Apache Kafka Concepts

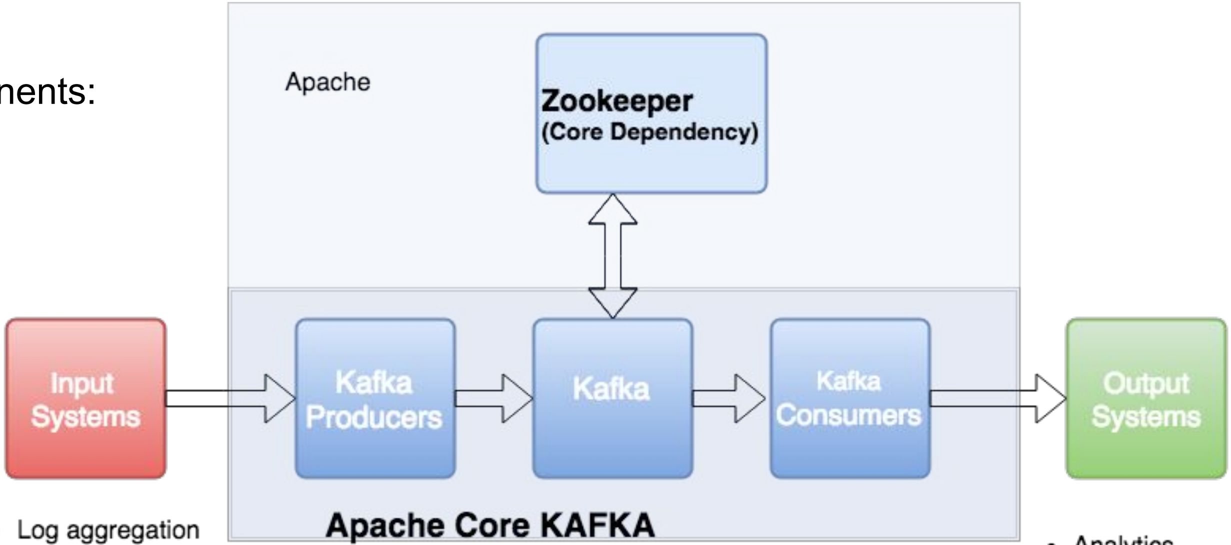
Kafka Architecture

Apache Kafka basic components:
(vanilla Kafka installation)

- Zookeeper
- Brokers
- Producer API
- Consumer API

Others:

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



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

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

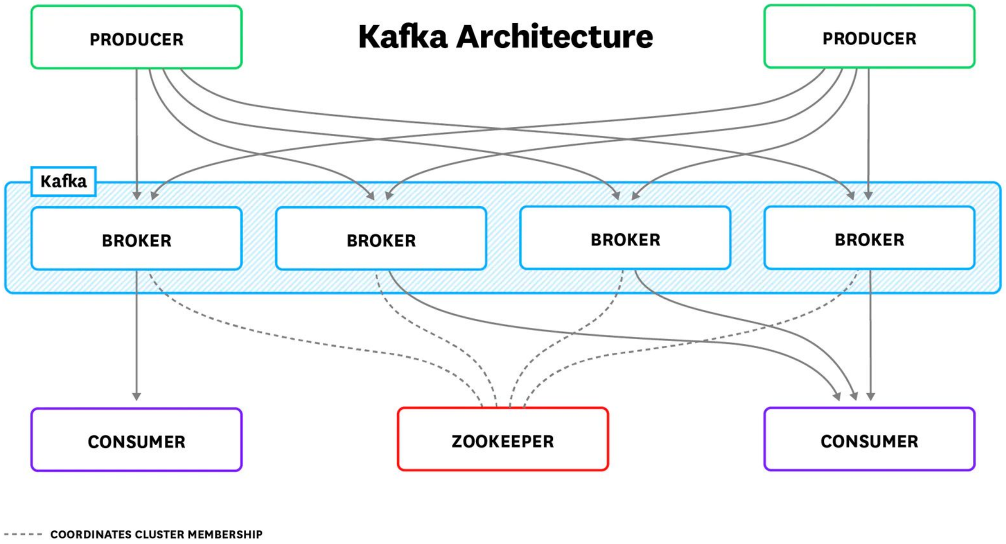
Kafka Architecture

Apache Kafka basic components:
(vanilla Kafka installation)

- Zookeeper
- Brokers
- Producer API
- Consumer API

Others:

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



Ref: https://www.datadoghq.com/blog/monitoring-kafka-performance-metrics/

Apache Kafka main concepts

- **Topics**
    - Messages
    - Partitions
    - Offsets
    - Keys
- **Producers**
- **Consumers**
- **Brokers**

Kafka concepts: messages

Message in Kafka can be anything that can be translated into a ByteArray (Kafka transports bytes);

Sensors information - sensors_info

Each message will contain the sensor_id + recordings at that moment

Tracks gps positions - tracks_positions (position of all the tracks)

Each message will contain the trackid + position

Recommended size: max 1MB (message.max.bytes)

Message in Kafka can have optionally as well a **key**. A key is sent if you need message ordering for a specific key/field (you want to see all temperatures recorded by station_id 123 altogether).

Messages in Kafka are organized in topics.

Kafka concepts: Topics, partitions, offsets

Topics: a particular stream of data/messages

- Similar to table concept (in a database), minus the metadata part
- Each topic has (**name, replication factor, number of partitions)**

Partitions: topics are split/saved on disk in partitions (the way to save data in a distributed system)

- **a partition is a single log, which resides on a single disk on a single machine (it may be replicated)**
- **Inside each partition the messages are ordered (note: the order of messages across multiple partitions is undefined)**
- In each partition each message gets an unique id (incremental) called offset
  - Partition 0: 0,1,2,3,4,5,6,7,8,9,10
  - Partition 1: 0,1,2,3
  - Partition 3: 0,1,2,3,4,5,6
- **Offsets**: unique per topic + partition (each partition has its own set of offsets). **One message is uniquely identified by (topic, partition, offset).** Order is guaranteed only inside a partition (offset 8 is written after offset 7 in Partition 0).

Kafka in a nutshell

Apache Kafka is a streaming platform based on a distributed publish/subscribe pattern.

Processes called **producers send messages into topics,** which are **managed and stored by a cluster of brokers**. Then, **processes called consumers subscribe to these topics for fetching and processing published messages**.

A **topic is distributed** across a number of brokers so that each broker manages subsets of messages for each topic - these subsets are called **partitions**. The number of partitions is defined when a topic is created and can be increased over time (but be careful with that operation).

