Chapter 3: Architecture and Core Concepts

The first two chapters have furnished a cushy debut of the essential concepts of event streaming and have given the reader an introduction to Apache Kafka as the premier event streaming technology that is increasingly used to power organisations of all shapes and sizes — from green-sprout startups to multi-national juggernauts.

Now that the scene has been set, it is time to take a deeper look at how Kafka works, and more to the point, how one works with it.

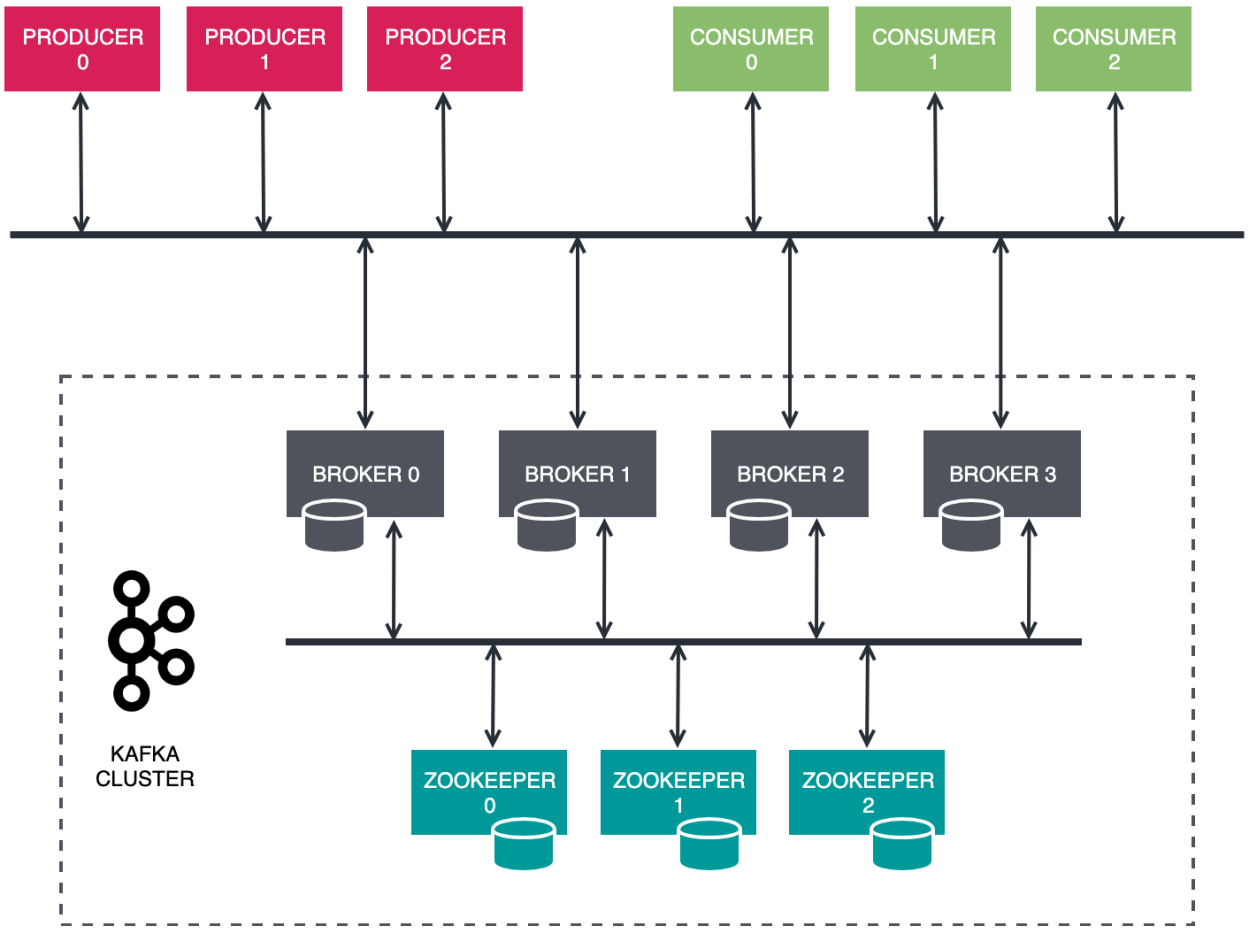
Architecture Overview

While the intention isn't to indoctrinate the reader with the minutia of Kafka's inner workings (for now), some appreciation of its design will go a long way in explaining the foundational concepts that will be covered shortly.

Kafka is a distributed system comprising several key components. At an outline level, these are:

- **Broker nodes**: Responsible for the bulk of I/O operations and durable persistence within the cluster.
- **ZooKeeper nodes**: Under the hood, Kafka needs a way of managing the overall controller status within the cluster. ZooKeeper fulfills this role, additionally acting as a consistent state repository that can be safely shared among the brokers.
- **Producers**: Client applications responsible for appending records to Kafka topics.
- **Consumers**: Client applications that read from topics.

The diagram below offers a brief overview of the Kafka component architecture, illustrating the relationships between its constituent parts. A further elaboration of the components follows.



Kafka Architecture Overview

Broker nodes

Before we begin, it is worth noting that industry literature uses the terminology 'Kafka Server', 'Kafka Broker' and 'Kafka Node' interchangeably to refer to the same concept. The official documentation refers to all three, while the shell scripts for starting Kafka and ZooKeeper refer to both as 'server'. This book favours the term 'broker' or, in some cases, the more elaborate 'broker node', avoiding the use of 'server' for its ambiguity.

So, what is a broker? If the reader comes from a background of messaging and middleware, the concept of a *broker* should resonate innately and intuitively. Otherwise, the reader is invited to consider Wikipedia's definition:

> *A broker is a person or firm who arranges transactions between a buyer and a seller for a commission when the deal is executed.*

Sans the commission piece, the definition fits Kafka like a glove. We would, of course, substitute 'buyer' and 'seller' for 'consumer' and 'producer', respectively, but one point is clear — the broker acts as an intermediary, facilitating the interactions between two parties — adding value in between.

This might make one wonder: *Why couldn't the parties interact directly?* A comprehensive answer would bore into the depths of computer science, specifically into the notion of *coupling*. We are not going to do this, to much relief; instead, the answer will be condensed to the following: the parties might not be aware of one another or they might not be jointly present at the same point in time. The latter places an additional demand on the broker: it must be stateful. In other words, it must persist the records emitted by the producer, so that they may be eventually delivered to the consumer when it is convenient to do so. The broker needs to be not only persistent, but also *durable*. By 'durable', it is implied that its persistence guarantees can be extended over a period of time and canvas scenarios that involve component failure.

Discussions of brokers as a means of decoupling communicating parties may conjure images of message queues from the days of yore. Many Kafka purists would protest: *Kafka is not a message queue, but an event streaming platform*. While the argument holds on the whole, the underpinning objectives remain largely unchanged. Fundamentally, we still have a publishing party, a subscribing party, and an intermediary to facilitate their interaction. And we would ideally like the parties to remain minimally coupled.

A Kafka broker is a Java process that acts as part of a larger cluster, where the minimum size of the cluster is one. (Indeed, we often use a singleton cluster for testing.) A broker is *one* of the units of scalability in Kafka; by increasing the number of brokers, one can achieve improved I/O, availability, and durability characteristics. (There are other ways of scaling Kafka, as we shall soon discover.)

A broker fulfills its persistence obligations by hosting a set of append-only log files that comprise the *partitions* hosted by the cluster. A more thorough discussion on partitions is yet to come; it will suffice to say for now that partitions are elemental units of storage that one can address in Kafka.

Each partition is mastered by exactly one broker — the partition *leader*. Partition data is replicated to a set of zero or more *follower* brokers. Collectively, the leader and the followers are referred to as *replicas*. Brokers share the load of leader and follower roles among themselves; a broker node may act as the leader for certain replicas, while being a follower for others. The roles may change — a follower replica may be promoted to leader status in the event of failure or as part of a manual rebalancing operation. The notion of replicas satisfies the durability guarantee; the more replicas in a cluster, the lower the likelihood of data loss due to an isolated replica failure.

Broker nodes are largely identical in every way; each node competes for the mastership of partition data on equal footing with its peers. Given the symmetric nature of the cluster, Kafka requires a mechanism for arbitrating the roles within the cluster and assigning partition leadership statuses among the broker nodes. Rather than making these decisions collectively, broker nodes follow a rudimentary chain-of-command. A single node is elected as the *cluster controller* which, in turn, directs all nodes (including itself) to assume specific roles. In other words, it is the controller's responsibility for managing the states of partitions and replicas, and for performing administrative tasks like reassigning partitions among the broker nodes.

