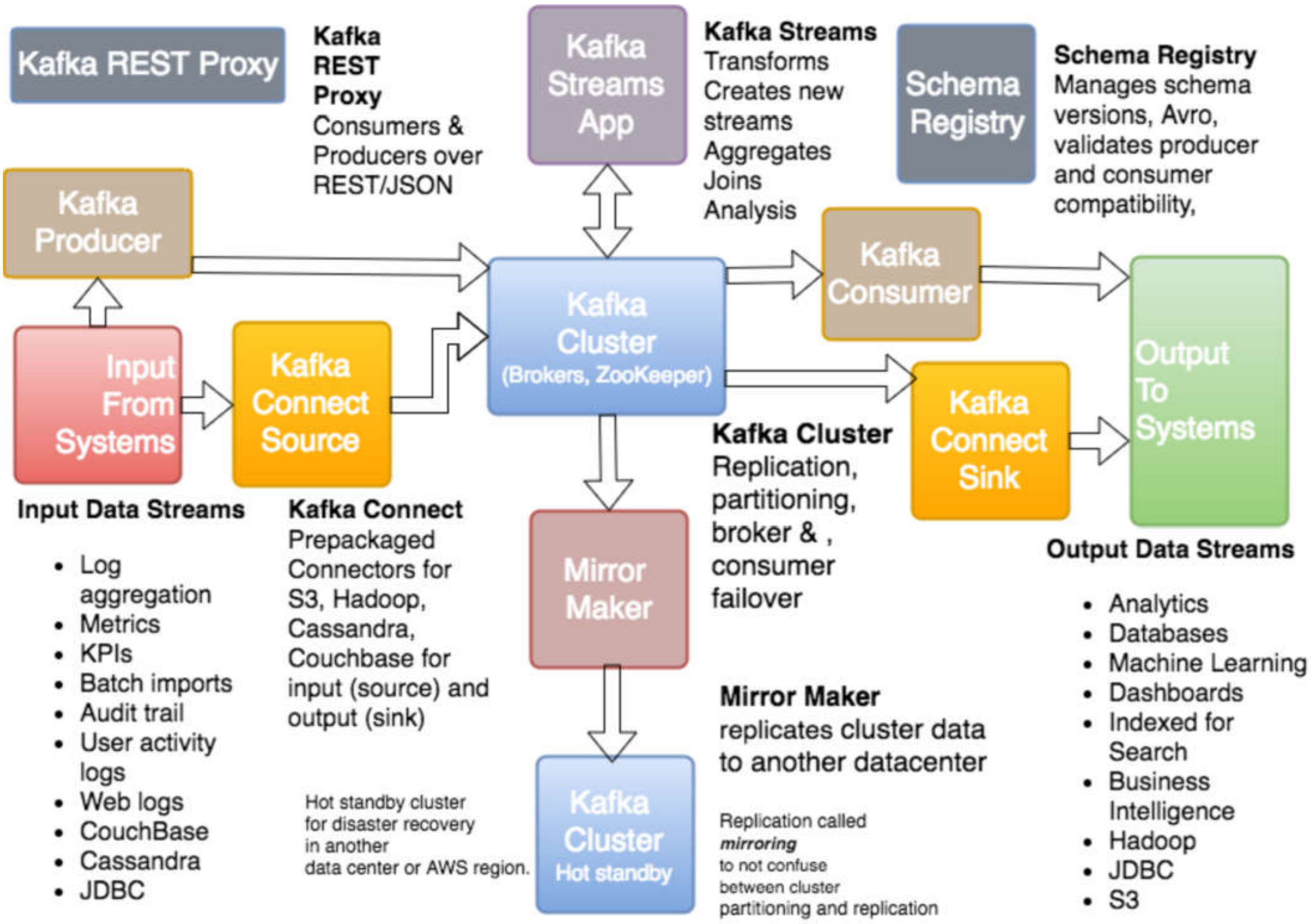
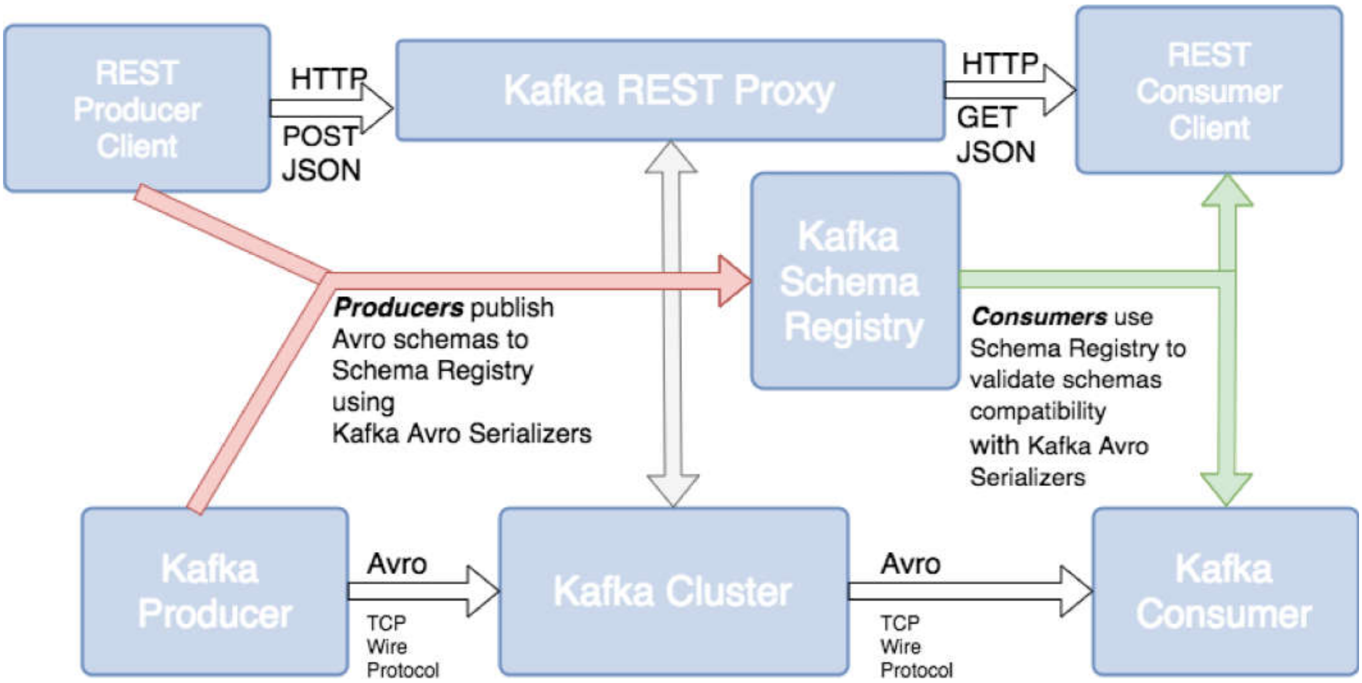
Outline

- Kafka Connect
- kafka相关文章

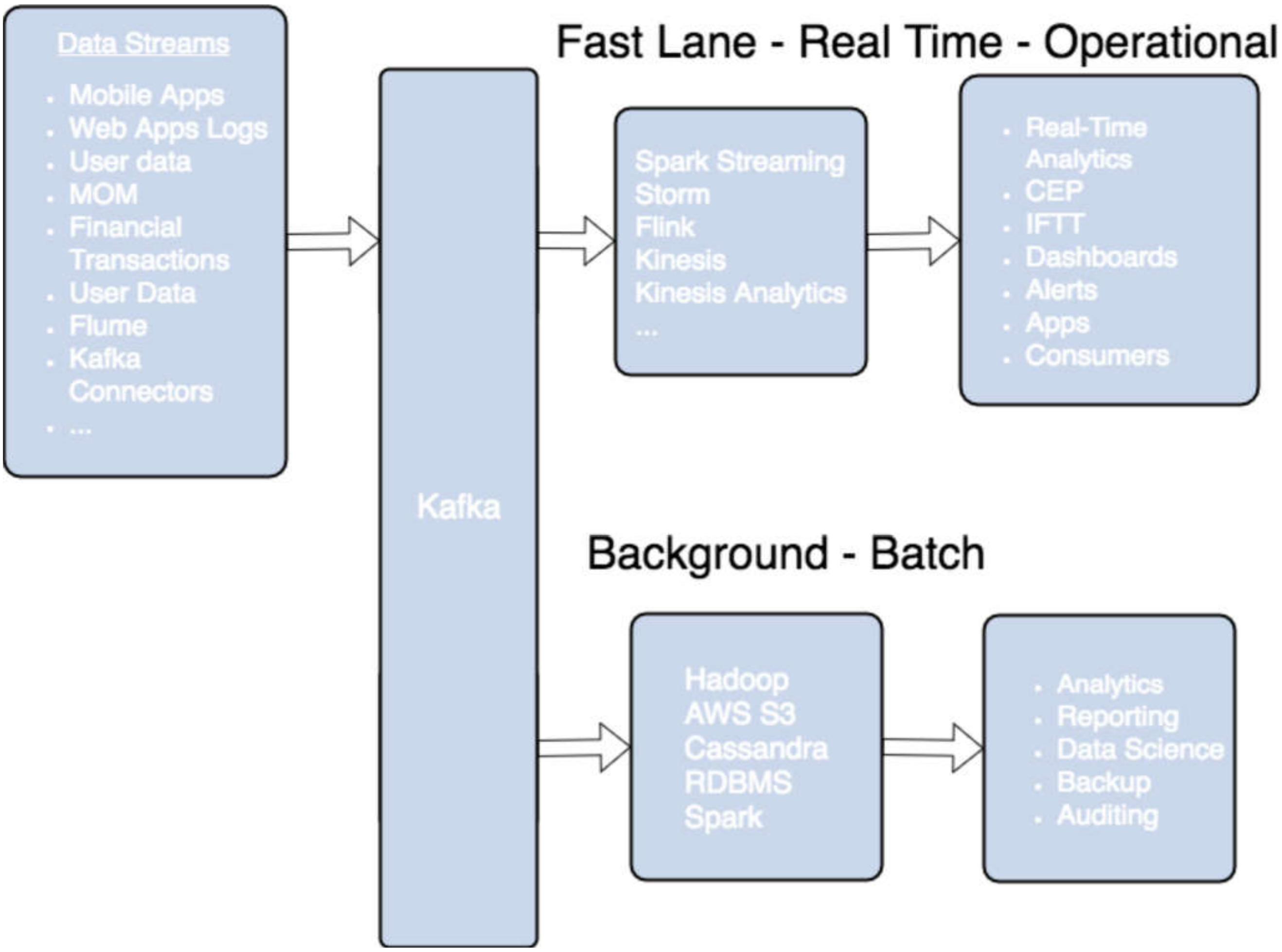
The Kafka Ecosystem - Kafka Core, Kafka Streams, Kafka Connect, Kafka REST Proxy, and the Schema Registry


