Apache Kafka

Matthias J. Sax
Confluent Inc., Palo Alto, CA, USA

Definitions

Apache Kafka (Apache Software Foundation 2017b; Kreps et al. 2011; Goodhope et al. 2012; Wang et al. 2015; Kleppmann and Kreps 2015) is a scalable, fault-tolerant, and highly available distributed streaming platform that can be used to store and process data streams.

Kafka consists of three main components:

- the Kafka cluster,
- the Connect framework (Connect API),
- and the Streams programming library (Streams API).

The Kafka cluster stores data streams, which are sequences of messages/events continuously produced by applications and sequentially and incrementally consumed by other applications. The Connect API is used to ingest data into Kafka and export data streams to external systems like distributed file systems, databases, and others. For data stream processing, the Streams API allows developers to specify sophisticated stream processing pipelines that read input streams from the Kafka cluster and write results back to Kafka.

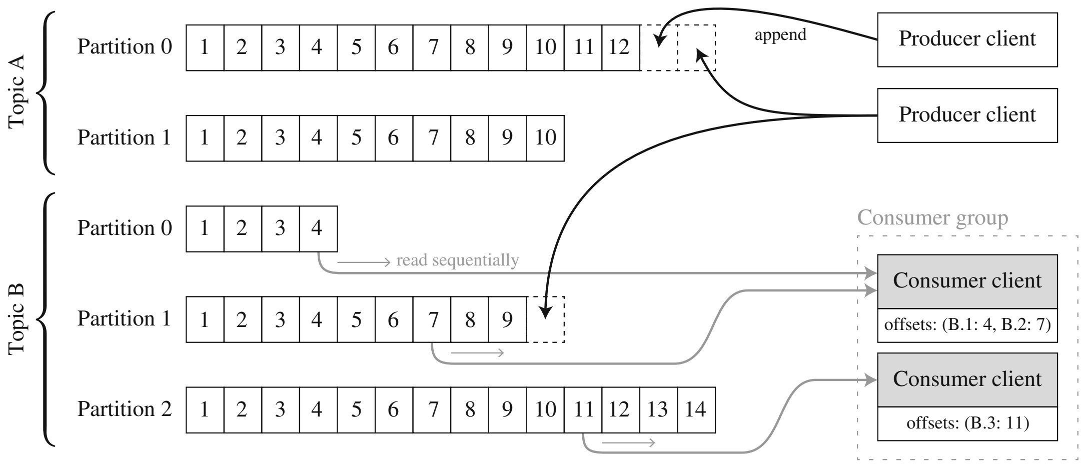
Kafka supports many different use cases categories such as traditional publish-subscribe messaging, streaming ETL, and data stream processing.

Overview

A Kafka cluster provides a publish-subscribe messaging service (Fig. 1). Producer clients (publishers) write messages into Kafka, and consumer clients (subscribers) read those messages from Kafka. *Messages* are stored in Kafka servers called *brokers* and organized in named *topics*. A topic is an append-only sequence of messages, also called a log. Thus, each time a message is written to a topic, it is appended to the end of the log. A Kafka message is a key-value pair where key and value are variable-length byte arrays. Additionally, each message has a time stamp that is stored as 64-bit integer.

Topics are divided into *partitions*. When a message is written to a topic, the producer must specify the partition for the message. Producers can use any partitioning strategy for the messages they write. By default, messages are hash-partitioned by key, and thus all messages with the same key are written to the same partition. Each message has an associated *offset* that is the message's position within the partition, i.e., a monotonically increasing sequence number. The message's offset is implicitly determined by the order in which messages are appended to a partition. Hence, each message within a topic is

© Springer International Publishing AG 2018
S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_196-1



Apache Kafka, Fig. 1 Kafka topics are divided into partitions that are ordered sequences of messages. Multiple producers can write simultaneously into the same topic. Consumers track their read progress and can form a consumer group to share the read workload over all consumers within the group (Source: Kleppmann and Kreps 2015)

uniquely identified by its partition and offset. Kafka guarantees strict message ordering within a single partition, i.e., it guarantees that all consumers reading a partition receive all messages in the exact same order as they were appended to the partition. There is no ordering guarantee between messages in different partitions or different topics.