ZooKeeper nodes

While the controller is entrusted with key administrative operations, the responsibility for electing a controller lies with another party — ZooKeeper. In fact, ZooKeeper is itself a cluster of cooperating processes called an *ensemble*. Every broker node will register its intent with ZooKeeper, but only one will be elected as the controller. ZooKeeper ensures that at most one broker node will be assigned the controller status, and should the controller node fail or leave the cluster, another broker node will promptly take its place.

A ZooKeeper ensemble also acts as a consistent and highly available configuration repository of sorts, maintaining cluster metadata, leader-follower states, quotas, user information, access control lists, and other housekeeping items. Owing to the underlying gossiping and consensus protocol of the ZooKeeper ensemble, *the number of ZooKeeper nodes must be odd.*

While ZooKeeper is bundled with Kafka for convenience, it is important to acknowledge that ZooKeeper is not an internal component of Kafka, but an open-source project in its own right.

Producers

A Kafka producer is a client application that can act as a source of data in a Kafka cluster. A producer communicates with the cluster over a set of persistent TCP connections, with an individual connection established with each broker. Producers can publish records to one or more Kafka topics, and any number of producers can append records to the same topic. Generally speaking, only producers are allowed to append records to topics; a consumer cannot modify a topic in any way.

Consumers

A consumer is a client application that acts as a data sink, subscribing to streams of records from one or more topics. Consumers are conceptually more complex than producers — they have to coordinate among themselves to balance the load of consuming records and track their progress through the stream.

Total and partial order

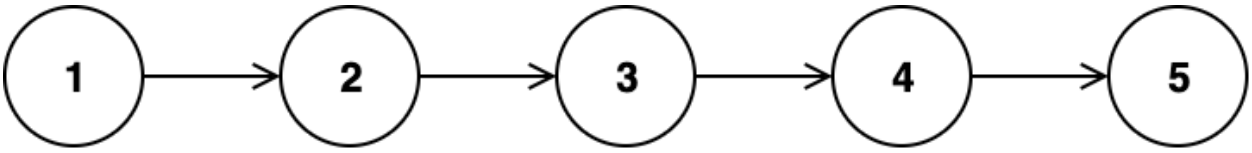
A minor note before we proceed: there is some theory ahead that one must endure to become an effective purveyor of Kafka. The upcoming content may seem esoteric and somewhat detached from the subject matter at first; however, the reader is assured that it is most relevant. We will be as brief as possible.

Without exaggeration, Kafka's entire event processing architecture is largely underpinned by the two primordial attributes of set theory: *partial order* and *total order*.

On the topic of set theory, what is a *set*? *A set is a collection of distinct elements* — objects that exist in their own right. For example, the numbers *2*, *4*, and *6* are distinct objects; when they are considered

collectively, they form a set of size three, written *{2, 4, 6}*. Developed at the end of the 19th century, set theory is now a ubiquitous part of mathematics; it is also generally considered fundamental to the construction of distributed and concurrent systems.

A *totally ordered* set is one where every element has a well-defined ordering relationship with every other element in the set. Consider, for example, the range of natural numbers one to five. When sorted in increasing order, it forms the following sequence:



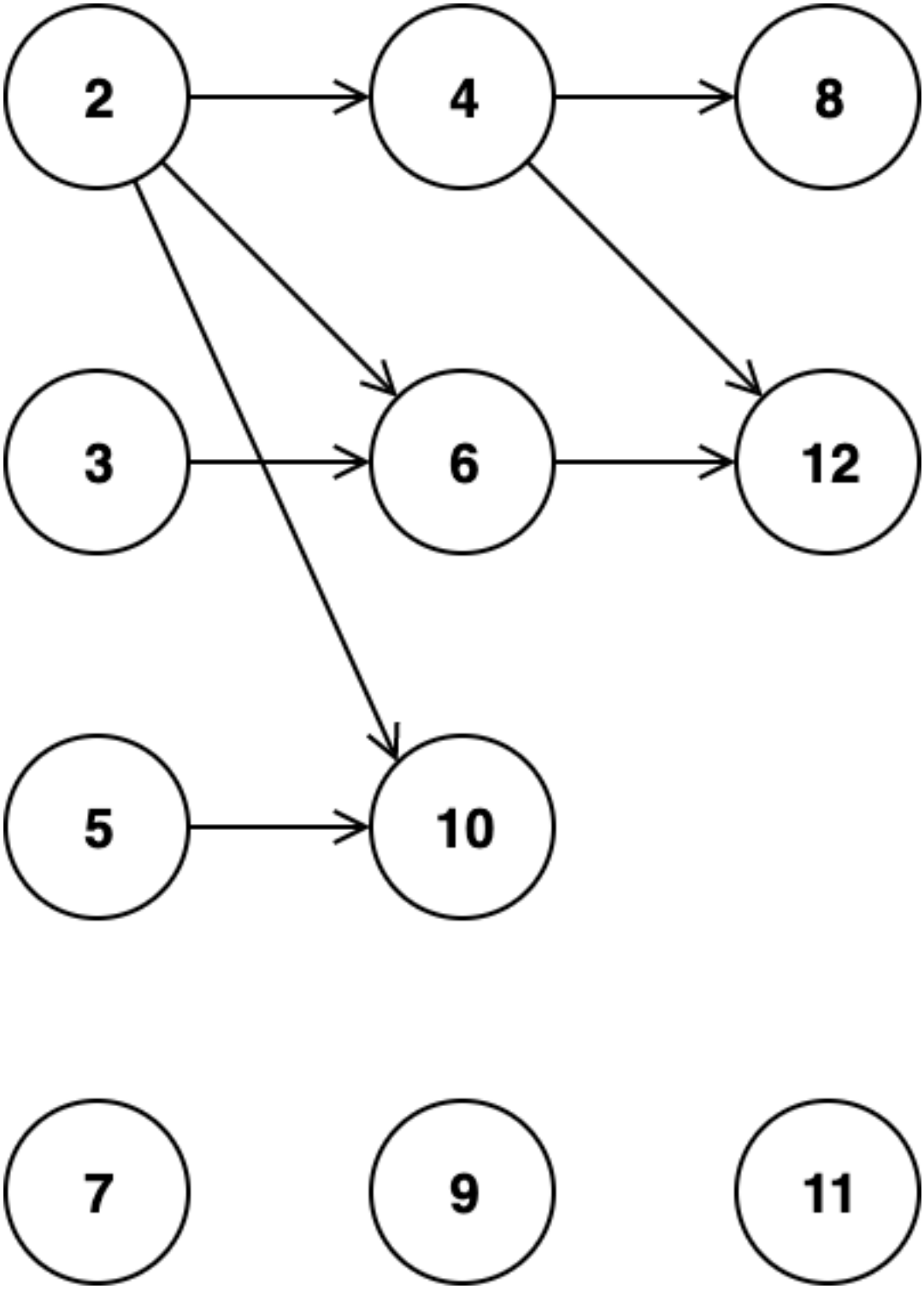
Totally ordered set

This is an ordered set, as every element has a well-defined predecessor-successor relationship with every other element. One can remove arbitrary values from this set and reinsert those values back into the set and arrive at the same sequence, no matter how many times this is attempted. Stated otherwise, there is only one permutation of elements that satisfies the ordering constraints.

Ordered sets exhibit the convenient property of *transitivity*. From the above example, we know that *2* must come after *1* and before *3*. We also know that *3* must come before *4*. Therefore, we can use the transitivity relation to deduce that *4* must come after *2*.

The direct antithesis of a totally ordered set is an unordered set. For example, an offhand list of capital cities *{Sydney, New York, London}* is unordered. Without applying further constraints, one cannot reason whether *Sydney* should appear before or after *New York*. One can arbitrarily permute the elements to arrive at different sequences of cities without upsetting anyone.

Between the two extremes, we find a partially ordered set. Consider the set of natural numbers ordered by divisibility, such that a number must appear after its divisor. For the range of numbers two to twelve, one instantiation of a sequence that satisfies this partial ordering constraint might be *[2, 3, 5, 7, 11, 4, 6, 9, 10, 8, 12]*. But that is just one instance — there are several such sequences that are distinct, yet equivalent. Looking at the set, we can state that the numbers *4* and *6* must appear after *2*, but there is no predecessor-successor relationship between *4* and *6* — they are mutually incomparable.