Kafka REST Proxy
Kafka REST Proxy Consumers & Producers over REST/JSON
Kafka Streams App
Kafka Streams Transforms Creates new streams Aggregates Joins Analysis
Schema Registry
Schema Registry Manages schema versions, Avro, validates producer and consumer compatibility,
Kafka Producer
Kafka Consumer
Kafka Cluster (Brokers, ZooKeeper)
Output To Systems
Input From Systems
Kafka Connect Source
Kafka Connect Sink
Kafka Cluster Replication, partitioning, broker & , consumer failover
Input Data Streams
Log aggregation
Metrics
KPIs
Batch imports
Audit trail
User activity logs
Web logs
CouchBase
Cassandra
JDBC
Kafka Connect Prepackaged Connectors for S3, Hadoop, Cassandra, Couchbase for input (source) and output (sink)
Mirror Maker
Output Data Streams
Analytics
Databases
Machine Learning
Dashboards
Indexed for Search
Business Intelligence
Hadoop
JDBC
S3
Mirror Maker replicates cluster data to another datacenter
Hot standby cluster for disaster recovery in another data center or AWS region.
Kafka Cluster Hot standby
Replication called mirroring to not confuse between cluster partitioning and replication



Kafka Streaming Architecture

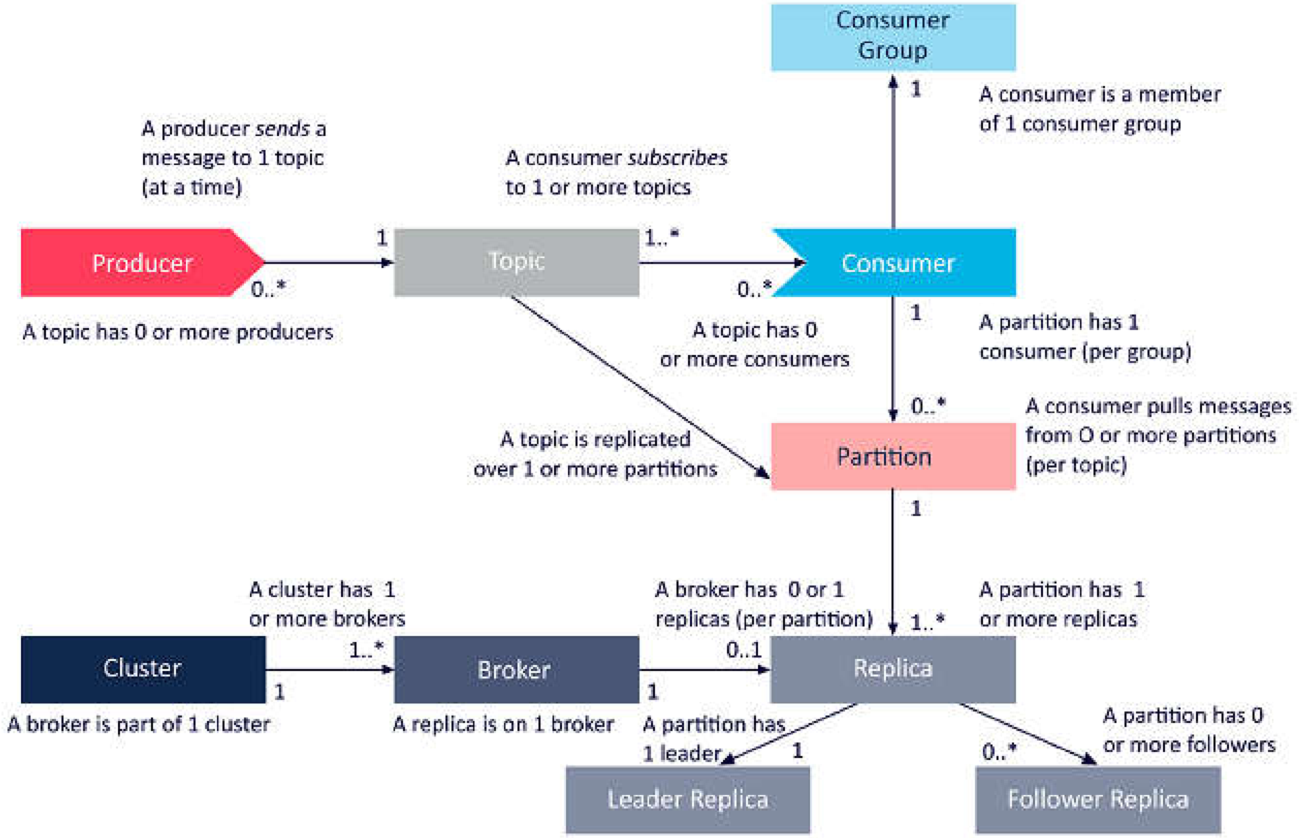
Kafka gets used most often for real-time streaming of data into other systems. Kafka is a middle layer to decouple your real-time data pipelines. Kafka core is not good for direct computations such as data aggregations, or CEP. Kafka Streaming which is part of the Kafka ecosystem does provide the ability to do real-time analytics. Kafka can be used to feed fast lane systems (real-time, and operational data systems) like Storm, Flink, Spark Streaming and your services and CEP systems. Kafka is also used to stream data for batch data analysis. Kafka feeds Hadoop. It streams data into your BigData platform or into RDBMS, Cassandra, Spark, or even S3 for some future data analysis. These data stores often support data analysis, reporting, data science crunching, compliance auditing, and backups.