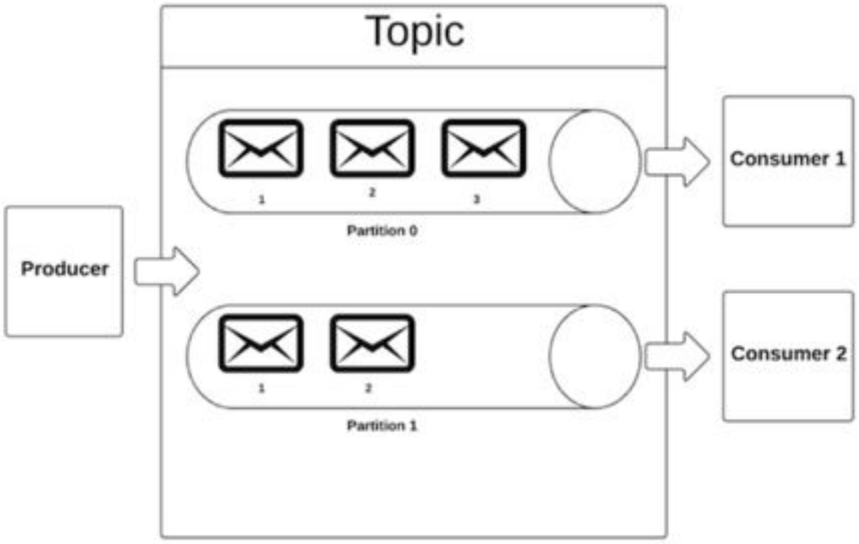
Topics and partitions (data distribution)

Kafka is a distributed system thus all topics will need to be stored on several machines and with a RF>1. This is done with partitions. Topics in Kafka can be subdivided into partitions. **A partition is an ordered, immutable record sequence. Kafka continually appends to partitions using the partition as a structured commit log.**

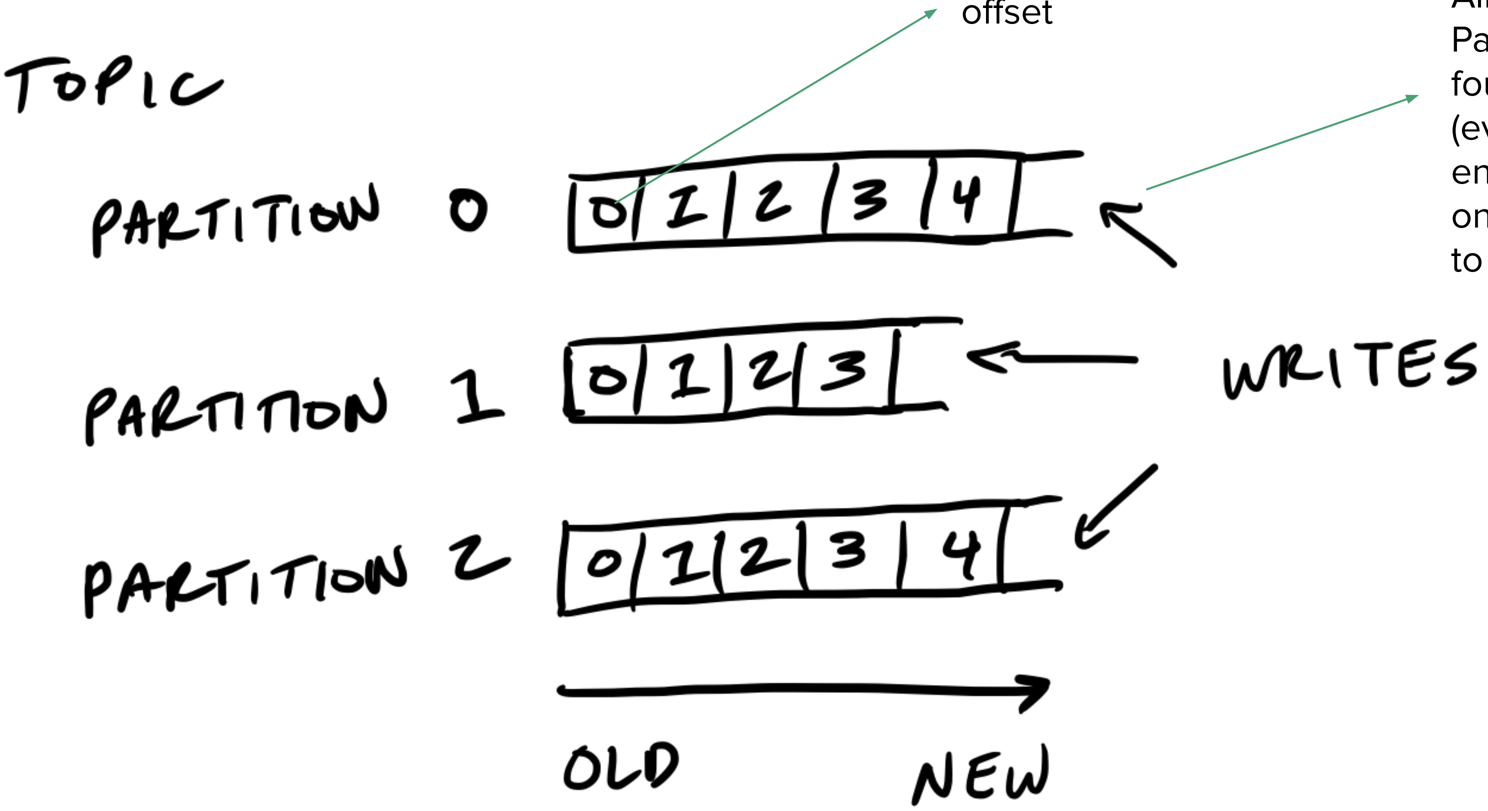
Partitions are the way to parallelize data in the cluster.

Topics data is saved on disk through partitions.

Partitions are the atom of storage in Kafka, a partition being a single log, which resides on a single disk on a single machine (it may be replicated).



Topics and partitions (data distribution)



offset

All the data from a Partition will be found on 1 node (even though the entire data might be on other nodes due to RF).

Partitioning of topics data

When data is sent towards Kafka Clusters, the partitioner balances data and requests load over brokers. It serves as a way to divvy up processing among consumer processes while **preserving order within the partition**. We call this **semantic partitioning**.

Semantic partitioning means using some key in the message to assign messages to partitions. For example if you were processing a click message stream you might want to partition the stream by the user id so that all data for a particular user would go to a single consumer. To accomplish this the client can take a key associated with the message and use some hash of this key to choose the partition to which to deliver the message.