Partially ordered set

Multiple totally ordered sets can be contained in a single partially ordered set. For example, consider the Latin and Cyrillic alphabet sets *{A, B, C, …, Z}* and *{А, Б, В, …, Я}*, with their elements (letters)

arranged in alphabetical order — forming two distinct, totally ordered sets. Their union would be a partially ordered set; it would still maintain the relative order *within* each alphabet, without imposing order *across* alphabets.

Like in a totally ordered set, the elements of partially ordered sets exhibit transitivity. In the example involving divisors, the number *2*, appearing before *4*, where *4* appears before *8* or *12*, implies that both *8* and *12* must appear after *2*.

Topping off this discussion is the term 'causal order'. This type of ordering was first brought up in Chapter 1: Event Streaming Fundamentals, as part of the discussion on the consistency of replicated state. Unlike the other flavours, causal order is not a carryover from 19th-century mathematics; it stems from the study of distributed systems. A notable challenge of constructing such systems is that messages sent between processes may arrive zero or more times at any point after they are sent. As a consequence, there is no agreeable notion of time among collaborating processes. If one process, such as a clock, sends a timestamped message to another process, there is no reliable way for a receiver to determine the time relative to the clock and synchronise the two processes. This is often cited as the sole reason that building distributed systems is hard.

In the absence of a global clock, the relative timing of a pair of events occurring in close succession may be indistinguishable to an outside observer; however, if the two events are causally related, it is possible to distinguish their order; in other words, they become comparable. Causal order is a semantic rendition of partial order, where two elements may be bound by a *happened-before* relationship. This is denoted by an arrow appearing between the two elements; for example, if $A \longrightarrow B$, then A is an event that must logically precede B. This further implies that A occurred before B in a chronological sense. Otherwise, if $not(A \longrightarrow B)$, A cannot have preceded B in a causal sense. The latter does not imply that A could not have physically occurred before B; one simply has no way of ascertaining this.

> Causal relationships in distributed systems do not necessarily correspond to the more prevalent deductive 'cause-and-effect' style of logical reasoning. A causal relationship between a pair of events simply implies that one event precedes, rather than induces, the other. And it may be that the original events themselves are *not* comparable, but the recorded observations of these events are. These observations are events in their own right, and may also exhibit causality.
>
> Consider, for example, two samples of temperature readings *R0* and *R1* taken at different sites. They are communicated to a remote receiver and recorded in the order of arrival, forming a causal relationship on the receiver. If the message from *R0* was received first, we could confidently state that $received(R0) \longrightarrow received(R1)$. This does not imply that $sent(R0) \longrightarrow sent(R1)$, and it most certainly does not imply that *R0* played any part in inducing *R1*.

In considering the connection between the terms 'ordered', 'unordered', 'partially ordered', 'causally ordered', and 'totally ordered', one can draw the following synopsis:

- A partially ordered set implies that not every pair of elements needs to be comparable.

- A totally ordered set is a special case of a partially ordered set, where there exists a well-defined order between every conceivable element pair.
- An unordered set is also a special case of a partially ordered set, where there is no pair of comparable elements.
- Causal order is a rendition of partial order, where each element represents an event, and some pairs of events have a happened-before relationship.
- On its own, the term 'ordered set' is ambiguous, suggesting that the elements of a set exhibit some order-inducing relationships. This term is generally avoided.

With partial and total order out of the way, we can proceed to a discussion of records, topics, and partitions. The link between the latter and set theory will shortly become apparent.

Records

A *record* is the most elemental unit of persistence in Kafka. In the context of event-driven architecture, which is chiefly how one is meant to use Kafka, a record typically corresponds to some event of interest. It is characterised by the following attributes:

- **Key**: A record can be associated with an optional non-unique key, which acts as a kind of classifier — grouping related records on the basis of their key. The key is entirely free-form; anything that can be represented as an array of bytes can serve as a record key.
- **Value**: A value is effectively the informational payload of a record. The value is the most interesting part of a record in a business sense — it is the record's value that ultimately describes the event. A value is optional, although it is rare to see a record with a `null` value. Without a value, a record is largely pointless; all other attributes play a supporting role in conveying the value.
- **Headers**: A set of free-form key-value pairs that can optionally annotate a record. Headers in Kafka are akin to their namesake in HTTP — they augment the main payload with additional metadata.
- **Partition number**: A zero-based index of the partition that the record appears in. A record must always be tied to exactly one partition; however, the partition need not be specified explicitly when the record is published.
- **Offset**: A 64-bit signed integer for locating a record within its encompassing partition. Records are stored sequentially; the offset represents a logical sequence number of the record.
- **Timestamp**: A millisecond-precise timestamp of the record. A timestamp may be set explicitly by the producer to an arbitrary value, or it may be automatically assigned by the broker when a record is appended to the log.

Newcomers to Kafka usually have no problems grasping the concept of a record and understanding its internals, with the possible exception of the *key* attribute. Because Kafka is often likened to a database (albeit one for storing events), a record's key is often incorrectly associated with a database

key. This warrants prompt clarification, so as to not cause confusion down the track. Kafka does have a primary key, but it is *not* the record key. A record's equivalent of a 'primary key' is the composition of the record's partition number and its offset. A record's *key* is not unique, and therefore cannot possibly serve as the primary key. Furthermore, Kafka does not have the concept of a secondary index, and so the record key cannot be used to isolate a set of matching records.

Instead, it is best to think of a key as a kind of a pigeonhole into which related records are placed. Records maintain an association with their key over their entire lifetime. One cannot alter the key, or any aspect of the record for that matter, once the record has been published. It is unfortunate that keys are named as they are; a *classifier* (or a synonym thereof) would have been more appropriate.

The reason why the term 'key' was chosen is likely due to its association with hashing. Kafka producers use keys to map records to partitions — an action that involves hashing of the key bytes and applying the modulo operator. This carries a close resemblance to how keys are hashed to yield a bucket in a hash table.

The correspondence between Kafka records and observed events may not be direct; for example, an event might spawn multiple Kafka records, typically emitted in close succession. Those records may, in turn, be processed by a staged event-driven pipeline — spawning additional records in the course of processing. An overview of the staged event-driven architecture (SEDA) pattern was presented in Chapter 2: Introducing Apache Kafka.

Kafka is often used as a communication medium between fine-grained application services or across entire application domains. In saying that, the employment of Kafka as an internal note-taking or ledgering mechanism within a bounded context, is a perfectly valid use case. In fact, both *Event Sourcing* and *CQRS* patterns have seen strong adoption within the confines of single a domain, as well as across domains.

The recorded event might not have a real-life equivalent, even indirectly or circumstantially; there is no assumption or implication that Kafka is used solely as a registry of events. This statement may ruffle a few feathers or spark an all-out *bellum sacrum*; after all, Kafka is an event streaming platform — if not for recording events, what could it possibly be used for? Well, it might be used to replace a more traditional message broker. Much to the despise of Kafka purists (or delight, depending on one's personal convictions), an increasingly-growing use for Kafka is to replace technologies such as RabbitMQ, ActiveMQ, AWS SQS and SNS, Google Cloud Pub/Sub, and so forth. Kafka renowned flexibility lets it comfortably deal with a broad range of messaging topologies and applications, some of which have little resemblance to classical event-driven architecture.

> The generally accepted relationship between the terms 'message' and 'event' is such that a message encompasses a general class of asynchronous communiqués, while an event is a semantic specialisation of a message that communicates that some action of significance has occurred. Events have one logical owner — the producer (or publisher); they are immutable; they can be subscribed to and unsubscribed from. The term 'event' is often contrasted with another term — 'command' — a specialised message encompassing a directive issued from one party to another, requesting it to

perform some action. The logical owner of a command is its sole recipient; it cannot be subscribed to or unsubscribed from.

Kafka documentation and client APIs mostly prefer the term 'record', where others might use 'message' or 'event'. Kafka literature occasionally uses 'message' as a substitute for 'record', but this has been generally discouraged within the community, for the angst of confusing Kafka with the more traditional message-oriented middleware. In this book, the term 'record' is preferred, particularly when working in the context of event streaming. The term 'event' will generally be used to refer to an external action that triggered the publishing of the record, but may also be metonymically used to refer to the record itself, as it is often convenient to do so. Finally, this book may occasionally use the term 'message' when describing records in the context of a more traditional message broker, where the use of this term aids clarity.