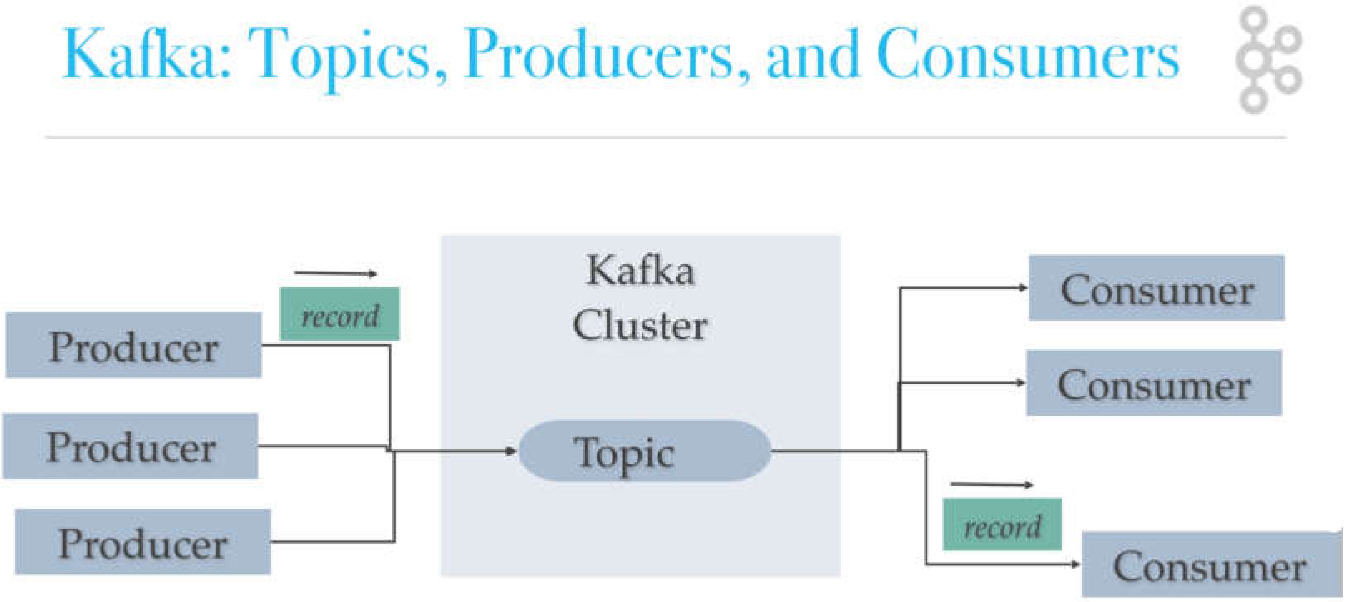
Kafka Architecture

The Kafka Components

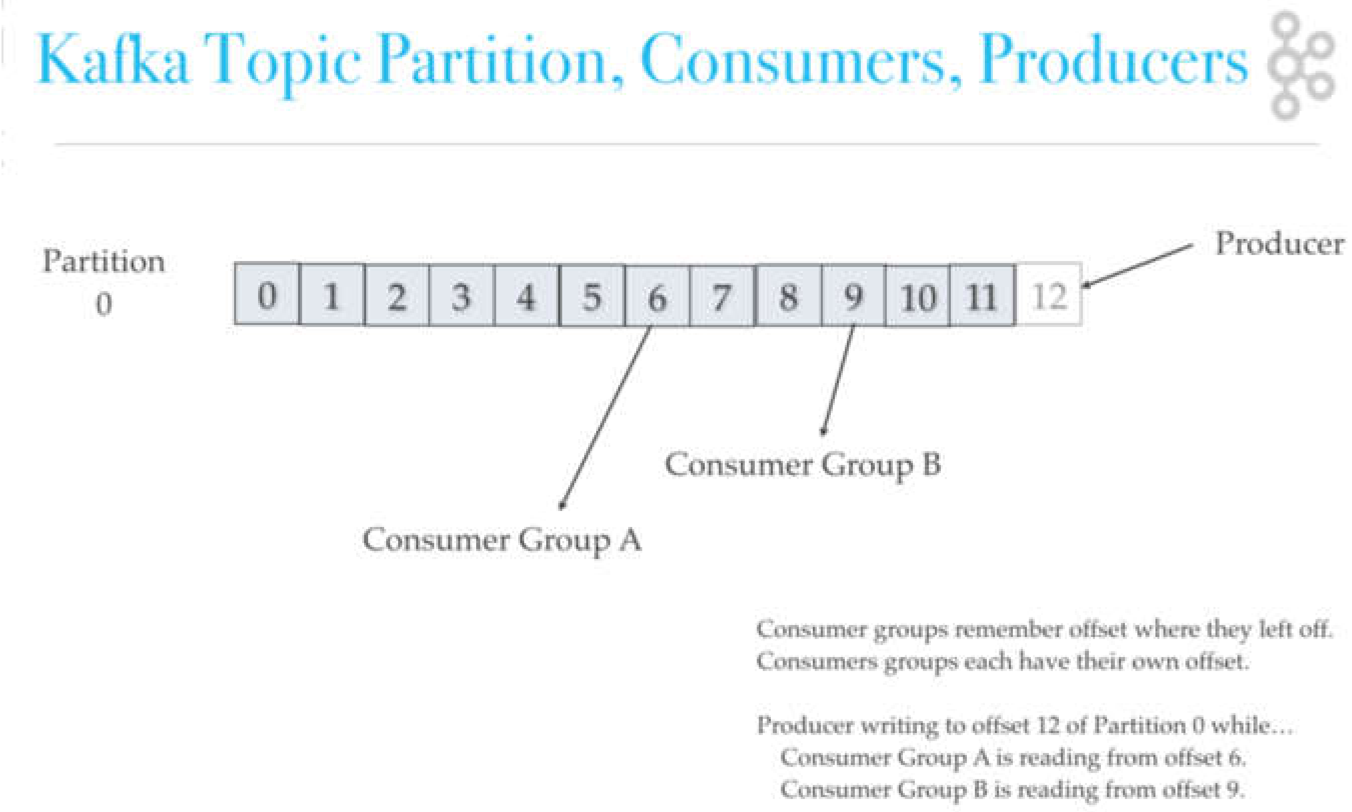
Kafka’ s main architectural components include Producers, Topics, Consumers, Consumer Groups, Clusters, Brokers, Partitions, Replicas, Leaders, and Followers. This simplified UML diagram describes the ways these components relate to one another:


Consumer Group
1
A consumer is a member of 1 consumer group
A producer *sends* a message to 1 topic (at a time)
A consumer *subscribes* to 1 or more topics
Producer
1
Topic
1..*
Consumer
0..*
0..*
A topic has 0 or more producers
A topic has 0 or more consumers
1
A partition has 1 consumer (per group)
0..*
A consumer pulls messages from 0 or more partitions (per topic)
A topic is replicated over 1 or more partitions
Partition
1
A cluster has 1 or more brokers
A broker has 0 or 1 replicas (per partition)
A partition has 1 or more replicas
1..*
Cluster
1..*
Broker
0..1
Replica
1
1
A broker is part of 1 cluster
A replica is on 1 broker
A partition has 1 leader
1
0..*
A partition has 0 or more followers
Leader Replica
Follower Replica

Topics, Producers and Consumers



Kafka: Topics, Producers, and Consumers



Kafka Topic Partition, Consumers, Producers

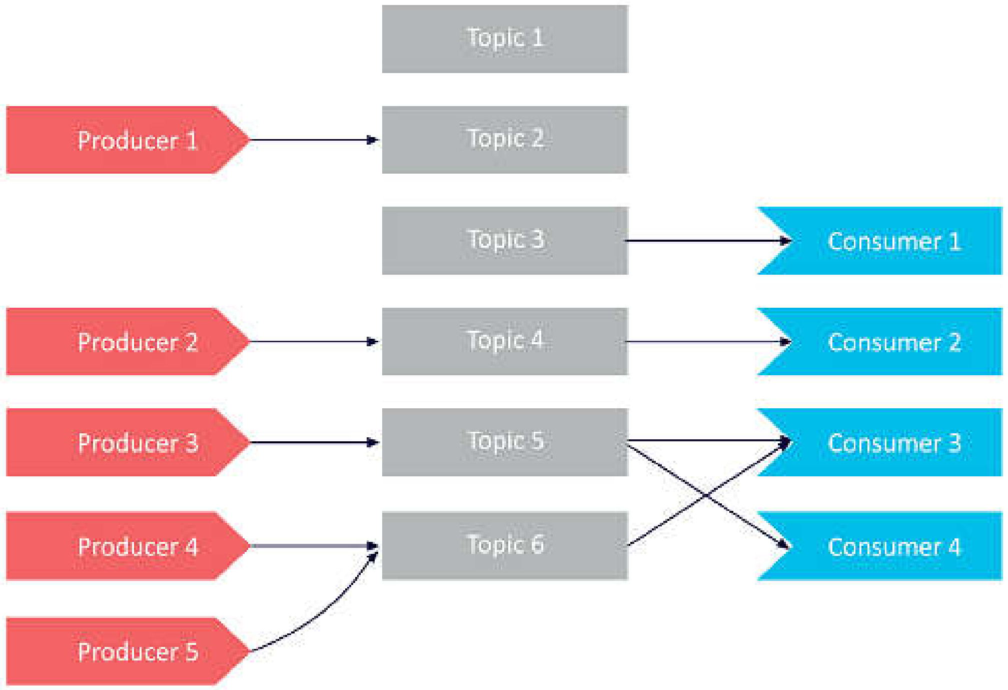
Consumer groups remember offset where they left off.
Consumers groups each have their own offset.

Producer writing to offset 12 of Partition 0 while…
Consumer Group A is reading from offset 6.
Consumer Group B is reading from offset 9.