Topics can be written by multiple producers at the same time. If multiple producers write to the same partition, their messages are interleaved. If consumers read from the same topic, they can form a so-called consumer group. Within a consumer group, each individual consumer reads data from a subset of partitions. Kafka ensures that each partition is assigned to exactly one consumer within a group. Different consumer groups or consumers that don't belong to any consumer group are independent from each other. Thus, if two consumer groups read the same topic, all messages are delivered to both groups. Because consumers are independent from each other, each consumer can read messages at its own pace. This results in decoupling—a desirable property for a distributed system—and makes the system robust against stragglers. In summary, Kafka supports multiple producers and can deliver the same data to multiple consumers.

Kafka Brokers

Kafka brokers store messages reliably on disk. In contrast to traditional messaging/publish-subscribe systems, Kafka can be used for long-term storage of messages, because Kafka does not delete messages after delivery. Topics are configured with a so-called *retention time* that specifies how long a message should be stored. Topic retention can also be specified in bytes instead of time, to apply an upper bound on disk space. If the retention boundaries are reached, Kafka truncates partitions at the end of the log.

From a semantic point of view, messages are immutable facts, and thus it is not reasonable to support deleting individual messages from a topic. Users can only apply a "time-to-live" via topic retention to truncate old data.

Log Compaction

Kafka also supports so-called *compacted* topics. If a topic is configured for log compaction, users apply different *semantics* to the stored messages. While regular topics store *immutable facts*, a compacted topic can be used to store *updates*. Note that a compacted topic is still an append-only sequence of messages, and there are no in-place updates. Appending an update message to a compacted topic implies that a newer messages "replaces" older messages with the same key. The difference of compacted topics to topics with log retention is that Kafka guarantees that the latest update of a key is never deleted, while older updates can be garbage collected.

Log compaction is applied on a per-partition basis; thus updates for the same key should be written to the same partition. For performance reasons, brokers don't delete older messages immediately, but compaction is triggered as background process in regular intervals. In contrast to "regular" topics, compacted topics also support delete semantics for individual record via so-called *tombstone* messages. A tombstone is a message with a `null` value, and it indicates that all previous updates for the corresponding key can be deleted (including the tombstone itself).

Scaling and Load Balancing

As described in section "Overview," messages are stored in topics, and topics are divided into partitions. Partitions allow brokers to scale out horizontally and to balance load within the cluster, because partitions are independent units within a topic. Even if partitions of the same topic are stored at different brokers, there is no need for broker synchronization, and thus a Kafka cluster scales linearly with the number of brokers. A single broker only limits the capacity of a single partition, but because topics can be created with an arbitrary number of partitions, this is not a limitation in practice. Overall the read/write throughput and storage requirements of topics are not limited by the size of a single server, and the cluster capacity can be increased by adding new brokers to the system. Last but not least, partitions can be reassigned from one broker to another to balance load within a cluster.

There is no master node in a Kafka cluster: all brokers are able to perform all services provided by the cluster. This design supports linear scale-out, as a master node could become a bottleneck. For broker coordination, Kafka uses Apache ZooKeeper (Apache Software Foundation 2017d; Hunt et al. 2010), a scalable, fault-tolerant, and highly available distributed coordination service. Kafka uses ZooKeeper to store all cluster metadata about topics, partitions, partition-to-broker mapping, etc., in a reliable and highly available manner.

Fault Tolerance and High Availability

To ensure fault tolerance and high availability, partitions can be replicated to multiple brokers. Each topic can be configured with an individual replication factor that indicates how many copies of a partition should be maintained. To ensure strict message ordering guarantees per partitions, replication uses a *leader-follower* pattern. Each partition has a single leader and a configurable number of followers. All read and write requests are handled by the leader, while the followers replicate all writes to the leader in the background. If the broker hosting the leader fails, Kafka initiates a leader election via ZooKeeper, and one of the followers becomes the new leader. All clients will be updated with the new leader information and send all their read/write request to the new leader. If the failed broker recovers, it will rejoin the cluster, and all hosted partitions become followers.