Partitions

A partition is a totally ordered, unbounded set of records. Published records are appended to the head-end of the encompassing partition. Where a record can be seen as an elemental unit of persistence, a partition is an elemental unit of record streaming.

Because records are totally ordered within their partition, any pair of records in the same partition is bound by a predecessor-successor relationship. This relationship is implicitly assigned by the producer application. For any given producer instance, records will be written in the order they were emitted by the application. By way of example, assume a pair of records *P* and *Q* destined for the same partition. If record *P* was published before *Q*, then *P* will precede *Q* in the partition. Furthermore, they will be read in the same order by *all* consumers; *P* will always be read before *Q*, for every possible consumer. This ordering guarantee is vital when implementing event-driven systems, more so than for peer-to-peer messaging or work queues; published records will generally correspond to or derive from real-life events, and preserving the timeline of these events is often essential.

Records published to one partition by the same producer are causally ordered. In other words, if *P* precedes *Q*, then *P* must have been observed before *Q* on the producer; the happened-before relationship is preserved — imparted from the producer onto the partition.

There is no recognised causal ordering *across* producers; if two (or more) producers emit records simultaneously for the same partition, those records may materialise in arbitrary order. The relative order will not depend on which producer application attempted to publish first, but rather, which record beat the other to the partition leader. That said, whatever the order, it will be *consistent* — observed uniformly across all consumers. Total order is still preserved, but some record pairs may only be related circumstantially.

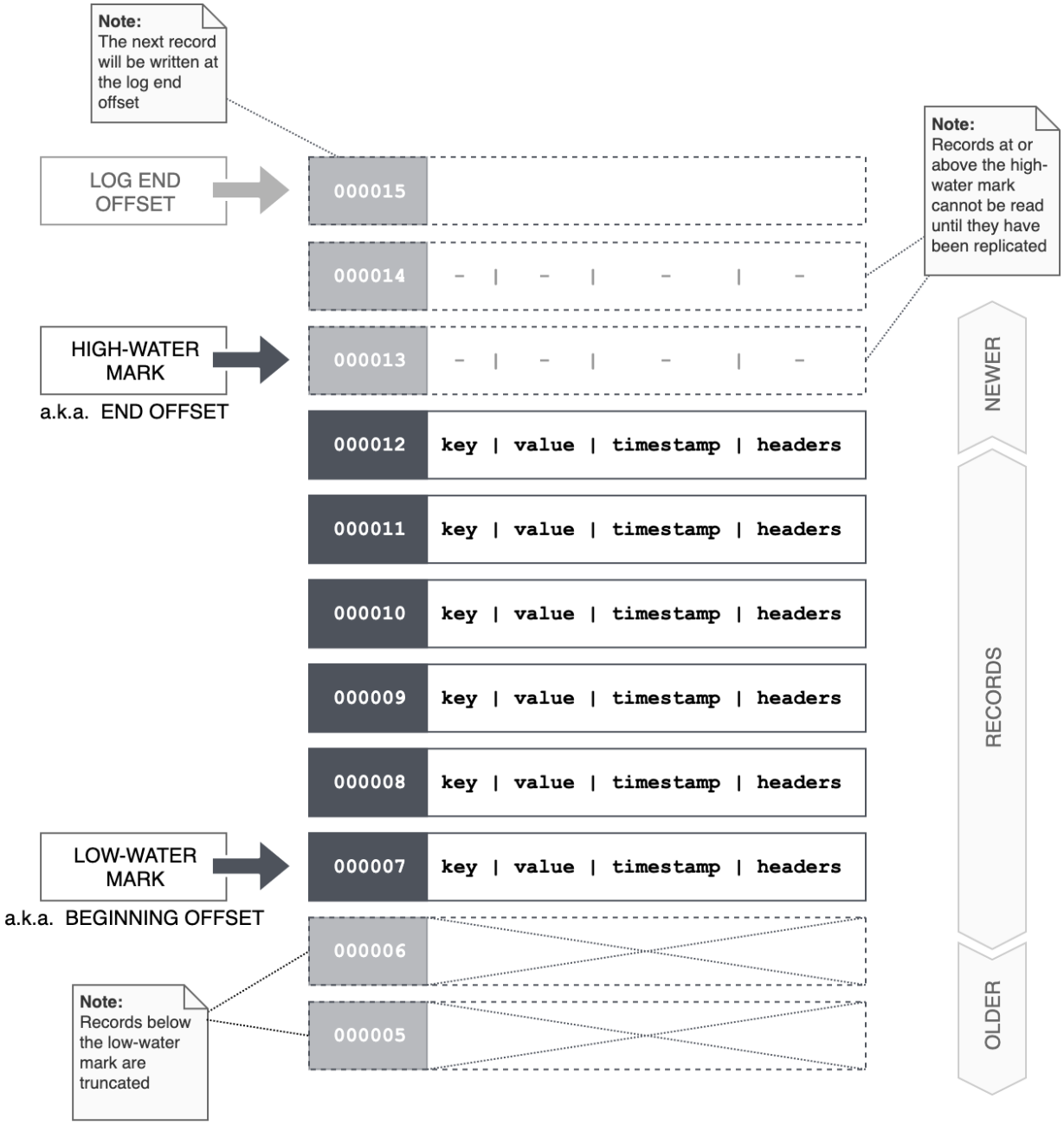
Corollary to the above, *in the absence of producer synchronisation, causal order can only be achieved when a single producer emits records to the same partition.* All other combinations — involving

multiple unrelated producers or different partitions — may result in a record stream that fails to depict causality. Whether or not this is an issue will depend largely on the application.

A record's offset uniquely identifies it in the partition. The offset acts as a primary key, allowing for fast, *O(1)* lookups. The offset is a strictly monotonically-increasing integer in a sparse address space, meaning that each successive offset is always higher than its predecessor, and there may be varying gaps between neighbouring offsets. Gaps might legitimately appear if compaction is enabled or as a result of transactions; we don't need to delve into the details at this stage, suffice it to say that offsets need not be contiguous.

Given a record, an application shouldn't attempt to literally interpret its offset or guess what the next offset might be. It may, however, actively exploit the properties of *total order* and *transitivity* to infer the relative order of any record pair based on their offsets, sort the records by their offset, and so forth.

The diagram below shows what a partition looks like on the inside.



Partition contents

The *beginning offset*, also called the *low-water mark*, is the first record that will be presented to a prospective consumer. Due to Kafka's bounded retention, this is not necessarily the first record that was published. Records may be pruned on the basis of time and/or partition size. When this occurs, the low-water mark will appear to advance, and records earlier than the low-water mark will be truncated.

Conversely, the *high-water mark* is the offset immediately following the last successfully replicated record. Consumers are only allowed to read up to the high-water mark. This prevents a consumer

from reading unreplicated data that may be lost in the event of leader failure. The equivalent term for a high-water mark is the *end offset*.

The statement above is a minor simplification. The end offset corresponds to the high-water mark for non-transactional consumers. Where a more strict isolation mode has been selected on the consumer, the end offset may trail the high-water mark. Transactional messaging is an advanced topic, covered in Chapter 18: Transactions.

The end offset should not be confused with the internal term 'log end offset', which is the offset immediately following that of the last written record. The 'log end offset' will be assigned to the next record that will be published. When the follower replicas lag behind the leader, the 'log end offset' will be greater than the high-water mark. When replication eventually catches up, the high-water mark will align with the 'log end offset'.

Subtracting the low-water mark from the high-water mark will yield the upper bound on the number of securely persisted records in the partition. The actual number may be slightly less, as the offsets are not guaranteed to be contiguous. The *total* number of records may be fewer or greater, as the high-water mark does not reflect the number of unreplicated records.

Topics

So, a partition is an unbounded sequence of records, an open ledger, a continuum of events — each definition as good as the next. Along with a record, a partition is an elemental building block of an event streaming platform. But a partition is too basic to be used effectively on its own.

A *topic* is a logical aggregation of partitions. It comprises one or more partitions, and a partition must be a part of exactly one topic. Topics are fundamental to Kafka, allowing for both parallelism and load balancing.