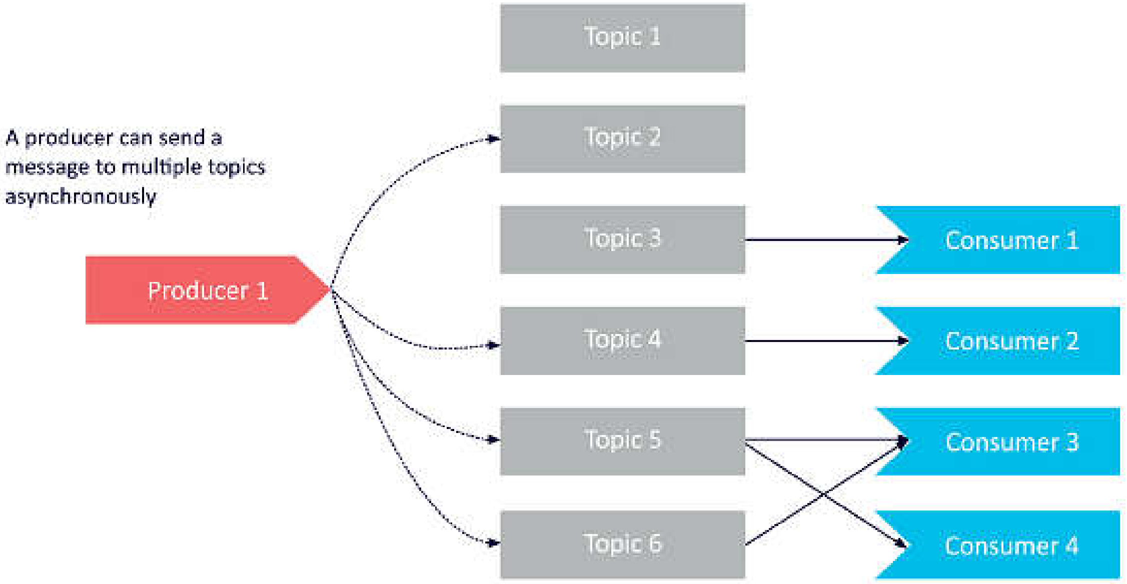
Kafka producers write to Topics. Kafka consumers read from Topics. A topic is associated with a log which is data structure on disk. Kafka appends records from a producer(s) to the end of a topic log. A topic log consists of many partitions that are spread over multiple files which can be spread on multiple Kafka cluster nodes. Consumers read from Kafka topics at their cadence and can pick where they are (offset) in the topic log. Each consumer group tracks offset from where they left off reading. Kafka distributes topic log partitions on different nodes in a cluster for high performance with horizontal scalability. Spreading partitions aids in writing data quickly. Topic log partitions are Kafka way to shard reads and writes to the topic log. Also, partitions are needed to have multiple consumers in a consumer group work at the same time. Kafka replicates partitions to many nodes to provide failover.

More Explaintion about Relationship between Producers, Topics, and Consumers

A producer sends a message to 1 topic(at a time):



While producers can only message to one topic at a time, they’ re able to send messages asynchronously. Using this technique allows a producer to functionally send multiple messages to multiple topics at once.

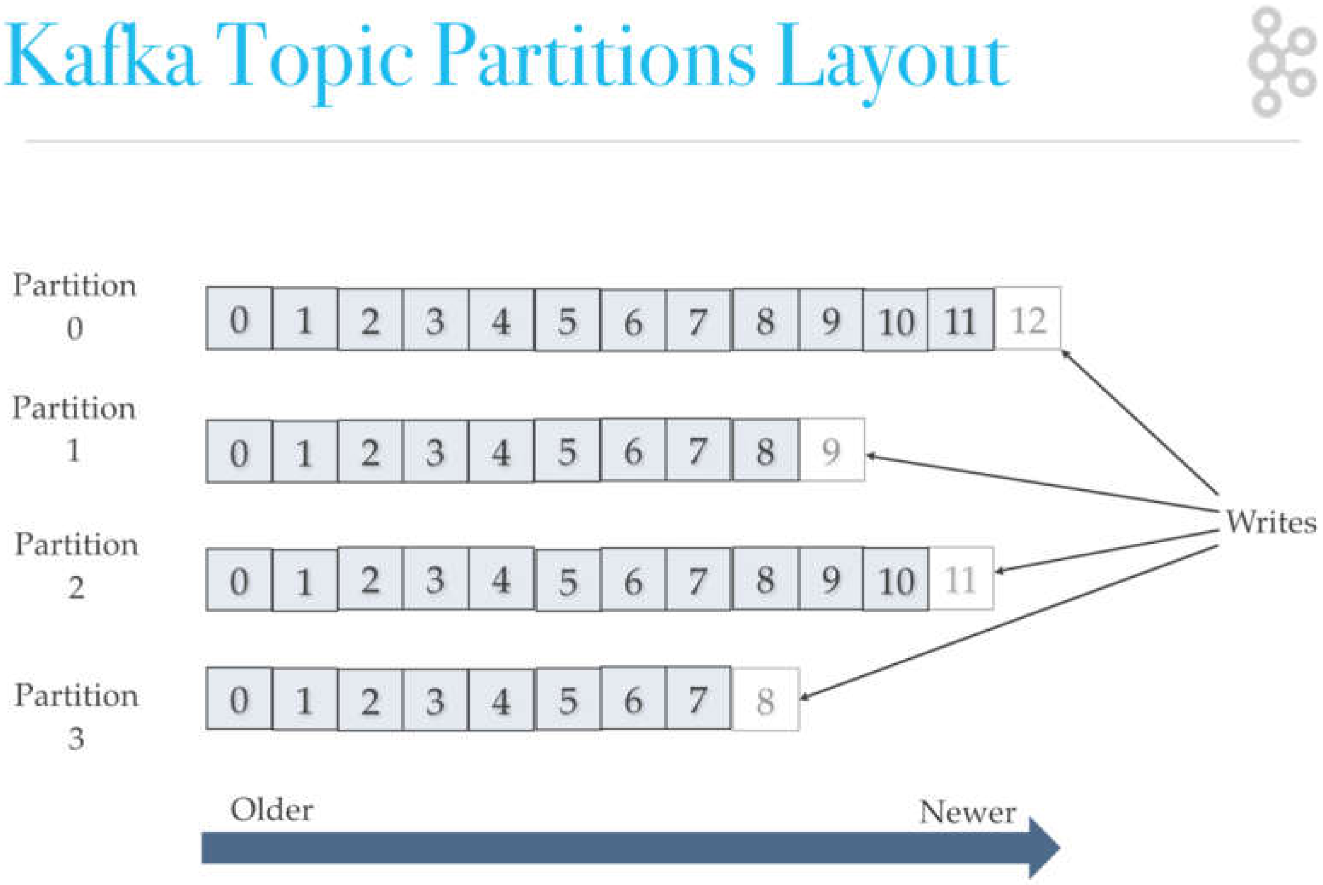


Kafka Topic Log Partition's Ordering and Cardinality

Kafka breaks topic logs up into partitions. A record is stored on a partition usually by record key if the key is present and round-robin if the key is missing (default behavior). The record key, by default, determines which partition a producer sends the record.

Kafka maintains record order only in a single partition. A partition is an ordered, immutable record sequence. Kafka continually appended to partitions using the partition as a structured commit log. Records in partitions are assigned sequential id number called the offset. The offset identifies each record location within the partition.

Topic partitions are a unit of parallelism - a partition can only be worked on by one consumer in a consumer group at a time. If a consumer stops, Kafka spreads partitions across the remaining consumer in the same consumer group.



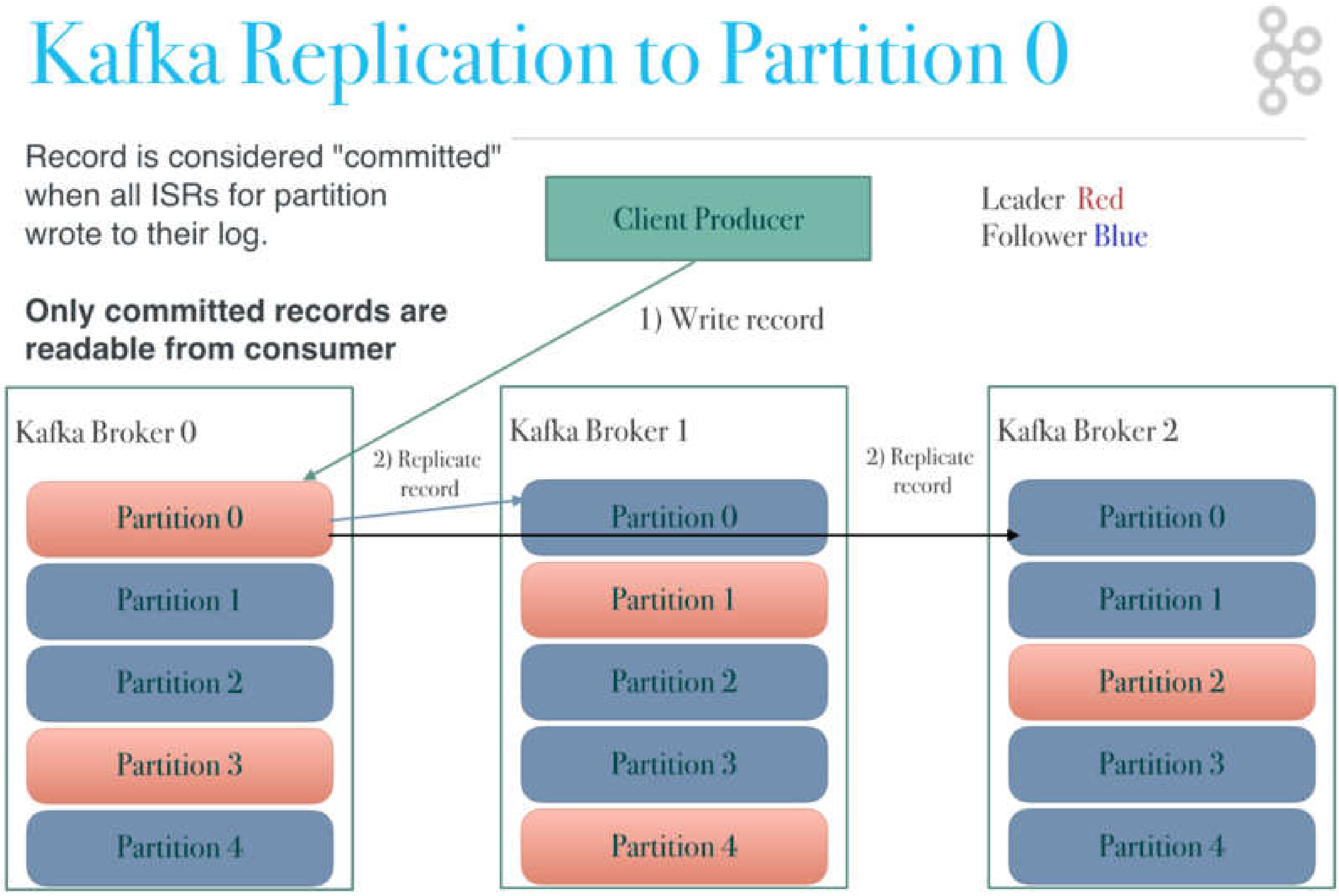
Kafka Topic Partition Replication

Kafka can replicate partitions across a configurable number of Kafka servers which is used for fault tolerance. Each partition has a leader server and zero or more follower servers. Leaders handle all read and write requests for a partition.