Partitioning Strategies

Own Strategy (Client side): you can send the data to a specific partition (partition id is specified);

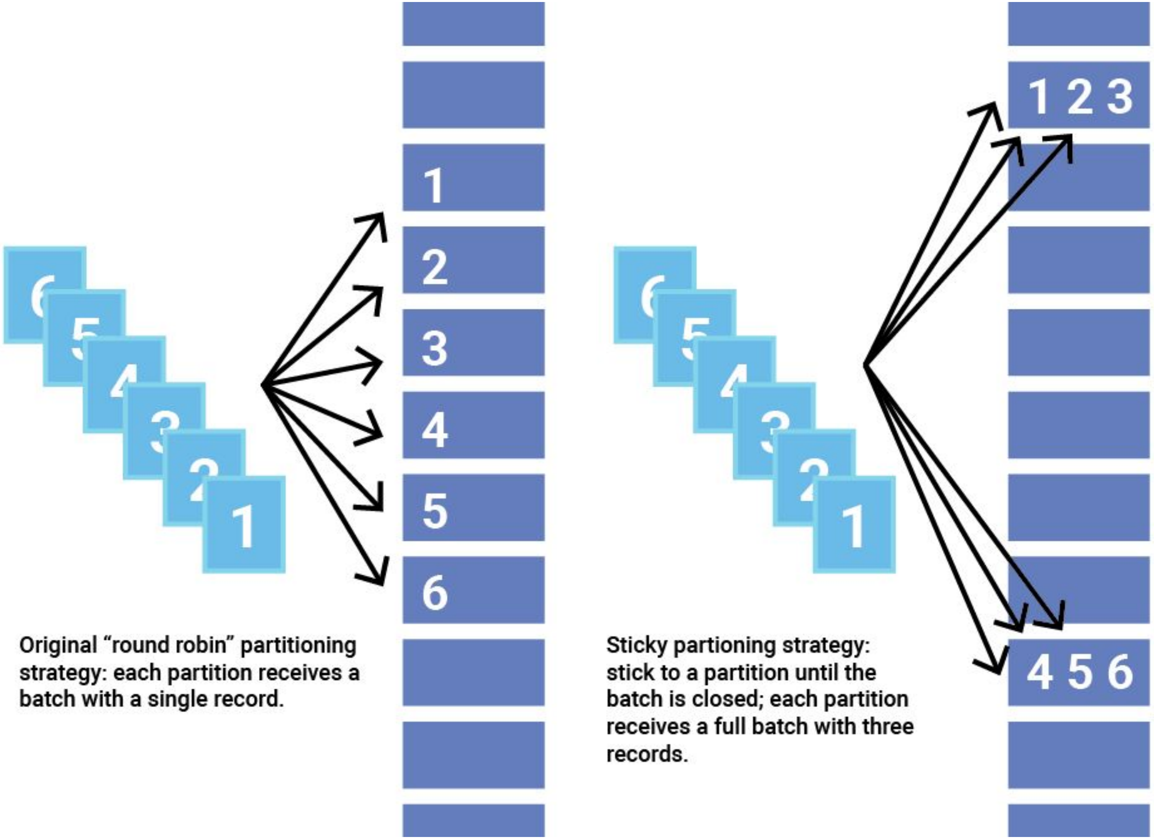
Random: Round Robin - no key messages;

Sticky Partitioner (default) - no key messages

Messages with Keys: All messages belonging to the same key will be placed in the same partition. When keys are present, each key messages are stored in the same partition (murmur hash)

One important property of Kafka is that the order of messages within a single partition is guaranteed, and the order of messages across multiple partitions is undefined. If you need to make sure a set of messages arrive in order, relative to each other, you need to make sure they're all routed to the same partition.

Partitioning Strategies: Random vs Sticky



Original “round robin” partitioning strategy: each partition receives a batch with a single record.

Sticky partioning strategy: stick to a partition until the batch is closed; each partition receives a full batch with three records.

Ref: https://www.confluent.io/blog/apache-kafka-producer-improvements-sticky-partitioner/

About partitions

Topic creation - we need to specify number of partitions.

The number of partitions/topic can be increased (but not really recommended), but cannot be decreased. If you need to decrease the number of partitions for a topic, you need to create a new topic with the desired number of partitions and re-insert the data.

As guideline for optimal performance, you should not have more than 4000 partitions per broker and not more than 200,000 partitions in a cluster (in reality recommended about 20k partitions/cluster).

The order of messages within a single partition is guaranteed, and the order of messages across multiple partitions is undefined.

How many partitions - no strict answer

Topic size? 5TB/day => 58MB/s

Conservative value: 10MB/s / partition => 6 partitions

Partition size? Aprox 0.8TB/day - 7 day retention => 5.6TB partition size

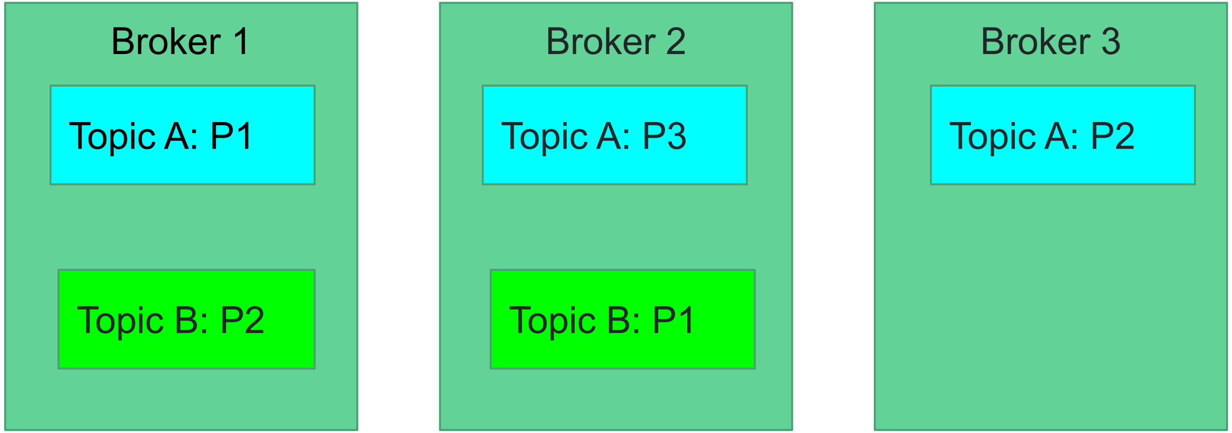
Number of brokers: if smaller than 6 => double number of partitions

Number of consumers/group: if at peak time you need 20 consumers for processing data => 20 partitions

Number of producers: more partitions than producers (high producers? => 3 X partitions)

Kafka Concepts: Brokers

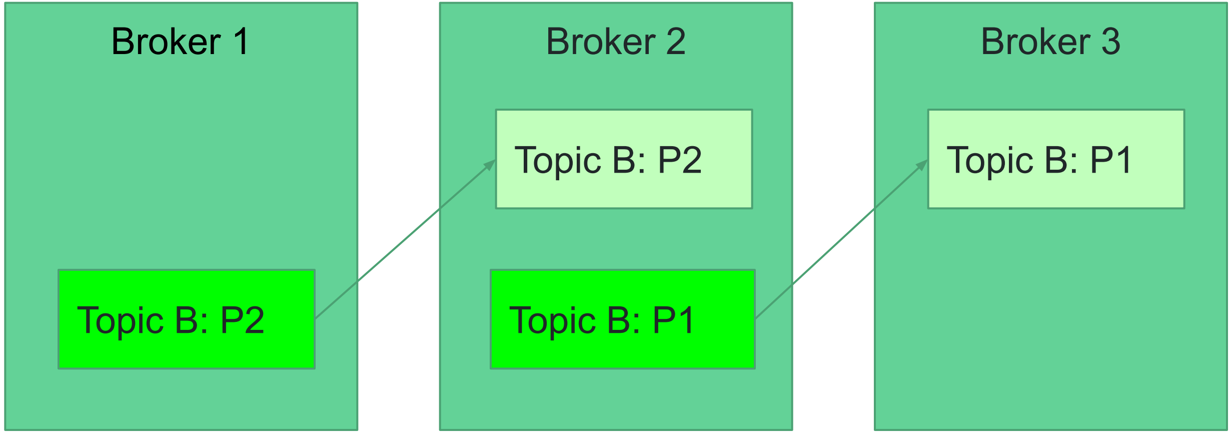
- A **Kafka cluster is composed of multiple brokers (hold the topics data - the partitions)**
- Each broker has an id
- Only parts of the topics data are present in a broker (through partitions)
- **Once you connect to a broker you're connected to the entire cluster (cluster metadata is available)**
- Example: Topic A: 3 partitions, Topic B: 2 partitions and a cluster with 3 brokers