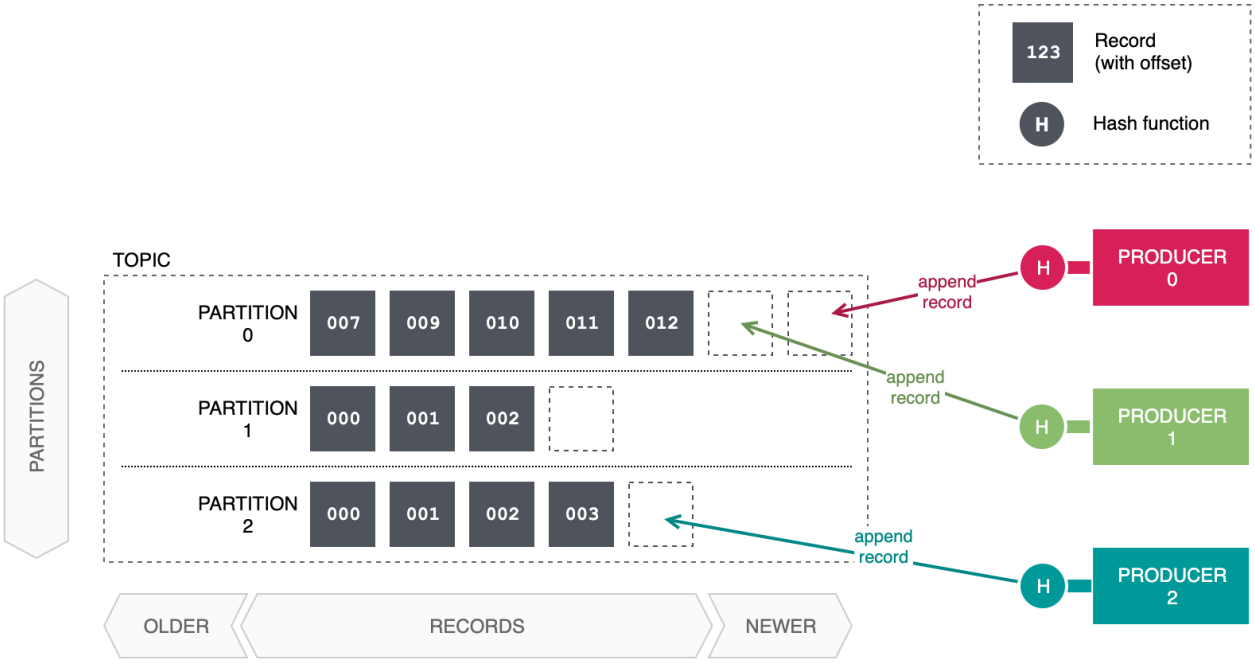
Earlier, it was said that partitions exhibit total order. Taking a set-theoretic perspective, a topic is just a union of the individual underlying sets; since partitions within a topic are mutually independent, the topic is said to exhibit *partial order*. In simple terms, this means that certain records may be ordered in relation to one another, while being unordered with respect to certain other records. A Kafka topic, and specifically its use of partial order, enables us to process records in parallel *where we can*, while maintaining order *where we must*. The concept of consumer parallelism will be explored shortly; for the time being, the focus will remain on the producer ecosystem.

Since Kafka is an event streaming platform, it may be more instructive to think of a topic and its partitions as a wide *stream*, comprising multiple parallel *substreams*. Events within a substream *may* be related objectively, insofar as one event must precede some other event. In other words, a causal relationship is in place. (The events are not required to be causally related to share a substream; the reason that will be touched on later.) Events across substreams are related subjectively — they might

refer to a similar class of observations and it may be advantageous to encompass them within the same stream.

Occasionally, this book will use the term 'stream' as a substitute for 'topic'; when referring to events, the use of the term 'stream' is often more natural and intuitive.

Precisely how records are partitioned is left to the discretion of the producer. A producer application may explicitly assign a partition number when publishing a record, although this approach is rarely used. A much more common approach is for the client application to deal exclusively with record keys and values, and have the producer library automatically select a partition on the basis of a record's key. A producer will digest the byte content of the key using a hash function (Kafka uses `murmur2` for this purpose). The highest-order bit of the hash value is masked off to force it to a positive integer, before taking the result, modulo the number of partitions, to arrive at the final partition number. The contents of the topic and the producers' interactions with the topic are depicted below.



Topic contents

While this partitioning scheme is deterministic, it is not *consistent*. Two records with the same key hashed at different points in time will correspond to an identical partition number *if and only if* the number of partitions has not changed in that time. Increasing the number of partitions in a topic (Kafka does not support non-destructive downsizing) results in the two records occupying potentially different partitions — leading to a breakdown of any prior order. There are many gotchas, such as this one, in Kafka; they will be called out as such from time to time.

Records sharing the same hash are guaranteed to occupy the same partition. Assuming a topic with

multiple partitions, records with a different key will likely end up in different partitions. However, due to hash collisions, records with different hashes may also end up in the same partition. Such is the nature of hashing; if the reader appreciates how a hash table works, this is no different. It was previously stated that records in the same partition may be causally related, but do not have to be. The reason is specifically to do with hashing; when there are more causally related record groupings then there are partitions in a topic, there will invariably be some partitions that contain multiple unrelated sets of records. In mathematics, this is referred to as the *Dirichlet's Drawer Principle* or the *Pigeonhole Principle*. In fact, due to the imperfect space distribution of hash functions, unrelated records will likely be grouped in the same partition even there are more partitions than distinct keys.

Producers rarely care which specific partition the records will map to, only that related records end up in the same partition, and that their order is preserved. Similarly, consumers are largely indifferent to their assigned partitions, so long that they receive the records in the same order as they were published, where those records are causally bound.

Consumer groups and load balancing

So far we have learned that producers emit records to a topic; these records are organised into neatly ordered partitions. Kafka's producer-topic-consumer topology adheres to a flexible and highly generalised *multipoint-to-multipoint* model, meaning that there may be any number of producers and consumers simultaneously interacting with a topic. Depending on the actual solution context, topologies may also be point-to-multipoint, multipoint-to-point, and point-to-point. Kafka does not impose the sorts of limits that one is used to seeing from the more 'orthodox' messaging middleware. It's about time we looked at how records are consumed.

A *consumer* is a process or thread that attaches to a Kafka cluster via a client library. A consumer generally, but not necessarily, operates as part of an encompassing *consumer group*. Consumer groups are effectively a *load-balancing* mechanism within Kafka — distributing partition assignments approximately evenly among the individual consumer instances within the group. When the first consumer in a group subscribes to the topic, it will receive all partitions in that topic. When a second consumer subsequently joins, it will get approximately half of the partitions, relieving the first consumer of half of its prior load. The process runs in reverse when consumers leave (by disconnecting or timing out) — the remaining consumers will absorb a greater number of partitions.

This book will occasionally use the term 'subscriber' to collectively refer to all consumer instances in a consumer group, as a single, logical entity. When referring to event streams, the notion of a subscriber is sometimes more intuitive and helps distinguish between those consumers who may not be a part of a consumer group at all. Those sorts of consumers will be discussed later.

So, a consumer siphons records from a topic, pulling from the share of partitions that have been assigned to it by Kafka, alongside the other consumers in its group. As far as load-balancing goes, this is nothing out of the ordinary. But here's the kicker — *consuming a record does not remove it from*

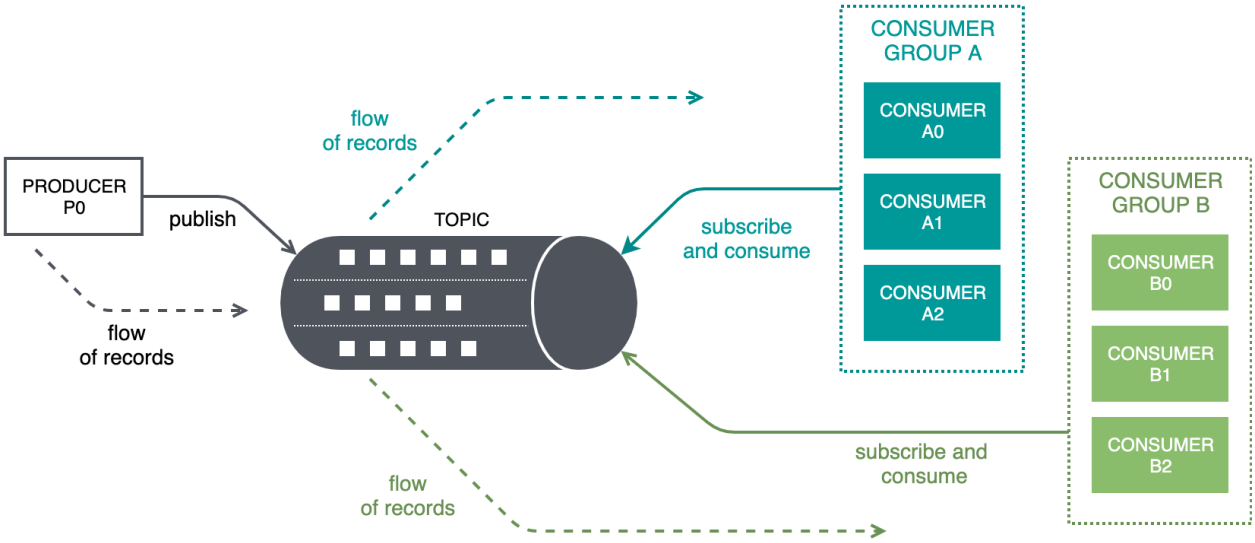
the topic. This might seem contradictory at first, especially if one associates the act of consuming with depletion. (If anything, a consumer should have been called a 'reader', but let's not dwell on the choice of terminology.) The simple fact is, consumers have absolutely no impact on the topic and its partitions; a topic is an append-only log that may only be mutated by the producer, or by Kafka itself as part of its housekeeping chores. Consumers are 'cheap', so to speak — you can have a fair number of them tail the logs without stressing the cluster. This is a yet another point of distinction between an event stream and a traditional message queue, and it's a crucial one.