Followers replicate leaders and take over if the leader dies.

Kafka Partition Leaders, Followers and ISRs.

Kafka chooses one broker’ s partition’ s replicas as leader using ZooKeeper. The broker that has the partition leader handles all reads and writes of records for the partition. Kafka replicates writes to the leader partition to followers (node/partition pair). A follower that is in-sync is called an ISR (in-sync replica). If a partition leader fails, Kafka chooses a new ISR as the new leader.



Kafka Producers write cadence and partitioning of records

Producers write at their cadence so the order of Records cannot be guaranteed across partitions. Producers pick the partition such that Record/messages go to a given partition based on the data. For example, you could have all the events of a certain 'employeeId' go to the same partition. If order within a partition is not needed, a 'Round Robin' partition strategy can be used, so Records get evenly distributed across partitions.

Kafka Producer Load Balancing

The producer asks the Kafka broker for metadata about which Kafka broker has which topic partitions leaders thus no routing layer needed. This leadership data allows the producer to send records directly to Kafka broker partition leader.

Because Kafka is designed for broker scalability and performance, producers (rather than brokers) are responsible for choosing which partition each message is sent to. The default partition is determined by a hashing function on the message key, or round-robin in the absence of a key. However, this may not always provide the desired behaviour (e.g. message ordering, fair distribution of messages to consumers, etc). Producers can therefore send messages to specific partitions – through the use of a custom partitioner, or by using manual or hashing options available with the default partitioner.

Kafka Producer Record Batching

Kafka producers support record batching. Batching can be configured by the size of records in bytes in batch. Batches can be auto-flushed based on time.

Batching is good for network IO throughput.

Batching speeds up throughput drastically.

Buffering is configurable and lets you make a tradeoff between additional latency for better throughput. Or in the case of a heavily used system, it could be both better average throughput and reduces overall latency.

Producer Durability

The producer can specify durability level. The producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message.

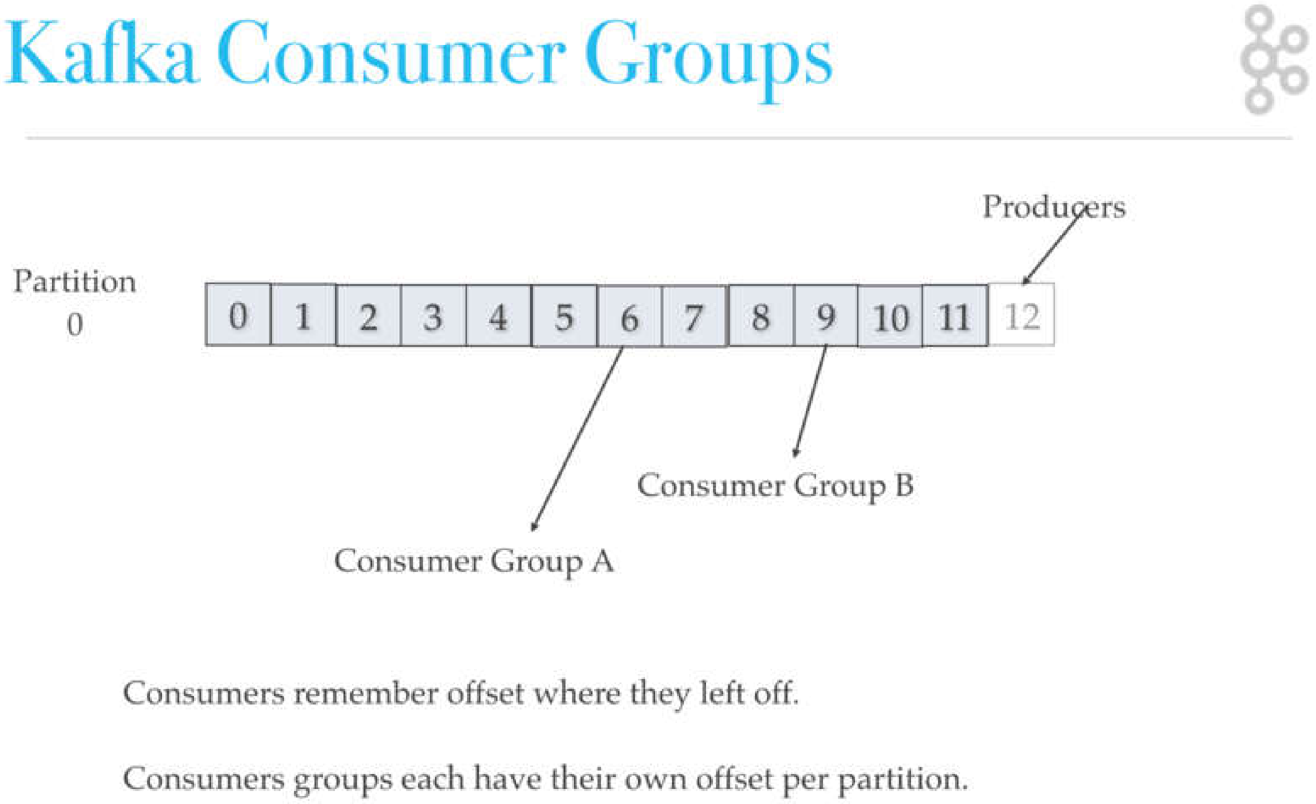
The producer can send with no acknowledgments (0). The producer can send with just get one acknowledgment from the partition leader (1). The producer can send and wait on acknowledgments from all replicas (-1), which is the default.

Improved Producer (June 2017 release)

Kafka now supports “exactly once” delivery from producer. They achieve this by the producer sending a sequence id, the broker keeps track if producer already sent this sequence, if producer tries to send it again, it gets an ack for duplicate message, but nothing is saved to log. This improvement requires no API change.

Kafka Consumer Groups

You group consumers into a consumer group by use case or function of the group. Consumer groups have names to identify them from other consumer groups. A consumer group has a unique id. Each consumer group is a subscriber to one or more Kafka topics. Each consumer group maintains its offset per topic partition. A record gets delivered to only one consumer in a consumer group. Each consumer in a consumer group processes records and only one consumer in that group will get the same record. Consumers in a consumer group load balance record processing.



Kafka Consumer Groups

Consumers remember offset where they left off.

Consumers groups each have their own offset per partition.

Kafka Consumer Load Share

Kafka consumer consumption divides partitions over consumer instances within a consumer group. Each consumer in the consumer group is an exclusive consumer of a "fair share" of partitions. This is how Kafka does load balancing of consumers in a consumer group. Consumer membership within a consumer group is handled by the Kafka protocol dynamically. If new consumers join a consumer group, it gets a share of partitions. If a consumer dies, its partitions are split among the remaining live consumers in the consumer group. This is how Kafka does fail over of consumers in a consumer group.

Kafka Consumer Failover

Consumers notify the Kafka broker when they have successfully processed a record, which advances the offset. If a consumer fails before sending commit offset to Kafka broker, then a different consumer can continue from the last committed offset. If a consumer fails after processing the record but before sending the commit to the broker, then some Kafka records could be reprocessed. In this scenario, Kafka implements the at least once behavior, and you should make sure the messages (record deliveries) are idempotent.

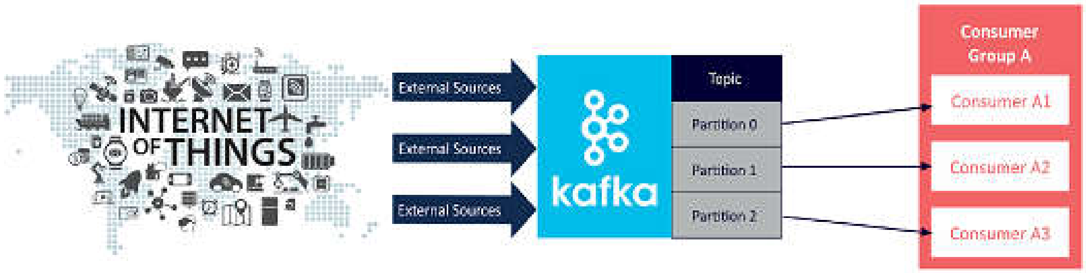
Consumer to Partition Cardinality - Load sharing redux

Only a single consumer from the same consumer group can access a single partition. If consumer group count exceeds the partition count, then the extra consumers remain idle. Kafka can use the idle consumers for failover. If there are more partitions than consumer group, then some consumers will read from more than one partition.

