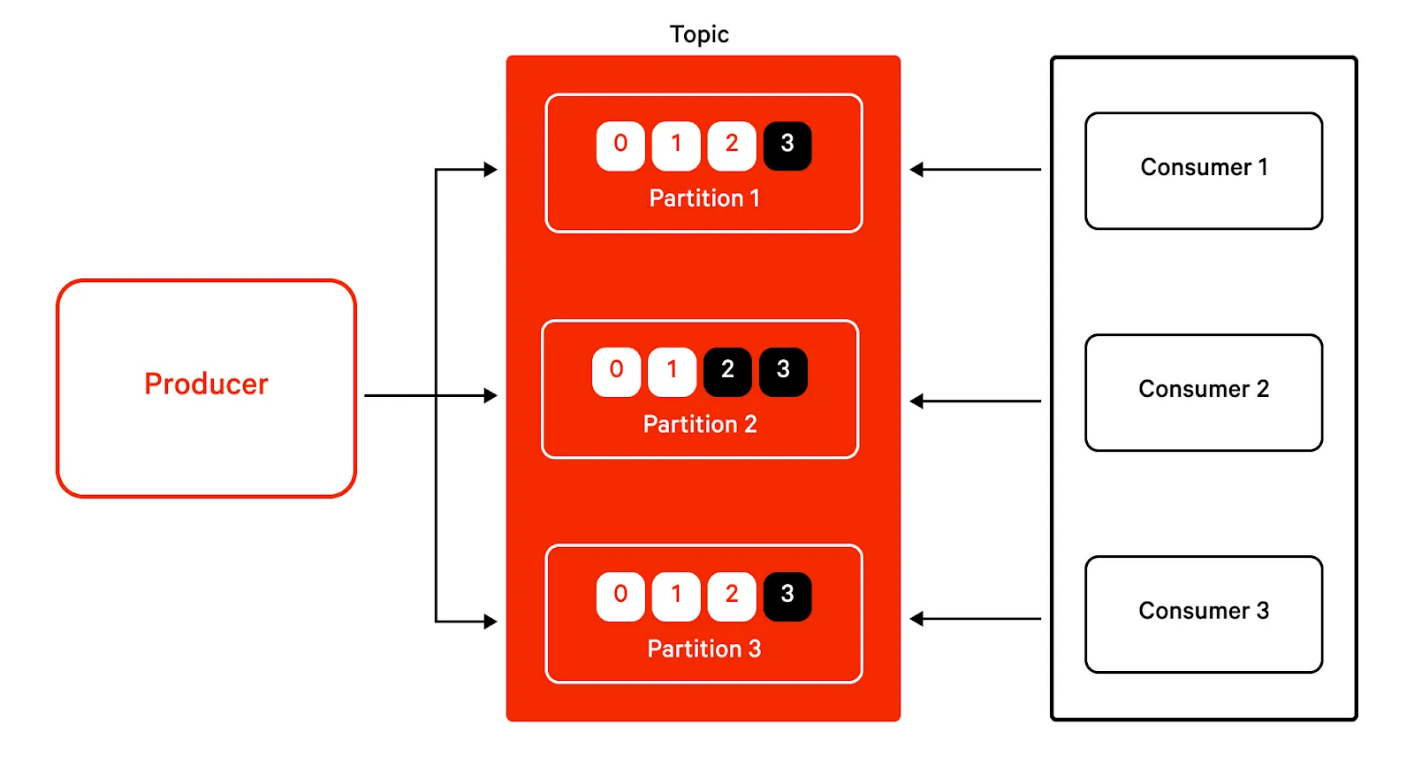
Kafka can be used as a message broker, a publish-subscribe mechanism, or a stream processing platform. A message broker sits between applications that interact using different protocols and helps in implementing decoupled solutions. A publish-subscribe mechanism helps applications broadcast messages to multiple target applications at the same time asynchronously. Kafka’s stream processing features enable developers to process high-velocity data in an orderly fashion and generate real-time insights.

Kafka is a distributed platform that works based on servers and clients. Organizations can use bare metal hardware, virtual machines, containers, on-premises instances, or cloud instances to deploy Kafka. A Kafka server is a cluster of nodes that can span multiple geographies or data centers. Some of the nodes work as a storage layer, and others work as Kafka Connect instances, which are responsible for managing connections from client applications.

Kafka’s working model contains four key elements:

* Messages are events that get dropped into Kafka for processing.
* Topics are logical groups of messages that signify a specific purpose.
* Producers are client applications that create messages.
* Consumers are client applications that listen to messages in topics and act according to the messages.

Kafka partition, offset, and consumer group architecture



Kafka partition strategy

Apache Kafka® groups related messages into topics, allowing consumers and producers to categorize messages. At a lower level, topics can be broken down into partitions, which is how a single topic can span across multiple brokers.