A consumer internally maintains an offset that points to the next record in a partition, advancing the offset for every successive read. In fact, a consumer maintains a vector of such offsets — one for each assigned partition. When a consumer first subscribes to a topic, whereby no offsets have been registered for the encompassing consumer group, it may elect to start at either the head-end or the tail-end of the topic. Thereafter, the consumer will acquire an offset vector and will advance the offsets internally, in line with the consumption of records.

In Kafka terminology, the 'head' of a partition corresponds to the location of the end offsets, while the 'tail' of the partition is the side closest to the beginning offsets. This might sound confusing if Kafka is perceived as a queue of sorts, where the head-end of a queue canonically corresponds to the side which has the oldest elements. In Kafka, the oldest elements are at the tail-end.

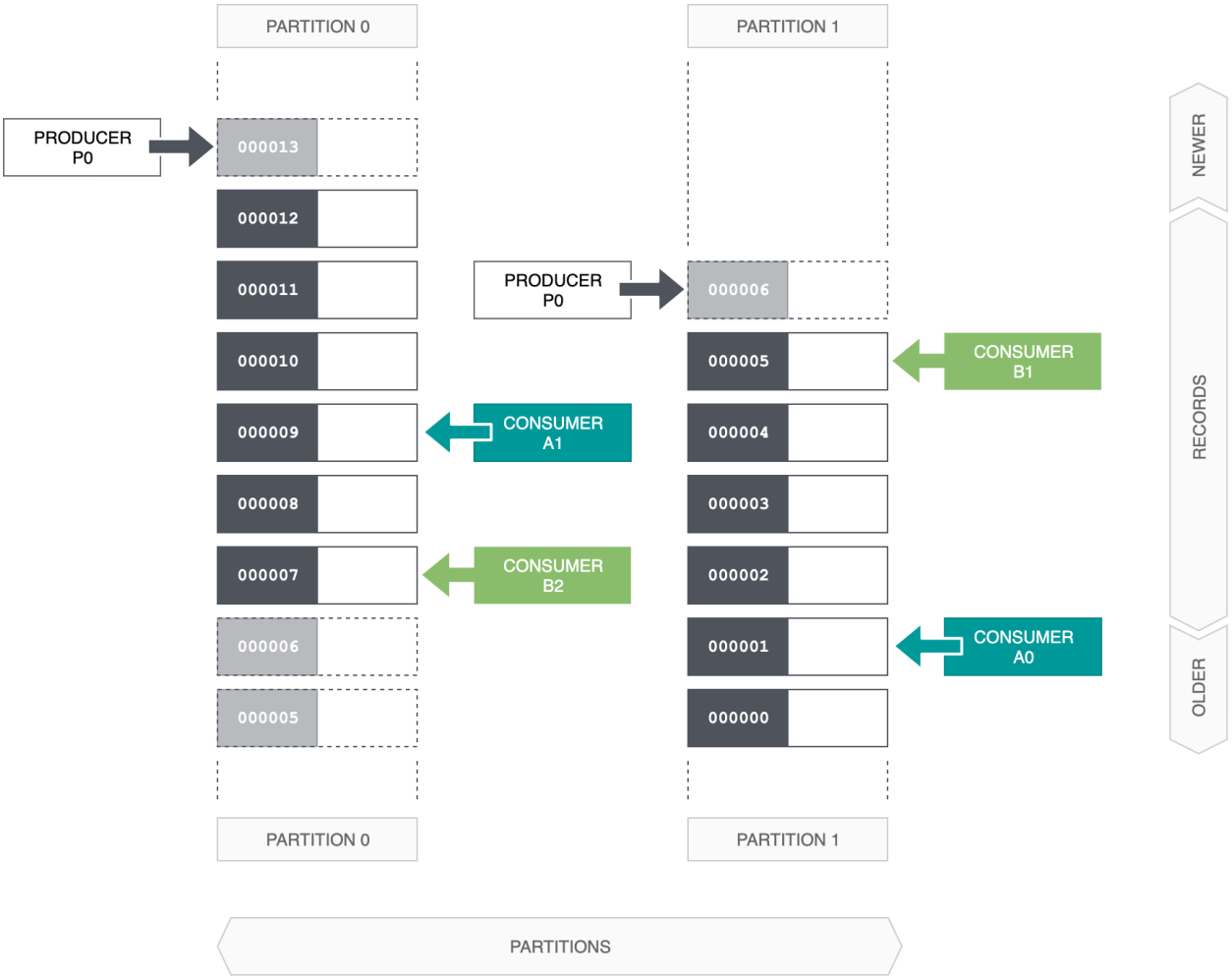
Since consumers across different consumer groups do not interfere, there may be any number of them reading concurrently from the same topic. Consumers run at their own pace; a slow or backlogged consumer has no impact on its peers.

To illustrate this concept, consider a contrived scenario involving a topic with two partitions. Two consumer groups — *A* and *B* — are subscribed to the topic. Each group has three consumer instances, named *A0*, *A1*, *A2*, *B0*, *B1*, and *B2*. The relationship between topics, consumers, and groups is illustrated below.



Multiple consumer groups sharing a topic

As part of fulfilling the subscriptions, Kafka will allocate partitions among the members of each group. In turn, each group will acquire and maintain a dedicated set of offsets that reflect the overall progress of the group through the topic. The sharing of the topic and the independent progress of consumer groups is diagrammatically depicted below.



Consumer offsets

Upon careful inspection, the reader will notice that something is missing. Two things, in fact: consumers *A2* and *B0* aren't there. That is because Kafka ensures that a partition may only be assigned to at most one consumer within its consumer group. (It is said 'at most' to cover the case when all consumers are offline.) Because there are three consumers in each group, but only two partitions, one consumer will remain idle — waiting for another consumer in its respective group to depart before being assigned a partition. In this manner, consumer groups are not only a load-balancing mechanism, but also a fence-like exclusion control, used to build highly performant pipelines without sacrificing *safety*, particularly when there is a requirement that a record may only be handled by one thread or process at any given time.

Consumer groups also ensure *availability*, satisfying the *liveness* property of a distributed consumer ecosystem. By periodically reading records from a topic, the consumer implicitly signals to the cluster that it is in a 'healthy' state, thereby extending the lease over its partition assignment. Should the consumer fail to read again within the allowable deadline, it will be deemed faulty and its partitions will be reassigned — apportioned among the remaining 'healthy' consumers within its group.

A thorough discussion of the *safety* and *liveness* properties of Kafka will be deferred until Chapter 15: Group Membership and Partition Assignment. For the time being, the reader is asked to accept an abridged definition: Liveness is a property that requires a system to eventually make progress, completing all assigned work. Safety is a property that requires the system to respect all its key invariants, at all times.

To employ a transportation analogy, a topic is like a highway, while a partition is a lane. A record is the equivalent of a car, and its occupants correspond to the record's value. Several cars can safely travel on the same highway, providing they keep to their lane. Cars sharing the same line ride in a sequence, forming an orderly queue. Now suppose each lane leads to an off-ramp, diverting its traffic to some location. If one off-ramp gets banked up, other off-ramps may still flow smoothly.