Message Delivery

Reading data from Kafka works somewhat differently compared to traditional messaging/publish-subscribe systems. As mentioned in section "Kafka Brokers," brokers do not delete messages after delivery. This design decision has multiple advantages:

- Brokers do not need to track the reading progress of consumers. This allows for an increased read throughput, as there is no progress tracking overhead for the brokers.
- It allows for in-order message delivery. If brokers track read progress, consumers need to acknowledge which messages they have processed successfully. This is usually done on a per-message basis. Thus, if an earlier message is not processed successfully, but a later message is processed successfully, redelivery of the first message happens out of order.

- Because brokers don't track progress and don't delete data after delivery, consumers can go back in time and reprocess old data again. Also, newly created consumers can retrieve older data.

The disadvantage of this approach is that consumer clients need to track their progress themselves. This happens by storing the offset of the next message a consumer wants to read. This offset is included in the read request to the broker, and the broker will deliver consecutive messages starting at the requested offset to the consumer. After processing all received messages, the consumer updates its offset accordingly and sends the next read request to the cluster.

If a consumer is stopped and restarted later on, it usually should continue reading where it left off. To this end, the consumer is responsible for storing its offset reliably so it can retrieve it on restart. In Kafka, consumers can commit their offsets to the brokers. On offset commit brokers store consumer offsets reliably in a special topic called *offset topic*. As offset topic is also partitioned and replicated and is thus scalable, fault-tolerant, and highly available. This allows Kafka to manage a large number of consumers at the same time. The offset topic is configured with log compaction enabled (cf. section "Log Compaction") to guarantee that offsets are never lost.

Delivery Semantics

Kafka supports multiple delivery semantics, namely, *at-most-once*, *at-least-once*, and *exactly once*. What semantics a user gets depends on multiple factors like cluster/topic configuration as well as client configuration and user code.

It is important to distinguish between the write and read path when discussing delivery semantics. Furthermore, there is the concept of end-to-end processing semantics that applies to Kafka's Streams API. In this section, we will only cover the read and write path and refer to section "Kafka Streams" for end-to-end processing semantics.

Writing to Kafka

When a producer writes data into a topic, brokers acknowledge a successful write to the producer. If a producer doesn't receive an acknowledgement, it can ignore this and follow at-most-once semantics, as the message might not have been written to the topic and thus could be lost. Alternatively, a producer can retry the write resulting in at-least-once semantics. The first write could have been successful, but the acknowledgement might be lost. Kafka also supports exactly once writes by exploiting idempotence. For this, each message is internally assigned a unique identifier. The producer attaches this identifier to each message it writes, and the broker stores the identifier as metadata of each message in the topic. In case of a producer write retry, the broker can detect the duplicate write by comparing the message identifiers. This deduplication mechanism is internal to producer and broker and does not guard against application-level duplicates.

Atomic Multi-partition Writes Kafka also support atomic multi-partition writes that are called *transactions*. A Kafka transaction is different to a database transaction, and there is no notion of ACID guarantees. A transaction in Kafka is similar to an atomic write of multiple messages that can span different topics and/or partitions. Due to space limitations, we cannot cover the details of transactional writes and can only give a brief overview. A transactional producer is first initialized for transactions by performing a corresponding API call. The broker is now ready to accept transactional writes for this producer. All messages sent by the producer belong to the current transaction and won't be delivered to any consumer as long as the transaction is not completed. Messages within a transaction can be sent to any topic/partition within the cluster. When all messages of a transaction have been sent, the producer commits the transaction. The broker implements a two-phase commit protocol to commit a transaction, and it either successfully "writes" all or none of the messages belonging to a transaction. A producer can also abort a transaction; in this case, none of the messages will be deleted from the log, but all messages will be marked as aborted.