Kafka Concepts: Brokers - topics replication

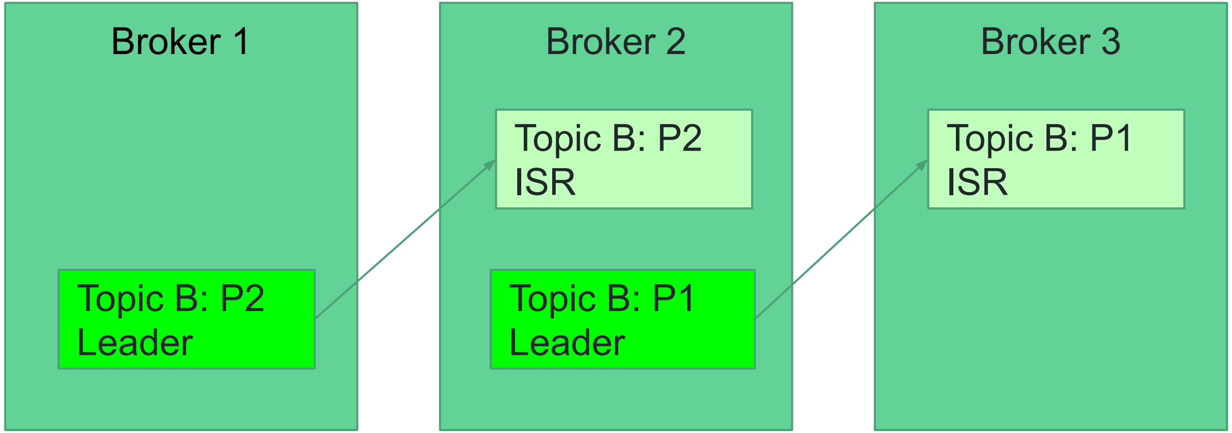
Each topic is identified by (name, **replication factor**, number of partitions)

- Example: 3 brokers, Topic B: 2 partitions, Replication 2



Kafka Concepts: Leader of a partition

- **Leader - Follower replica**
- One broker can be the only leader at one time for a given partition. **Only this leader can write and read data for that partition (for both producers and consumers).**
- The followers will sync their data with the leader data (through timely fetch requests)
- **One partition has 1 leader and multiple ISR (insync replicas). Please note the leader is part of ISR list.**



Partitions allocation to brokers - allocation goals

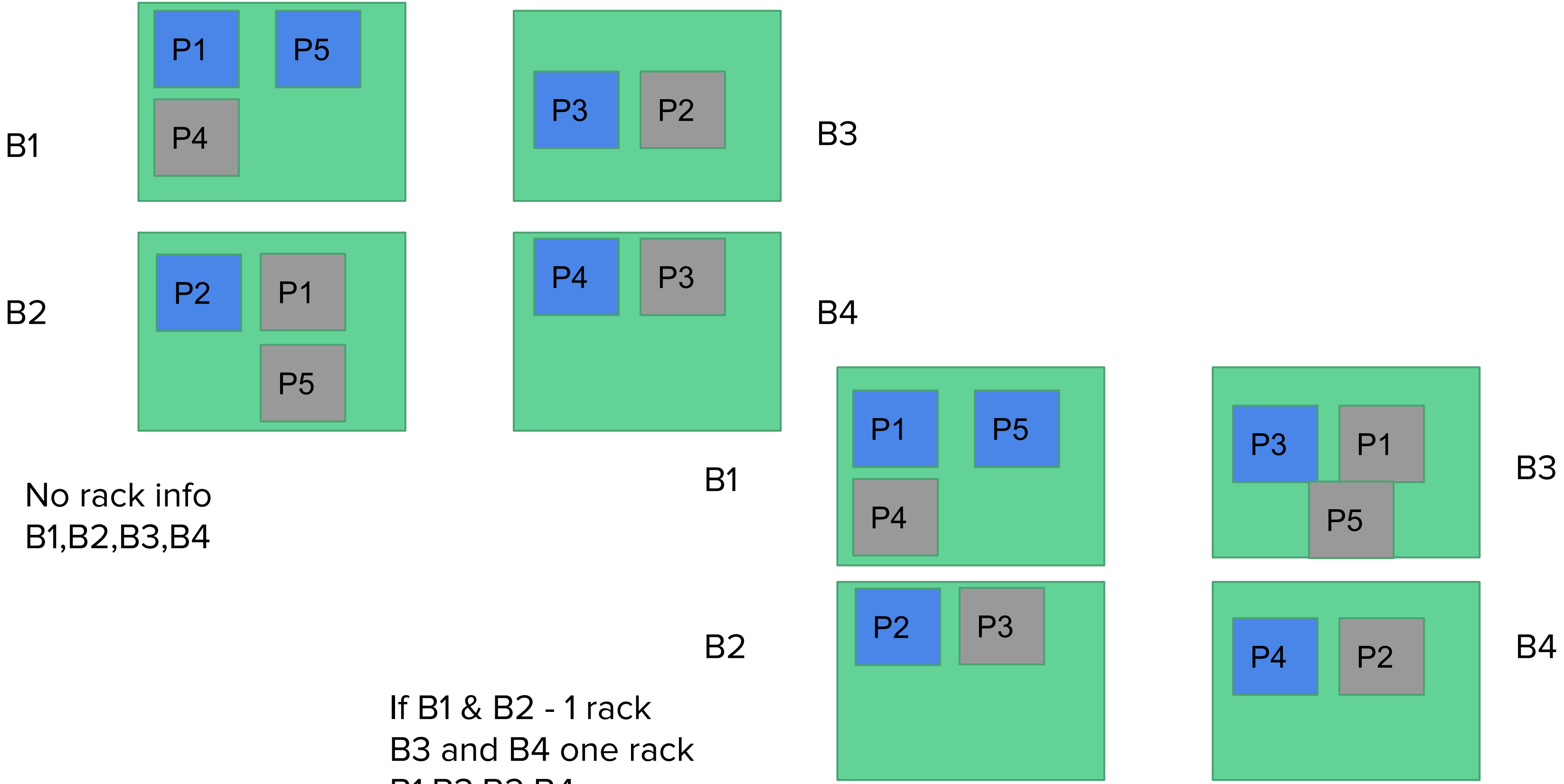
Whenever a new topic is created, Kafka runs it's leader election algorithm to figure out the preferred leader of a partition.