It is precisely this highway-lane metaphor that Kafka exploits to achieve its trademark end-to-end throughput, easily reaching millions of records per second on commodity hardware. When creating a topic, one can set the partition count — the number of lanes, if you will. The partitions are divided approximately evenly among the individual consumers in a consumer group, with a guarantee that no partition will be assigned to two (or more) consumers at the same time, providing that these consumers are part of the *same consumer group*. Referring to our analogy, a car will never end up in two off-ramps simultaneously; however, two lanes might conceivably merge to the same off-ramp.

The High Five Interchange, Dallas, Texas

Committing offsets

It has already been said that consumers maintain an internal state with respect to their partition offsets. At some point, that state must be shared with Kafka, so that when a partition is reassigned, the new consumer can resume processing from where the outgoing consumer left off. Similarly, if the consumers were to disconnect, upon reconnection they would ideally skip over any records that have already been processed.

Persisting the consumer state back to the Kafka cluster is called *committing* an offset. Typically, a consumer will read a record (or a batch of records) and commit the offset of the last record *plus one*. If a new consumer takes over the topic, it will commence processing from the last committed offset — hence the plus-one step is essential. (Otherwise, the last processed record would be handled a second time.)

Curious fact: Kafka employs a recursive approach to managing committed offsets, elegantly utilising itself to persist and track offsets. When an offset is committed, the group coordinator will publish a binary record on the internal `__consumer_offsets` topic. The contents of this topic are compacted in the background, creating an efficient event store that progressively reduces to only the last known commit points for any given consumer group.

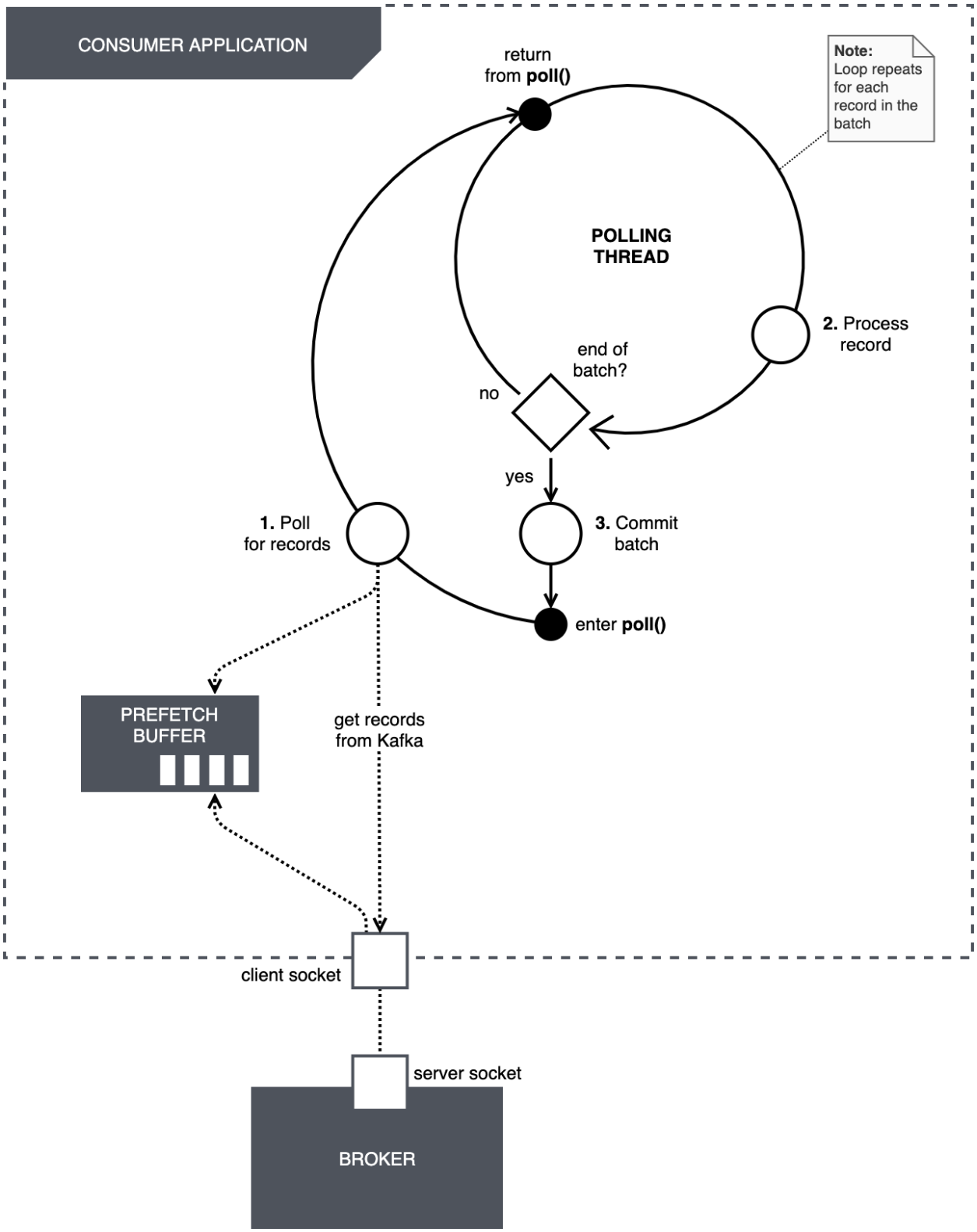
Controlling the point when an offset is committed provides a great deal of flexibility around *delivery guarantees*, further highlighting Kafka's adaptability towards various messaging scenarios. The term 'delivery' assumes not just reading a record, but the full processing cycle, complete with any side-effects. (For example, updating a database, or invoking a service.) One can shift from an *at-most-once* to an *at-least-once* delivery model by simply moving the commit operation from a point *before* the processing of a record is commenced, to a point sometime *after* the processing is complete. With this model, should the consumer fail midway through processing a record, it will be re-read following partition reassignment.

By default, a Kafka consumer will automatically commit offsets at an interval of *at least* every five seconds. The interval will automatically be extended in the presence of in-flight records — the records that are still being processed on the consumer. The lower bound on this interval can be controlled by the `auto.commit.interval.ms` configuration property, which is discussed in Chapter 10: Client Configuration. An implication of the offset auto-commit feature is that it extends the window of uncommitted offsets beyond the set of in-flight records; in other words, the consumer might finish processing a batch of records without necessarily committing the offsets. If the consumer's partitions are then reassigned, the new consumer will end up processing the same batch a second time. To constrain the window of uncommitted records, one needs to take offset committing into their own hands. This can be done by setting the `enable.auto.commit` client property to `false`.

Getting offset commits right can be tricky, and routinely catches out beginners. A committed offset implies that the record *one below that offset and all prior records have been dealt with by the consumer*. When designing at-least-once applications, an offset should only be committed when the application has dealt with the record in question, and all records before it. In other words, the record

has been processed to the point that any actions that would have resulted from the record have been carried out and finalised. This may include calling other APIs, updating a database, committing transactions, persisting the record's payload, or publishing more records. Stated otherwise, if the consumer were to fail after committing the record, then not ever seeing this record again must not be detrimental to its correctness.

In the at-least-once scenario, a typical consumer implementation will commit its offsets linearly, in tandem with the processing of a record batch. That is, read a record batch from a topic, process the individual records, commit the outstanding offsets, read the next batch, and so on. This is called a *poll-process* loop, illustrated below:



Consumer poll-process loop

The poll-process loop in the above diagram is a somewhat simplified take on reality. We will not go into the details of how records are fetched from Kafka when `KafkaConsumer.poll()` is called; a more thorough description is presented in Chapter 7: Serialization. We will remark on one optimisation: a consumer does not always fetch records directly from the cluster; it employs a prefetch buffer to pipeline this process.

A common tactic is to process a batch of records concurrently (where this makes sense), using a thread pool, and only confirm the last record when the entire batch is done. The commit process in Kafka is very efficient, the client library will send commit requests asynchronously to the cluster using an in-memory queue, without blocking the consumer. The client application can register an optional callback, notifying it when the commit has been acknowledged by the cluster. And there is also a blocking variant available should the client application prefer it.

Free consumers

The association of a consumer with a consumer group is an optional one, indicated by the presence of a `group.id` consumer property. If unset, a *free consumer* is presumed. Free consumers do not subscribe to a topic; instead, the consuming application is responsible for manually assigning a set of topic-partitions to itself, individually specifying the starting offset for each topic-partition pair. *Free consumers do not commit their offsets to Kafka*; it is up to the application to track the progress of such consumers and persist their state as appropriate, using a datastore of their choosing. The concepts of automatic partition assignment, rebalancing, offset persistence, partition exclusivity, consumer heartbeating and failure detection (safety and liveness, in other words), and other so-called 'niceties' accorded to consumer groups cease to exist in this mode.