Reading from Kafka

As discussed in section "Message Delivery," consumers need to track their read progress themselves. For fault-tolerance reasons, consumers commit their offsets regularly to Kafka. Consumers can apply two strategies for this: after receiving a message, they can either first commit the offset and process the message afterwards, or they do it in reverse order and first process the message and commit the offset at the end. The commit-first strategy provides at-most-once semantics. If a message is received and the offset is committed before the message is processed, this message would not be redelivered in case of failure: after a failure, the consumer would recover its offsets as the last committed offset and thus would resume reading after the failed message.

In contrast, the process-first strategy provides at-least-once semantics. Because the consumer doesn't update its offsets after processing the receive message successfully, it would always fall back to the old offset after recovering from an error. Thus, it would reread the processed message, resulting in potential duplicates in the output.

Transactional Consumers Consumers can be configured to read all (including aborted messages) or only committed messages (cf. paragraph *Atomic multi-partition writes* in section "Writing to Kafka"). This corresponds to a read uncommitted and read committed mode similar to other transactional systems. Note that aborted messages are not deleted from the topics and will be delivered to all consumers. Thus, consumers in read committed mode will filter/drop aborted messages and not deliver them to the application.

Kafka Connect Framework

Kafka Connect—or the Connect API—is a framework for integrating Kafka with external systems like distributed file systems, databases, key-value stores, and others. Internally, Kafka Connect uses producer/consumer clients as described in section "Kafka Brokers," but the framework implements much of the functionality and best practices that would otherwise have to be implemented for each system. It also allows running in a fully managed, fault-tolerant, and highly available manner.

The Connect API uses a *connector* to communicate with each type of external system. A *source connector* continuously reads data from an external source system and writes the records into Kafka, while a *sink connector* continuously consumes data from Kafka and sends the records to the external system. Many connectors are available for a wide variety of systems, including HDFS, S3, relational databases, document database systems, other messaging systems, file systems, metric systems, analytic systems, and so on. Developers can create connectors for other systems.

Kafka Connect can be either used as standalone or deployed to a cluster of machines. To connect with an external system, a user creates a configuration for a connector that defines the specifics of the external system and the desired behavior, and the user uploads this configuration to one of the Connect workers. The worker, which is running in a JVM, deploys the connector and distributes the connector's *tasks* across the cluster. Source connector tasks load data from the external system and generate records, which Connect then writes to the corresponding Kafka topics. For sink connector tasks, Connect consumes the specified topics and passes these records to the task, which then is responsible for sending the records to the external system.

Single-Message Transforms

The Connect API allows to specify simple transformation—so-called single-message transforms, SMT—for individual messages that are imported/exported into/from Kafka. Those transformations are independent of the connector and allow for stateless operations. Standard transformation functions are already provided by Kafka, but it is also possible to implement custom transformations to perform initial data cleaning. If SMTs are not sufficient because a more complex transformation is required, the Streams API (described in the next section) can be used instead.

Kafka Streams

Kafka Streams—or the Streams API—is the stream processing library of Apache Kafka. It provides a high-level DSL that supports stateless as well as stateful stream processing operators like joins, aggregations, and windowing. The Streams API also supports exactly once processing guarantees and event-time semantics and handles out-of-order and late-arriving data. Additionally, the Streams API introduces *tables* as a first-class abstraction next to data *streams*. It shares a few ideas with Apache Samza (cf. article on "Apache Samza") (Apache Software Foundation 2017c; Noghabi et al. 2017; Kleppmann and Kreps 2015) such as building on top of Kafka's primitives for fault tolerance and scaling. However, there are many notable differences to Samza: for example, Kafka Streams is implemented as a library and can run in any environment, unlike Samza, which is coupled to Apache Hadoop's resource manager YARN (Apache Software Foundation 2017a; Vavilapalli et al. 2013); it also provides stronger processing guarantees and supports both streams and tables as core data abstractions.

Streams and Tables

Most stream processing frameworks provide the abstraction of a *record stream* that is an append-only sequence of immutable facts. Kafka Streams also introduces the notion of a *changelog* stream, a mutable collection of data items. A changelog stream can also be described as a continuously updating table. The analogy between a changelog and a table is called the *stream-table duality* (Kleppmann 2016, 2017). Supporting tables as first-class citizens allow to enrich data streams via stream-table joints or populate tables as self-updating caches for an application.