Create Topic (name, partitions nr, replication factor) => Kafka immediately will assign partitions replicas in the Kafka Cluster - on brokers.

Leader election algorithm goals:

- **To spread the replicas evenly among brokers** (e.g. 6 brokers, 10 partitions/topic, RF 3 => avg 5 partitions/broker)
- **To make sure that for each partition each replica is on a different broker** (e.g. if partition 0 has the leader on broker 2, the followers can be placed on brokers 3 and 4 - but not on 2 and not both on 3)
- **If the broker has rack information (0.10.0.0 onwards), then assign the replicas for each partition to different racks.**

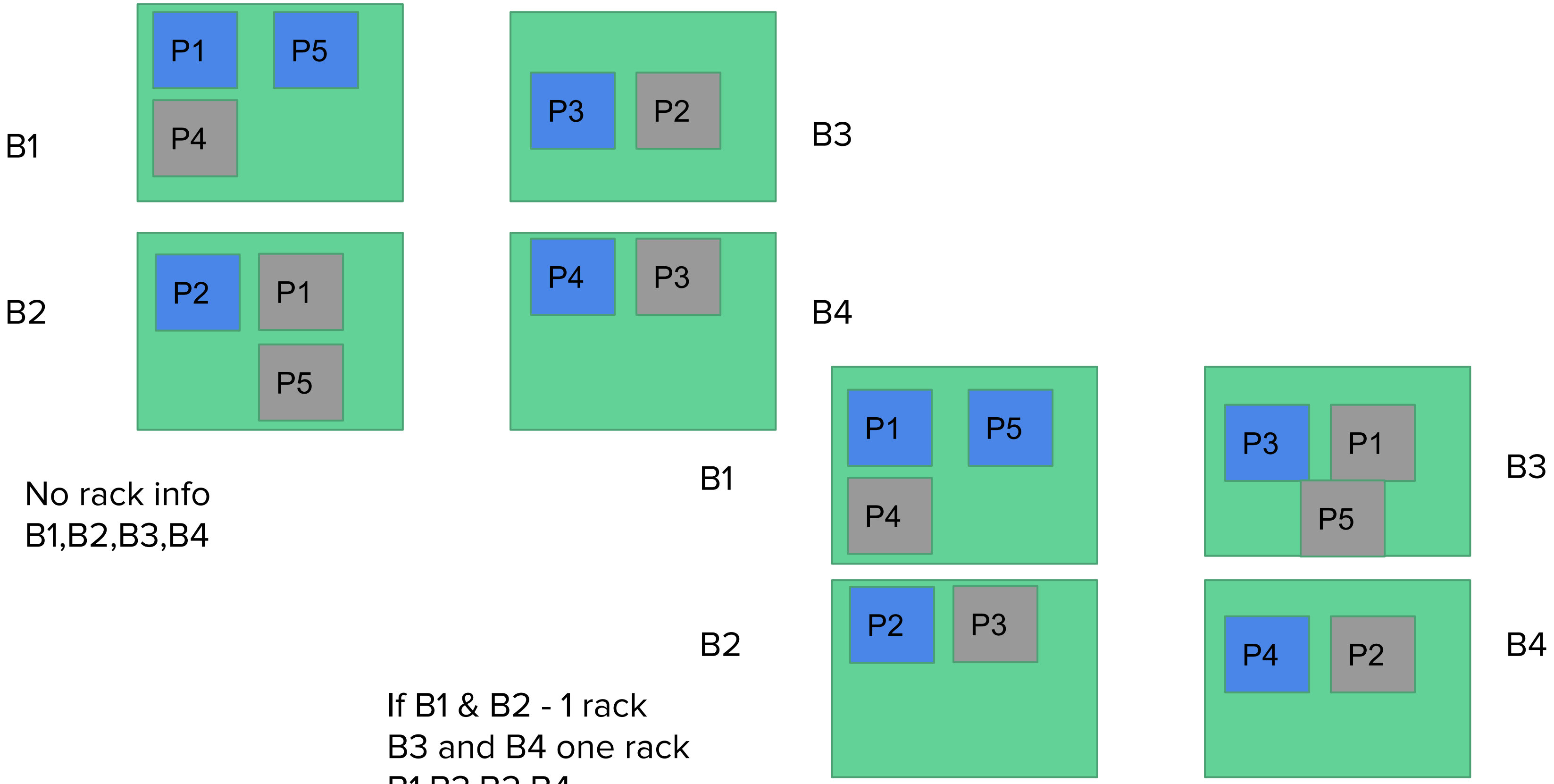
Topic, 5 partitions, 4 brokers, RF 2



No rack info
B1,B2,B3,B4

If B1 & B2 - 1 rack
B3 and B4 one rack
B1,B3,B2,B4

Initial placement of the partitions: preferred partition leaders



No rack info
B1,B2,B3,B4

If B1 & B2 - 1 rack
B3 and B4 one rack
B1,B3,B2,B4

Kafka Concepts: Replication as Leader-Follower

- One broker can be the only leader at one time for a given partition. **Only this leader can write and read data for that partition (for producers and consumers). Note: in Kafka 2.4 consumers can fetch data as well from closest replica.**
- The followers will sync the data with the leader (send a fetch and leader will push the requested data)
- Each partition has a leader and a number of followers replicas

InSync Replicas

- **The leader maintains a set of in-sync replicas (ISR): the set of replicas that have fully caught up with the leader;**
- **the leader can be chosen only from ISR's - for clean leader selection;**
- **For each partition, we store in Zookeeper the current ISR and the current leader.**
- **Please note: P1 has B1 as leader and ISR B1,B2,B3 (RF 3). The leader is part of ISR.**

Kafka Concepts: ISR’s

The criteria for considering a partition follower in-sync with its leader is the following:

- It has **fetched messages from the partition leader in the last X seconds.** (configurable through replica.lag.time.max.ms). It is not enough to fetch any messages—the **fetch request must have requested all messages up to the leader log’s end offset.** This ensures that it is as in-sync as possible.
- It has sent a heartbeat to Zookeeper in the last X seconds. (configurable through zookeeper.session.timeout.ms)

For an ISR out of sync **Kafka does not require that crashed nodes recover with all their data intact:**

- allowing a replica to rejoin the ISR ensures that - before being ISR again - it must fully re-sync (with the leader);

Kafka Concepts: Controller

Controller Broker (KafkaController) is a Kafka service that runs on every broker in a Kafka cluster, but only one can be active (elected) at any point in time. In a Kafka cluster, one of the brokers serves as the controller, which is responsible for **managing the states of partitions and replicas** and for performing administrative tasks like re-assigning partitions.

A Controller is the broker that reacts to the event of another broker failing. It gets notified from a ZooKeeper Watch. A ZooKeeper Watch is basically a subscription to some data in ZooKeeper. When said data changes, ZooKeeper will notify everybody who is subscribed to it. ZooKeeper watches are crucial to Kafka—they serve as input for the Controller. The controller gets notified of this and acts upon it. It decides which nodes should become the new leaders for the affected partitions. It then informs every associated broker that it should either become a leader or start replicating from the new leader via a LeaderAndIsr request.