The use of the nominal expression *'free consumer'* to denote a consumer without an encompassing group is a coined term. It is not part of the standard Kafka nomenclature; indeed, there is no widespread terminology that marks this form of consumer.

Free consumers are not observed in the wild as often as their grouped counterparts. There are predominantly two use cases where a free consumer is an appropriate choice. One such case is when an application genuinely requires full control of the partition assignment scheme, likely utilising a dedicated datastore to track consumer offsets. This is *very* rare. Needless to say, it is also difficult to implement correctly, given the multitude of scenarios one must account for. It is mentioned here only for completeness.

The more commonly seen use case is when a stateless or ephemeral consumer needs to monitor a topic. For example, an application might tail a topic to identify specific records, or just as a monitoring or debugging aid. One might only care about records that were published when the stateless consumer was online, so concerns such as persisting offsets and resuming from the last processed record become largely irrelevant. A good example of where this is used routinely is the Kafdrop tool, which we will explore in one of the upcoming chapters. When the user clicks on a

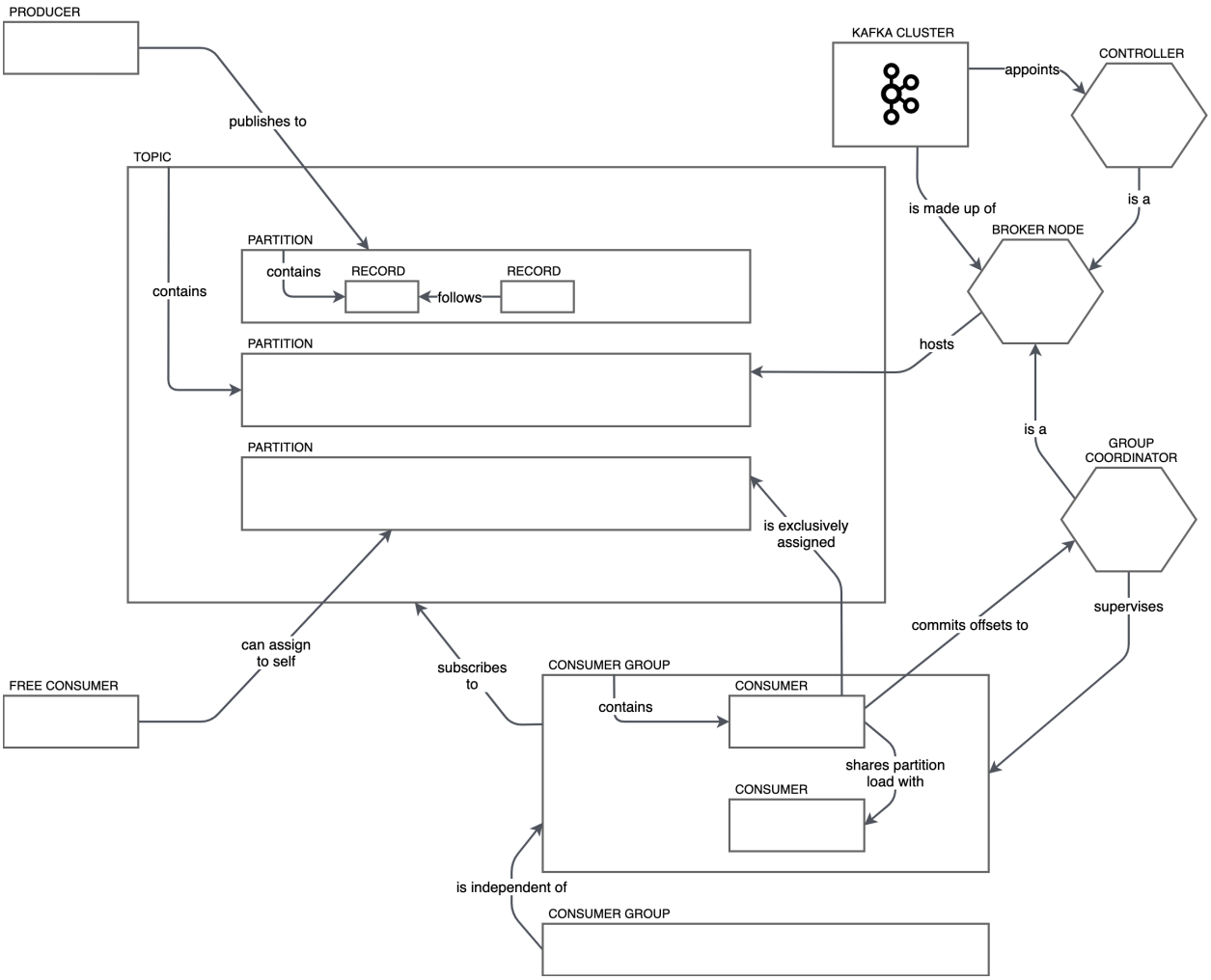
topic to view the records, Kafdrop creates a free consumer and assigns the requested partition to it, reading the records from the supplied offsets. Navigating to a different topic or partition will reset the consumer, discarding any prior state.

One scenario that benefits from free consumers is the implementation of the *sync-over-async* pattern using Kafka. For example, a producer might issue a command-style request (or query) to a downstream consumer and expect a response on the same or different topic. The initiating producer might be operating in a synchronous context; for example, it might be responding to a synchronous request of its own, and so it has no choice but to wait for the downstream response before proceeding. To complicate matters, there might be multiple such initiators in operation, and it is essential that the response is processed by the same initiator that issued the original request.

The sync-over-async scenario is a special case of the stateless consumer scenario presented above. The initiator starts by assigning itself all partitions of the response topic and resetting the offsets to the high-water mark. It then publishes the request command with a unique identifier that will be echoed in the response. (Typically, this is a UUID.) The downstream consumer will eventually process the message and publish its response. Meanwhile, the initiator will poll the topic for responses, filtering by ID. Eventually, either the response will arrive within a set deadline, or the initiator will time out. Either way, the assignment of the partitions to the initiator is temporary, and no state is preserved between successive assignments.

Summary of core concepts

The illustration below outlines the relationship between the core concepts presented in this chapter, including entities such as the cluster, broker nodes, producers, topics, partitions, consumers, and consumer groups.



Relationships between core concepts

The key takeaways are:

- **A cluster hosts multiple topics**, each having an assigned leader and zero or more follower replicas.
- **Topics are subdivided into partitions**, with each partition forming an independent, totally-ordered sequence within a wider, partially-ordered stream.
- **Multiple producers are able to publish to a topic**, picking a partition at will. The partition may be selected directly — by specifying a partition number, or indirectly — by way of a record key, which deterministically hashes to a partition number.
- **Partitions in a topic can be load-balanced** across a population of consumers in a consumer group, allocating partitions approximately evenly among the members of that group.
- **A consumer in a group is not guaranteed a partition assignment**. Where the group's population outnumbers the partitions, some consumers will remain idle until this balance equalises or tips in favour of the other side.

- **A consumer will commit the offset of a record when it is done processing it**. The commits are directed to a consumer coordinator, which will end up written to an internal `__consumer_offsets` topic. The offset of the record is incremented by one before committing, to prevent unnecessary replay.
- **Partitions may be manually assigned to free consumers**. If necessary, an entire topic may be assigned to a single free consumer — this is done by individually assigning all partitions.

Event streaming platforms are a highly effective building block in the construction of modular, loosely-coupled, event-driven applications. Within the world of event streaming, Kafka has solidified its position as the go-to open-source solution that is both flexible and highly performant. Concurrency and parallelism are at the heart of Kafka's architecture, forming partially-ordered event streams that can be load-balanced across a scalable consumer ecosystem. A simple reconfiguration of consumers and their encompassing groups can bring about vastly different event distribution and processing semantics; shifting the offset commit point can invert the delivery guarantee from an at-most-once to an at-least-once model.

The consumer group is a somewhat understated concept that is pivotal to the versatility of an event streaming platform. By simply varying the affinity of consumers with their groups, one can arrive at vastly different distribution topologies — from a topic-like, pub-sub behaviour to an MQ-style, point-to-point model. Because records are never truly consumed (the advancing offset only creates the illusion of consumption), one can concurrently superimpose disparate distribution topologies over a single event stream.