More explaintion about relationship between Consumer and Partition

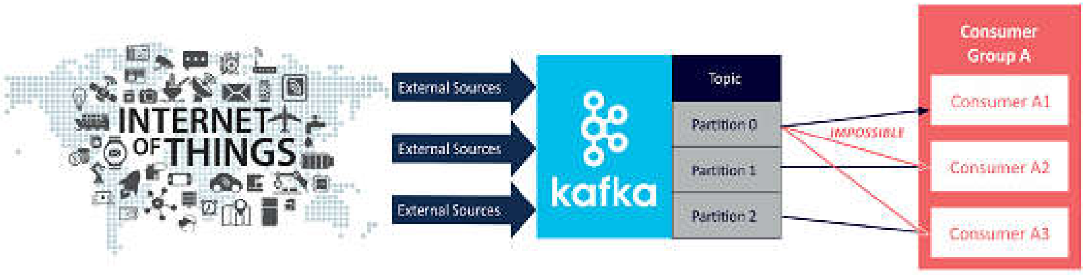
A partition can connect to at most 1 consumer per group:

Rule: A partition can connect to at most 1 consumer per group. 1-1partitions to consumer instances in a group



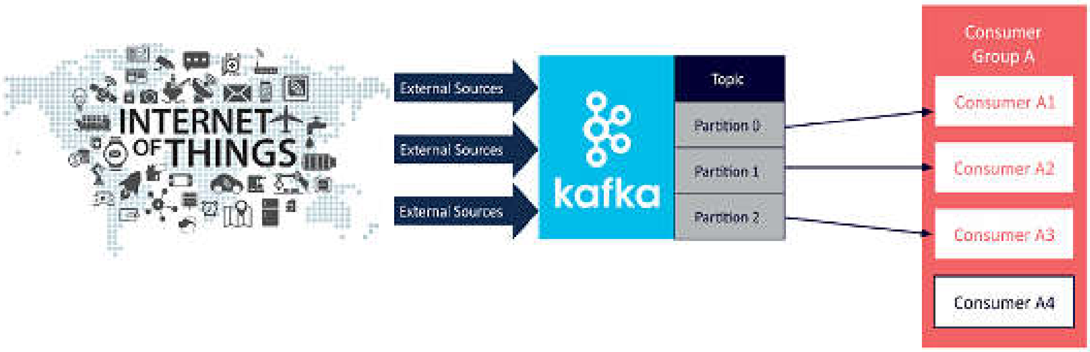
A partition cannot have more than one dynamically connected consumers in the same consumer group:

A partition cannot have more than one dynamically connected consumer in the same group

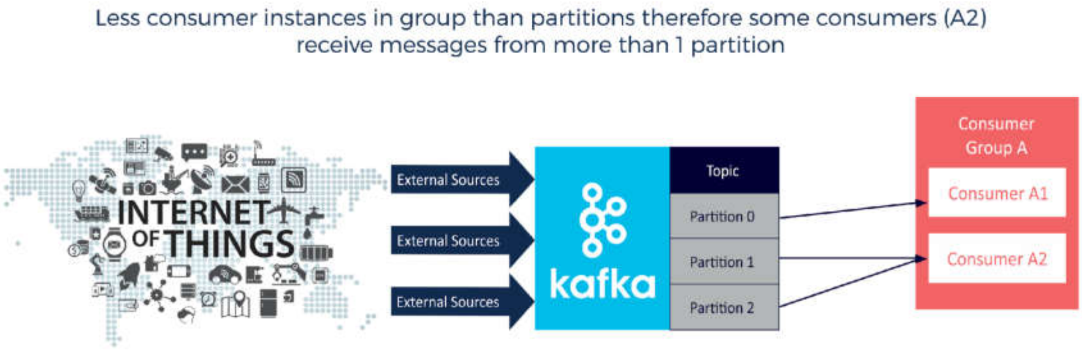


If consumer group count exceeds the partition count, then the extra consumers remain idle. Kafka is able to failover to such idle consumers in cases where an active consumer dies, or when a new partition is added:

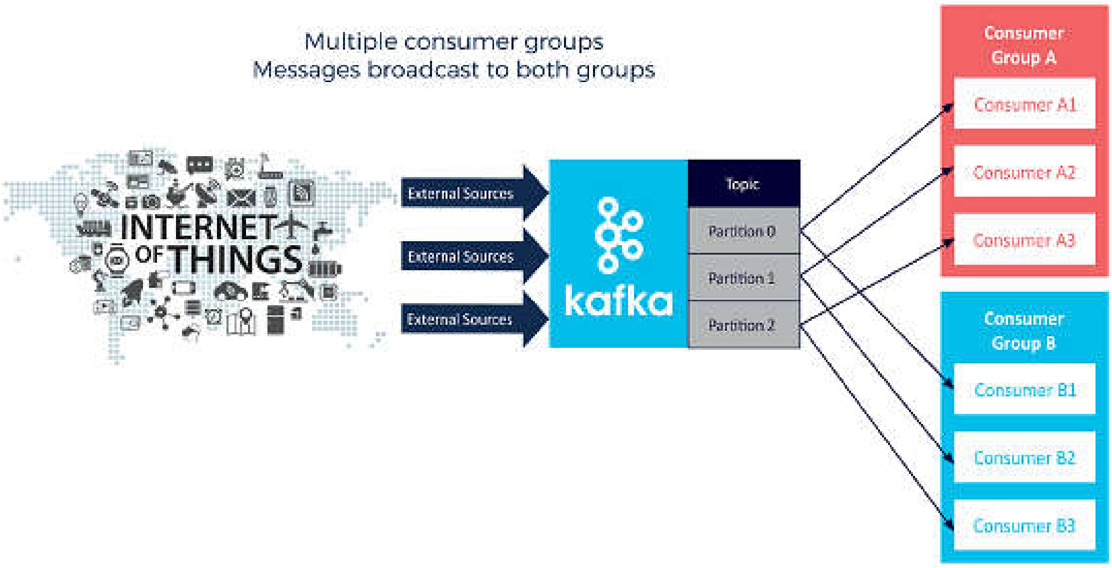
More consumer instances in group than partitions therefore idle consumer (A4)



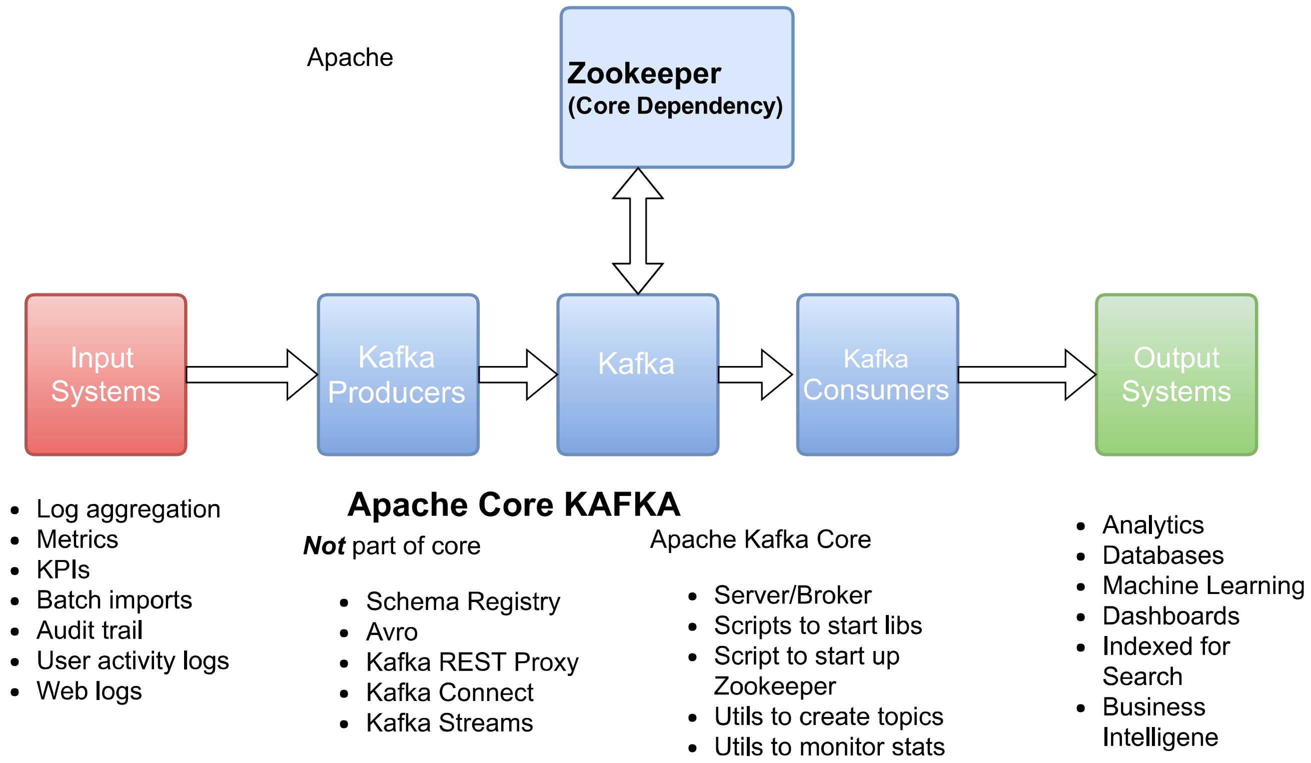
If there are more partitions than consumer group, then some consumers will read from more than one partition:



If there are multiple consumer groups subscribing the same topic, then every event from each partition of the topic gets broadcast to each group:



Core Kafka



- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs

Apache Core KAFKA

Not part of core

- Schema Registry
- Avro
- Kafka REST Proxy
- Kafka Connect
- Kafka Streams

Apache Kafka Core

- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

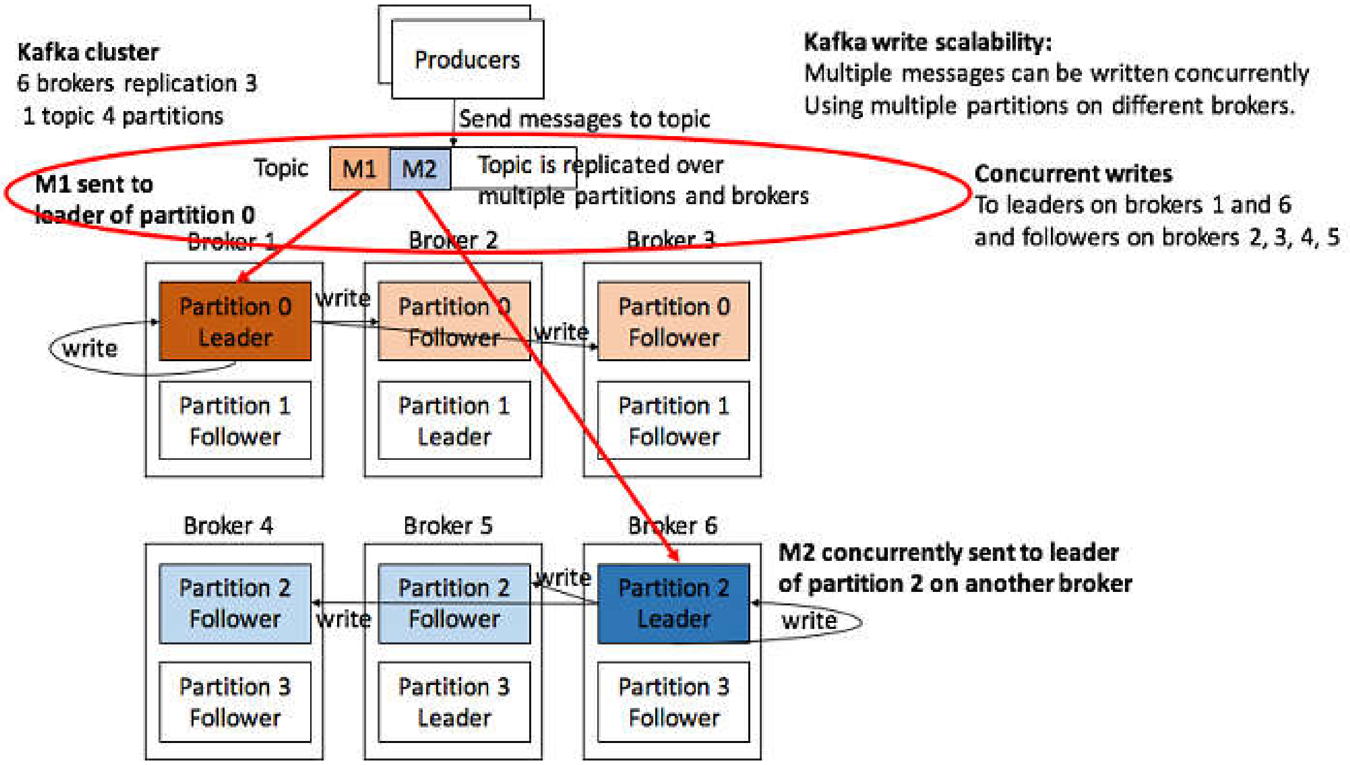
- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligene

Kafka needs ZooKeeper

- Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs.
- Kafka uses Zookeeper to manage service discovery for Kafka Brokers that form the cluster.
- Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc.
- Zookeeper provides an in-sync view of Kafka Cluster configuration.

Kafka Scale and Speed

How can Kafka scale if multiple producers and consumers read and write to same Kafka topic log at the same time? First Kafka is fast, Kafka writes to filesystem sequentially which is fast. On a modern fast drive, Kafka can easily write up to 700 MB or more bytes of data a second. Kafka scales writes and reads by sharding topic logs into partitions. Recall topics logs can be split into multiple partitions which can be stored on multiple different servers, and those servers can use multiple disks. Multiple producers can write to different partitions of the same topic. Multiple consumers from multiple consumer groups can read from different partitions efficiently.

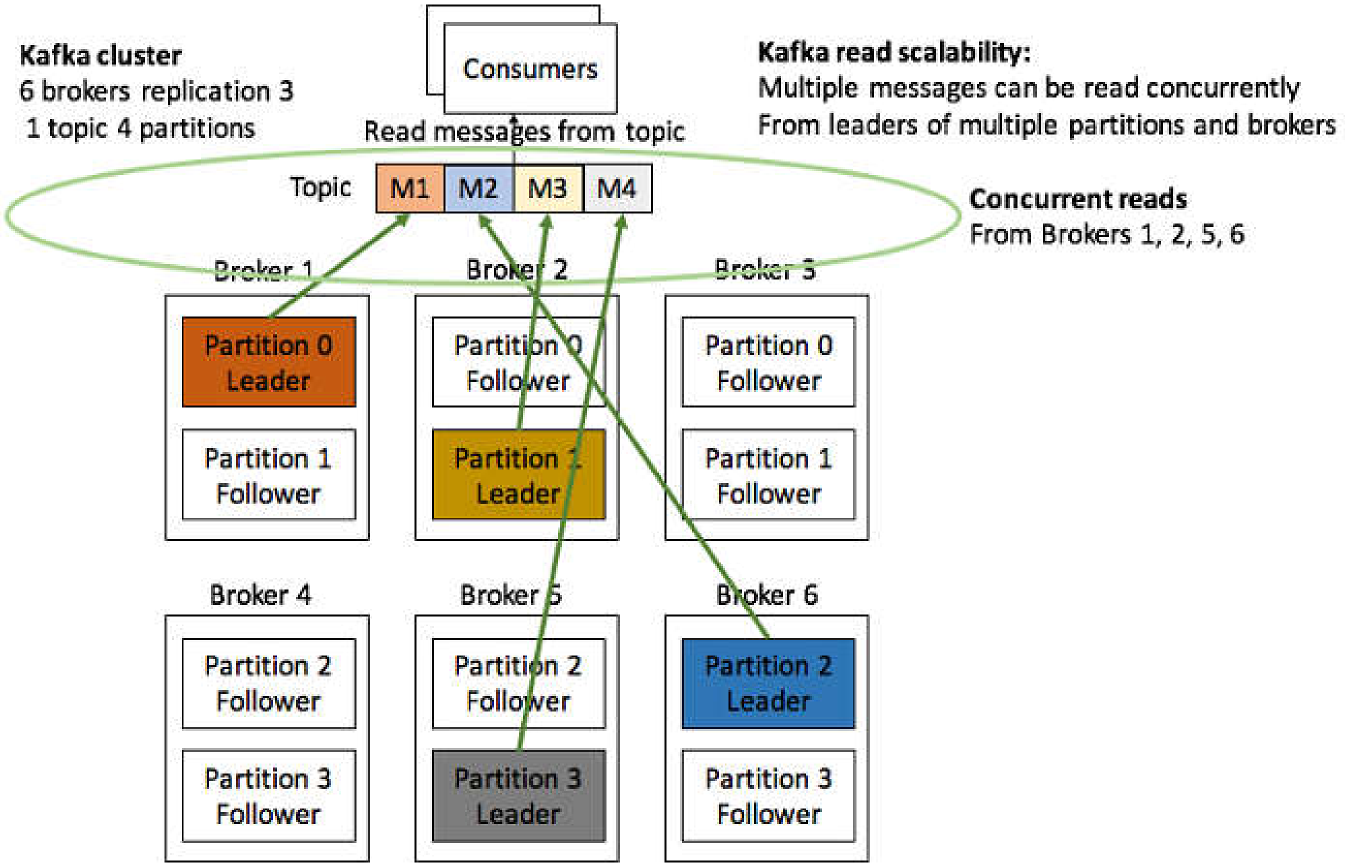

Kafka cluster
6 brokers replication 3
1 topic 4 partitions

Kafka write scalability:
Multiple messages can be written concurrently
Using multiple partitions on different brokers.