Unclean leader election

Kafka's guarantee with respect to data loss is predicated on at least one replica remaining in sync. What if all replicas fail?

There are two behaviors that could be implemented:

Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data) - default from Kafka 0.11.0.0;

Choose the first replica (not necessarily in the ISR) that comes back to life as the leader. Set unclean.leader.election.enable = true;

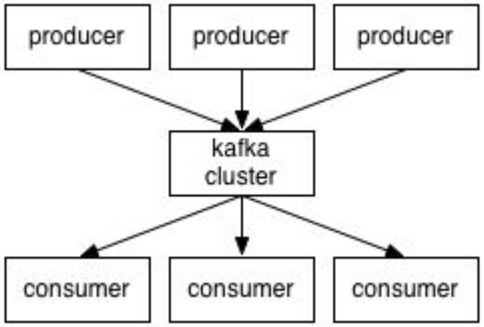
Let’s create some topics

See Kafka Exercises - document.

Producers

Kafka concepts: Producers

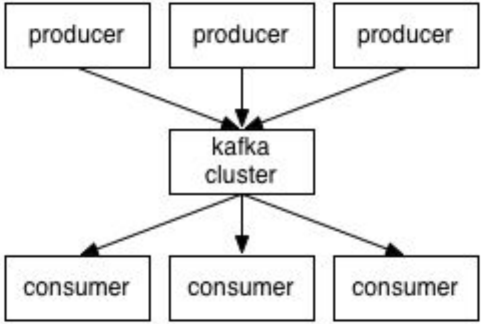
Producers are client applications that push/send messages into a Kafka topic. Any application that wants to send data to Kafka needs to embed the Producer code. Producer serializes the key and the message value => ByteArrays



A producer partitioner maps each message to a topic partition, and the producer sends a request to the leader of that partition. **The partitioners shipped with Kafka guarantee that all messages with the same non-empty key will be sent to the same partition.**

Kafka concepts: Producers

Producers are client applications that push/send messages into a Kafka topic. Any application that wants to send data to Kafka needs to embed the Producer code. Producer serializes the key and the message value => ByteArrays



- Producer Console API
- **Java Producer API**
- Languages: **Scala, Python (Kafka-Python, PyKafka, Confluent Python Kafka)**
- **Parameters for ProducerRecord (topic,broker,key(optional),partition_id(optional))**

Kafka Concepts: How producers write data

Producer serializes the key and the value objects => ByteArrays => **Partitioner** => **Batches** => Kafka Cluster

(all in blue above is part of the Producer Architecture)

Partitioner

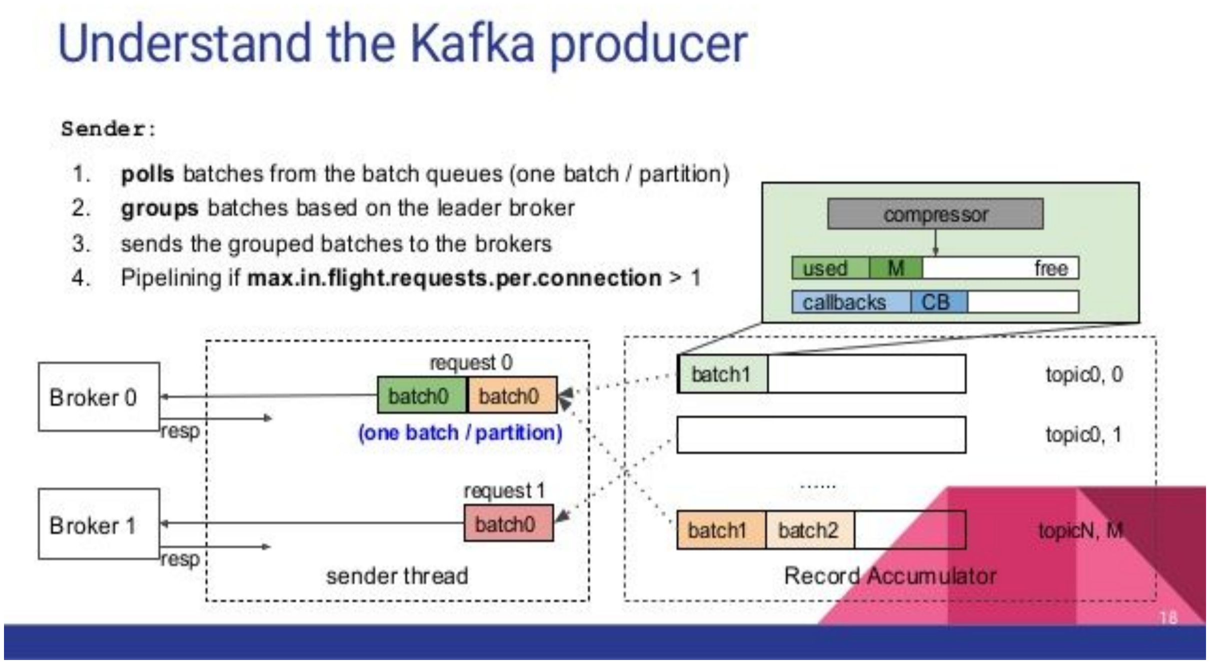
- If we specified a partition in the ProducerRecord , the partitioner doesn’t do anything and simply returns the partition we specified;
- If we specified a key in the ProducerRecord, the partition API uses the key and the number of available broker partitions to return a partition id. This id is used as an index into a sorted list of broker_ids and partitions to pick a broker partition for the producer request.
- If we didn’t specified a key (key is null), the partitioner will choose a partition for us, round robin manner.
- A custom partitioning strategy can also be plugged in using the partitioner.class config parameter.

Kafka Concepts: How producers write data

Partitioner adds the record to a **batch of records that will be send to same topic and partition.**

Batches are sent to Kafka Brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a RecordMetadata object with the **topic, partition, and the offset of the record within the partition.** If the broker failed to write the messages, it will return an error.