Kafka lets you choose how producers should publish messages to partitions and how partitions are assigned to consumers. However, there are multiple ways to route messages to different partitions. Each strategy comes with its own pros and cons, and learning which strategy is optimal for a certain circumstance differentiates Kafka experts from newbies.

In this article, we’ll show you the major strategies for allocating a topic’s messages to partitions. Then, we’ll give you an in-depth understanding of the theory behind how these strategies work using simple language and diagrams. Finally, we’ll walk you through different strategies, using real code samples to help you understand the practical implications of each approach.

Brief summary of partition strategies

Partitions increase parallelization and allow Kafka to scale. As we mentioned before, many strategies exist for distributing messages to a topic’s partitions. Before we dive deeper into the background of each strategy, the table below gives a brief overview of each strategy.

A Kafka message is sent by a producer and received by consumers. The strategies differ between the two, so we have two tables below, one summarizing each strategy.

Producer partition strategies

|  |  |
| --- | --- |
| **Strategy** | **Description** |
| Default partitioner | The key hash is used to map messages to partitions. Null key messages are sent to a partition in a round-robin fashion. |
| Round-robin partitioner | Messages are sent to partitions in a round-robin fashion. |
| Uniform sticky partitioner | Messages are sent to a sticky partition (until the batch.size is met or linger.ms time is up) to reduce latency. |
| Custom partitioner | This approach implements the Partitioner interface to override the partition method with some custom logic that defines the key-to-partition routing strategy. |

Consumer assignment strategies

|  |  |
| --- | --- |
| **Strategy** | **Description** |
| Range assignor (default) | (Total number of partitions) / (Number of consumers) partitions are assigned to each consumer. The aim is to have co-localized partitions, i.e., assigning the same partition number of two different topics to the same consumer (P0 of Topic X and P0 of Topic Y to the same consumer). |
|  |  |
| Round-robin assignor | Partitions are picked individually and assigned to consumers (in any rational order, say from first to last). When all the consumers are used up but some partitions still remain unassigned, they are assigned again, starting from the first consumer. The aim is to maximize the number of consumers used. |
| Sticky assignor | This approach works similar to round robin assignor but preserves as many existing assignments as possible when reassignment of partitions occurs. The aim is to reduce or completely avoid partition movement during rebalancing. |
| Custom assignor | Extends the AbstractPartitionAssignor class and overrides the assign method with custom logic. |

Producer partitioning strategies

Producers are applications that write data to partitions in Kafka topics. Kafka provides the following partitioning strategies when producing a message.

Default partitioner

As the name suggests, this is the default strategy for producing messages. When the key is null, the record is sent randomly to one of the available partitions of the topic. If a key exists, Kafka hashes the key, and the result is used to map the message to a specific partition. This ensures that messages with the same key end up in the same partition. This mapping, however, is consistent only as long as the number of partitions in the topic remains the same: If new partitions are added, new messages with the same key might get written to a different partition than old messages with the same key.

Round robin partitioner

Use this approach when the producer wants to distribute the writes equally among all partitions. This distribution is irrespective of the key’s hash value (or the key being null), so messages with the same key can end up in different partitions.

This strategy is useful when the workload becomes skewed by a single key, meaning that many messages are being produced for the same key. Suppose the ordering of messages is immaterial and the default partitioner is used. In that case, imbalanced load results in messages getting queued in partitions and an increased load on a subset of consumers to which those partitions are assigned. The round-robin strategy will result in an even distribution of messages across partitions.

Uniform sticky partitioner

Currently, when no partition and key are specified, a producer’s default partitioner partitions records in a round-robin fashion. That means that each record in a series of consecutive records will be sent to a different partition until all the partitions are covered, and then the producer starts over again. While this spreads records out evenly among the partitions, it also results in more batches that are smaller in size, leading to more requests and queuing as well as higher latency.

The uniform sticky partitioner was introduced to solve this problem. It has two rules:

* If a partition is specified with the record, that partition is used as it is.
* If no partition is specified, a sticky partition is chosen until the batch is full or linger.ms (the time to wait before sending messages) is up.

“Sticking” to a partition enables larger batches and reduces latency in the system. After sending a batch, the sticky partition changes. Over time, the records are spread out evenly among all the partitions.

The record key is not used as part of the partitioning strategy, so records with the same key are not guaranteed to be sent to the same partition.

Custom partitioner

Sometimes a use case does not fit well with any of the standard partitioners. For example, let’s suppose we want to write transaction log data to Kafka and one of the users (called “CEO”) accounts for more than 40% of all transactions.

If default hash partitioning is used, the “CEO” user’s records will be allocated to the same partition as other users. This would result in one partition being much larger than the rest, leading to brokkers running out of space and processing slowing down. An ideal solution is giving the user “CEO” a dedicated partition and then using hash partitioning to map the rest of the users to the remaining partitions.