Concurrent writes
To leaders on brokers 1 and 6
and followers on brokers 2, 3, 4, 5

M1 sent to
leader of partition 0

M2 concurrently sent to leader
of partition 2 on another broker


Kafka cluster
6 brokers replication 3
1 topic 4 partitions

Kafka read scalability:
Multiple messages can be read concurrently
From leaders of multiple partitions and brokers

Concurrent reads
From Brokers 1, 2, 5, 6

Kafka Brokers

A Kafka cluster is made up of multiple Kafka Brokers. Each Kafka Broker has a unique ID (number). Kafka Brokers contain topic log partitions. Connecting to one broker bootstraps a client to the entire Kafka cluster. For failover, you want to start with at least three to five brokers. A Kafka cluster can have, 10, 100, or 1,000 brokers in a cluster if needed.

Kafka keeps track of which Kafka brokers are alive. To be alive, a Kafka Broker must maintain a ZooKeeper session using ZooKeeper’ s heartbeat mechanism, and must have all of its followers in-sync with the leaders and not fall too far behind.

Both the ZooKeeper session and being in-sync is needed for broker liveness which is referred to as being in-sync. An in-sync replica is called an ISR. Each leader keeps track of a set of “in sync replicas” .

If ISR/follower dies, falls behind, then the leader will remove the follower from the set of ISRs. Falling behind is when a replica is not in-sync after replica.lag.time.max.ms period.

A message is considered “committed” when all ISRs have applied the message to their log. Consumers only see committed messages. Kafka guarantee: committed message will not be lost, as long as there is at least one ISR.

Kafka Cluster, Failover, ISRs

Kafka supports replication to support failover. Recall that Kafka uses ZooKeeper to form Kafka Brokers into a cluster and each node in Kafka cluster is called a Kafka Broker. Topic partitions can be replicated across multiple nodes for failover. The topic should have a replication factor greater than 1 (2, or 3). For example, if you are running in AWS, you would want to be able to survive a single availability zone outage. If one Kafka Broker goes down, then the Kafka Broker which is an ISR (in-sync replica) can serve data.

Kafka Failover vs. Kafka Disaster Recovery

Kafka uses replication for failover. Replication of Kafka topic log partitions allows for failure of a rack or AWS availability zone (AZ). You need a replication factor of at least 3 to survive a single AZ failure. You need to use Mirror Maker, a Kafka utility that ships with Kafka core, for disaster recovery. Mirror Maker replicates a Kafka cluster to another data-center or AWS region. They call what Mirror Maker does mirroring as not to be confused with replication.

Kafka Consumer and Message Delivery Semantics

There are three message delivery semantics: at most once, at least once and exactly once. At most once is messages may be lost but are never redelivered. At least once is messages are never lost but may be redelivered. Exactly once is each message is delivered once and only once. Exactly once is preferred but more expensive, and requires more bookkeeping for the producer and consumer.

To implement “at-most-once” consumer reads a message, then saves its offset in the partition by sending it to the broker, and finally process the message. The issue with “at-most-once” is a consumer could die after saving its position but before processing the message. Then the consumer that takes over or gets restarted would leave off at the last position and message in question is never processed.

To implement “at-least-once” the consumer reads a message, process messages, and finally saves offset to the broker. The issue with “at-least-once” is a consumer could crash after processing a message but before saving last offset position. Then if the consumer is restarted or another consumer takes over, the consumer could receive the message that was already processed. The “at-least-once” is the most common set up for messaging, and it is your responsibility to make the messages idempotent, which means getting the same message twice will not cause a problem (two debits).

To implement “exactly once” on the consumer side, the consumer would need a two-phase commit between storage for the consumer position, and storage of the consumer’s message process output. Or, the consumer could store the message process output in the same location as the last offset.

Kafka offers the first two, and it up to you to implement the third from the consumer perspective.

Kafka and Quorum

Quorum is the number of acknowledgments required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap for availability. Most systems use a majority vote, Kafka does not use a simple majority vote to improve availability.

In Kafka, leaders are selected based on having a complete log. If we have a replication factor of 3, then at least two ISRs must be in-sync before the leader declares a sent message committed. If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages.

Among the followers there must be at least one replica that contains all committed messages. Problem with majority vote Quorum is it does not take many failures to have inoperable cluster.

Kafka Quorum Majority of ISRs

Kafka maintains a set of ISRs per leader. Only members in this set of ISRs are eligible for leadership election. What the producer writes to partition is not committed until all ISRs acknowledge the write. ISRs are persisted to ZooKeeper whenever ISR set changes. Only replicas that are members of ISR set are eligible to be elected leader.

This style of ISR quorum allows producers to keep working without the majority of all nodes, but only an ISR majority vote. This style of ISR quorum also allows a replica to rejoin ISR set and have its vote count, but it has to be fully re-synced before joining even if replica lost un-flushed data during its crash.

All nodes die at same time. Now what?

Kafka’s guarantee about data loss is only valid if at least one replica is in-sync.

If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid. If all replicas are down for a partition, Kafka, by default, chooses first replica (not necessarily in ISR set) that comes alive as the leader (config unclean.leader.election.enable=true is default). This choice favors availability to consistency.

If consistency is more important than availability for your use case, then you can set config unclean.leader.election.enable=false then if all replicas are down for a partition, Kafka waits for the first ISR member (not first replica) that comes alive to elect a new leader.

Producers pick Durability

Producers can choose durability by setting acks to - none (0), the leader only (1) or all replicas (-1).

The acks=all is the default. With all, the acks happen when all current in-sync replicas (ISRs) have received the message.

You can make the trade-off between consistency and availability. If durability over availability is preferred, then disable unclean leader election and specify a minimum ISR size.

The higher the minimum ISR size, the better the guarantee is for consistency. But the higher minimum ISR, the more you reduces availability since partition won’t be unavailable for writes if the size of ISR set is less than the minimum threshold.

Quotas

Kafka has quotas for consumers and producers to limits bandwidth they are allowed to consume. These quotas prevent consumers or producers from hogging up all the Kafka broker resources. The quota is by client id or user. The quota data is stored in ZooKeeper, so changes do not necessitate restarting Kafka brokers.

How Kafka’s Storage Internals Work

Kafka’s storage unit is a partition

A partition is an ordered, immutable sequence of messages that are appended to. A partition cannot be split across multiple brokers or even multiple disks.

The retention policy governs how Kafka retains messages

You specify how much data or how long data should be retained, after which Kafka purges messages in-order—regardless of whether the message has been consumed.

Partitions are split into segments

So Kafka needs to regularly find the messages on disk that need purged. With a single very long file of a partition’s messages, this operation is slow and error prone. To fix that (and other problems we’ll see), the partition is split into segments. When Kafka writes to a partition, it writes to a segment - the active segment. If the segment’s size limit is reached, a new segment is opened and that becomes the new active segment. Segments are named by their base offset. The base offset of a segment is an offset greater than offsets in previous segments and less than or equal to offsets in that segment.

On disk, a partition is a directory and each segment is an index file and a log file.

```
$ tree kafka | head -n 6
kafka
├── events-1
│ ├── 00000000003064504069.index
│ ├── 00000000003064504069.log
│ ├── 00000000003065011416.index
│ ├── 00000000003065011416.log
```

Segment logs are where messages are stored

Each message is its value, offset, timestamp, key, message size, compression codec, checksum, and version of the message format.

Segment indexes map message offsets to their position in the log

The segment index maps offsets to their message’ s position in the segment log.

The index file is memory mapped, and the offset look up uses binary search to find the nearest offset less than or equal to the target offset.

The index file is made up of 8 byte entries, 4 bytes to store the offset relative to the base offset and 4 bytes to store the position. The offset is relative to the base offset so that only 4 bytes is needed to store the offset. For example: let’ s say the base offset is 10000000000000000000, rather than having to store subsequent offsets 10000000000000000001 and 10000000000000000002 they are just 1 and 2.

Kafka Streams - Kafka Streams for Stream Processing

Kafka Streams is a library for developing distributed applications for processing record streams with Apache Kafka as the data storage for input and output records (with keys and values).

The Kafka Stream API builds on core Kafka primitives and has a life of its own. Kafka Streams enables real-time processing of streams. Kafka Streams supports stream processors. A stream processor takes continual streams of records from input topics, performs some processing, transformation, aggregation on input, and produces one or more output streams. For example, a video player application might take an input stream of events of videos watched, and videos paused, and output a stream of user preferences and then gear new video recommendations based on recent user activity or aggregate activity of many users to see what new videos are hot. Kafka Stream API solves hard problems with out of order records, aggregating across multiple streams, joining data from multiple streams, allowing for stateful computations, and more.

Kafka Connect

Kafka Connect is the connector API to create reusable producers and consumers (e.g., stream of changes from DynamoDB). Kafka Connect Sources are sources of records. Kafka Connect Sinks are a destination for records.

kafka相关文章

Kafka tutorial

Developing a Deeper Understanding of Apache Kafka Architecture

distributed streaming platform

How Kafka’s Storage Internals Work

Here’s what makes Apache Kafka so fast

kafka中文教程

kafka存储机制

kafka系列：kafka基本架构

kafka 学习 非常详细的经典教程

Kafka史上最详细原理总结