The concept of changelogs aligns with the concept of compacted topics (cf. section "Log Compaction"). A changelog can be stored in a compacted topic, and the corresponding table can be recreated by reading the compacted topic without data loss as guaranteed by the compaction contract.

Kafka Streams also uses the idea of a changelog stream for (windowed) aggregations. An aggregation of a record stream yields both a table and a changelog stream as a result—not a record stream. Thus, the table always contains the current aggregation result that is updated for each incoming record. Furthermore, each update to the result table is propagated downstream as a changelog record that is appended to the changelog stream.

State Management

Operator state is a first-class citizen in Kafka's Streams API similar to Samza and uses the aforementioned table abstraction. For high performance, state is kept local to the stream processing operators using a RocksDB (Facebook Inc. 2017) store. Local state is not fault-tolerant, and thus state is additionally backed by a topic in the Kafka cluster. Those topics have log compaction (cf. section "Log Compaction") enabled and are called *changelog topics*. Using log compaction ensures that the size of the changelog topic is linear in the size of the state. Each update to the store is written to the changelog topic; thus, the persistent changelog topic is the source of truth, while the local RocksDB store is an ephemeral materialized view of the state.

If an application instance fails, another instance can recreate the state by reading the changelog topic. For fast fail-over, Kafka Streams also support *standby replicas*, the hold hot standbys of state store. Standby replicas can be maintained by continuously reading all changes to the primary store from the underlying changelog topic.

Fault Tolerance and Scaling

Kafka Streams uses the same scaling/parallelism abstraction as Samza, namely, partitions. This is a natural choice as Kafka Streams reads input data from partitioned topics. Each input topic partition is mapped to a *task* that processed the records of this partition. Tasks are independent units of parallelism and thus can be executed by different threads that might run on different machines. This allows to scale out a Kafka Streams application by starting multiple instances on different machines. All application instances form a consumer group, and thus Kafka assigns topic partitions in a load balanced manner to all application instances (cf. section "Kafka Brokers").

Because Kafka Streams is a library, it cannot rely on automatic application restarts of failed instances. Hence, it relies on the Kafka cluster to detect failures. As mentioned above, a Streams application forms a consumer group, and the Kafka cluster monitors the liveness of all members of the group. In case of a failure, the cluster detects a dead group member and reassigns the corresponding input topic partitions of the failed application instance to the remaining instances. This process is called an consumer group *rebalance*. During a rebalance, Kafka Streams also ensures that operator state is migrated from the failing instance (cf. section "State Management"). Kafka's consumer group management mechanism also allows for fully elastic deployments. Application instances can be added or removed during runtime without any downtime: the cluster detects joining/leaving instances and rebalances the consumer group automatically. If new instances are joining, partitions are *revoked* from existing members of the groups and assigned to the new members to achieve load balancing. If members are leaving a consumer group, this is just a special case of fail-over, and the partitions of the leaving instance are assigned to the remaining ones. Note that in contrast to a fail-over rebalance, a scaling rebalance guarantees a clean hand over of partitions, and thus each record is processed exactly once.

Time Semantics

Time is a core concept in stream processing, and the operators in Kafka's stream processing DSL are based on time: for example, windowed aggregations require record time stamps to assign records to the correct time windows. Handling time also requires to handle late-arriving data, as record might be written to a topic out of order (note: Kafka guarantees offset based in-order delivery; there is no time stamp-based delivery guarantee).

Stream processing with Kafka supports three different time semantics: event time, ingestion time, and processing time. From a Streams API point of view, there are only two different semantics though: *event time* and *processing time*.

Processing time semantics are provided when there is no record time stamp, and thus Streams needs to use wall clock time when processing data for any time-based operation like windowing. Processing time has the disadvantage that there is no relationship between the time when data was created and when data gets processed. Furthermore, processing time semantics is inherently non-deterministic.