Understand the Kafka producer

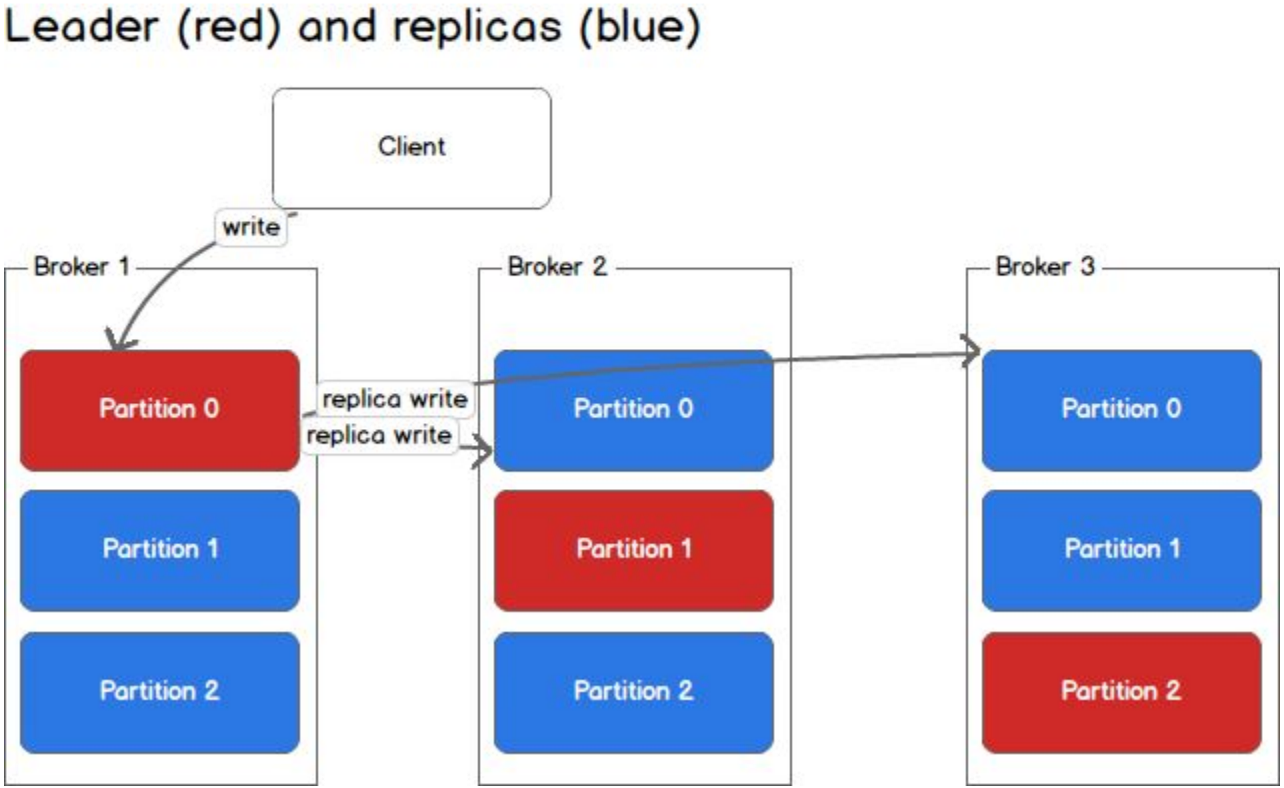
`Sender:`

1. **polls** batches from the batch queues (one batch / partition)
2. **groups** batches based on the leader broker
3. sends the grouped batches to the brokers
4. Pipelining if **max.in.flight.requests.per.connection** > 1

https://www.slideshare.net/JiangjieQin/producer-performance-tuning-for-apache-kafka-63147600

Kafka Concepts: How producers write data?

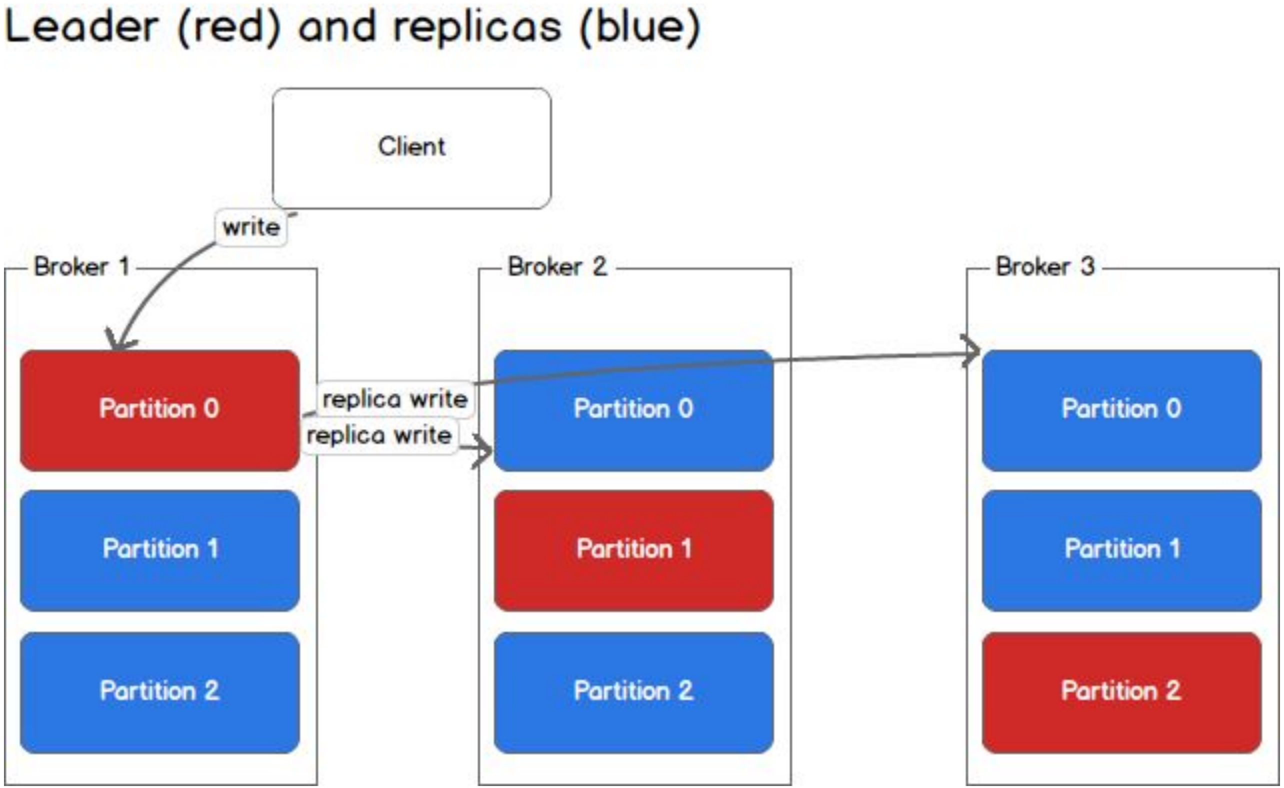
- Producers write data to topics
- **Producers know which broker and partition to write to**
- **Write is done only in the leader replica broker**
- In case of leader broker failure one of the brokers in the ISR will take the lead



Kafka Concepts: How producers write data?

Producers can choose to receive ack of data writes

- ack=0 - will not wait for ack
- ack = 1(default), will wait for leader ack
- ack = all, all ISR acknowledge the receipt of data



Kafka Concepts: How producers write data?

Acks = 0: the producer does have no idea how much data actually reached the server once it finished: it could be buffered in the client, in flight over the wire, or in the server but not yet on disk (in memory not sync to the disk yet).

Acks=1: tells the broker that the producers wants an acknowledgement once the leader partition has written the message to the partition data file (but not necessarily synced to the disk).