Event time semantics are provided when the record contains a time stamp in its payload or metadata. This time-stamp is assigned when a record is created and thus allows for deterministic data reprocessing. Applications may include a time stamp in the payload of the message, but the processing of such time stamps is then application-dependent. For this reason, Kafka supports record metadata time stamps that are stored in topics and automatically set by the producer when a new record is created.

Additionally, Kafka topics can be configured to support *ingestion time* for time-based operations: ingestion time is the time when data is appended to the topic, i.e., the current broker wall clock time on write. Ingestion time is an approximation of event time, assuming that data is written to a topic shortly after it was created. It can be used if producer applications don't provide a record metadata time stamp. As ingestion time is an approximation of event time, the Streams API is agnostic to ingestion time (it is treated as a special case of event time).

Exactly Once Processing Semantics

Processing an input record can be divided into three parts: first, the actual processing including any state updates; second, writing the result records to the output topics; third, recording the progress by committing the consumer offset. To provide exactly once processing semantics, all three parts must be performed "all or nothing."

As described in section "State Management," updating operator state is actually a write to a changelog topic that is the source of truth (the local state is only a materialized view). Furthermore, committing input offsets is a write to the special *offset topic* as discussed in section "Message Delivery." Hence, all three parts use the same underlying operation: writing to a topic.

This allows the Streams API to leverage Kafka's transactions (cf. section "Writing to Kafka") to provide end-to-end exactly once stream processing semantics. All writes that happen during the processing of a record are part of the same transaction. As a transaction is an atomic multi-partition write, it ensures that either all writes are committed or all changes are aborted together.

Summary

Apache Kafka is a scalable, fault-tolerant, and highly available distributed streaming platform. It allows fact and changelog streams to be stored and processed, and it exploits the stream-table duality in stream processing. Kafka's transactions allow for exactly once stream processing semantics and simplify exactly once end-to-end data pipelines. Furthermore, Kafka can be connected to other systems via its Connect API and can thus be used as the central data hub in an organization.

Cross-References

- ▶ Apache Flink
- ▶ Apache Heron
- ▶ Apache Samza
- ▶ Continuous Queries
- ▶ Publish/Subscribe

References

Apache Software Foundation (2017a) Apache Hadoop project web page. https://hadoop.apache.org/

Apache Software Foundation (2017b) Apache Kafka project web page. https://kafka.apache.org/

Apache Software Foundation (2017c) Apache Samza project web page. https://samza.apache.org/

Apache Software Foundation (2017d) Apache ZooKeeper project web page. https://zookeeper.apache.org/

Facebook Inc (2017) RocksDB project web page. http://rocksdb.org/

Goodhope K, Koshy J, Kreps J, Narkhede N, Park R, Rao J, Ye VY (2012) Building Linkedin's real-time activity data pipeline. IEEE Data Eng Bull 35(2):33–45. http://sites.computer.org/debull/A12june/pipeline.pdf

Hunt P, Konar M, Junqueira FP, Reed B (2010) ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX conference on USENIX annual technical conference, USENIX ATC'10. USENIX Association, Berkeley, p 11. http://dl.acm.org/citation.cfm?id=1855840.1855851

Kleppmann M (2016) Making sense of stream processing, 1st edn. O'Reilly Media Inc., 183 pages

Kleppmann M (2017) Designing data-intensive applications. O'Reilly Media Inc., Sebastopol

Kleppmann M, Kreps J (2015) Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Eng Bull 38(4):4–14. http://sites.computer.org/debull/A15dec/p4.pdf

Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp 1–7

Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza: stateful scalable stream processing at LinkedIn. Proc VLDB Endow 10(12):1634–1645. https://doi.org/10.14778/3137765.3137770

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: 4th ACM symposium on cloud computing (SoCC). https://doi.org/10.1145/2523616.2523633

Wang G, Koshy J, Subramanian S, Paramasivam K, Zadeh M, Narkhede N, Rao J, Kreps J, Stein J (2015) Building a replicated logging system with Apache Kafka. PVLDB 8(12):1654–1655. http://www.vldb.org/pvldb/vol8/p1654-wang.pdf