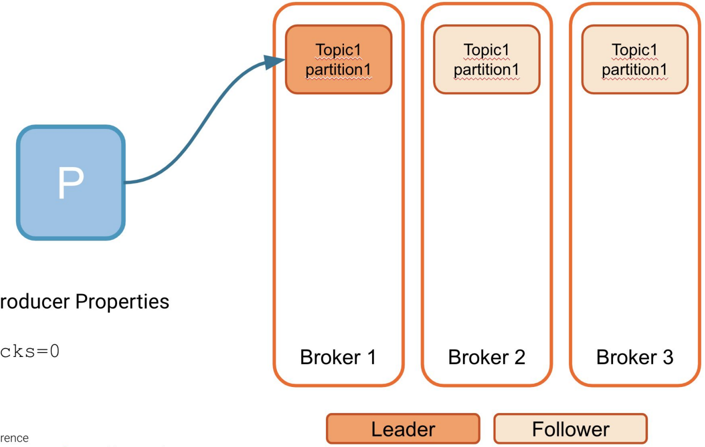
Acks = all: When a producer sets acks to "all" (or "-1"), acknowledgement happens as soon as all the current in-sync replicas have received the message. Note that "ack by all replicas" does not guarantee that the full set of assigned replicas have received the message, only the ISR's. **Min.insync.replicas** specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).

Writing in Kafka

There are two settings here that affect the producer:

acks - this is a producer-level setting // The acks property determines how you want to handle writing to kafka

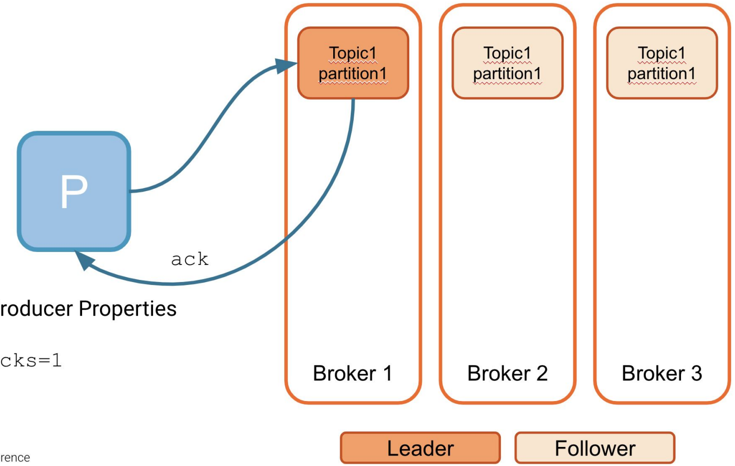
min.insync.replicas - this is a topic-level setting



Producer Properties

```
acks=0
```

Reference



Producer Properties

```
acks=1
```

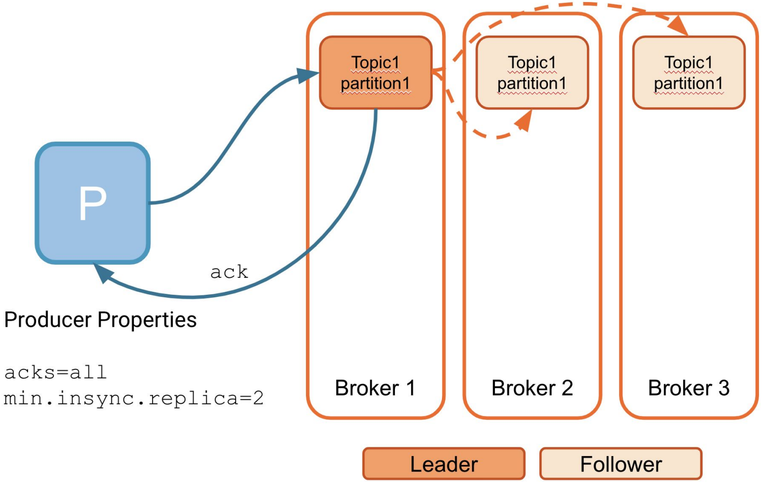
Reference

Writing in Kafka

There are two settings here that affect the producer:

acks - this is a producer-level setting // The acks property determines how you want to handle writing to kafka

min.insync.replicas - this is a topic-level setting which used in conjunction with ack=all specifies the number of replicas that must acknowledge a write for the write to be considered successful.

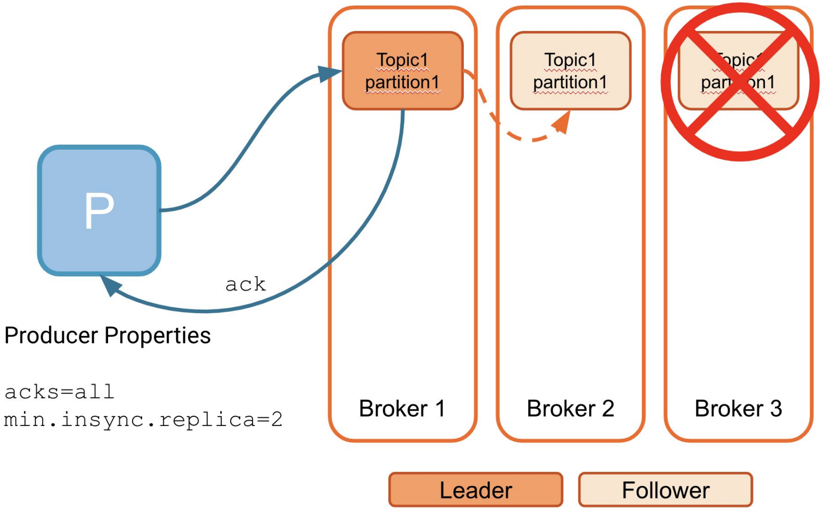


Producer Properties

```
acks=all
min.insync.replica=2
```

Writing in Kafka

min.insync.replicas is used when there is a problem in the topic, maybe one of the partitions is not in-sync, or offline. When this is the case the cluster will send an ack when min.insync.replicas is satisfied. So 3 replicas, with min.insync.replicas=2 will still be able to write:



Producer Properties

```
acks=all
min.insync.replica=2
```

Kafka consistency vs availability

Setting min.insync.replicas favours **higher consistency** (requiring writes to more than one broker) vs higher availability.

Unclean Leader election -- **favours availability** vs consistency

Let’s use Kafka producer console

See Kafka Exercises document

Internals of data storage

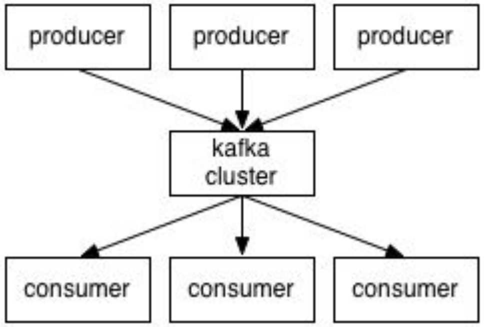
External presentation

Consumers

Kafka Concepts: Consumers

Consumers are client applications that read messages from Kafka topics. In order to read Kafka topics data you need Consumer software embedded in your app.

- Read data from a topic name;
- Consumers will know which broker to read from;
- **Data is read in order within each partition (consumer will not see offset 4 before offset 3);**
- **Consumers are part of Consumer Groups - a way to parallelize topics data consumption;**



Available Consumers:

- Kafka Consumer Console
- **Consumer Java API**
- Languages: Python, Scala, ..

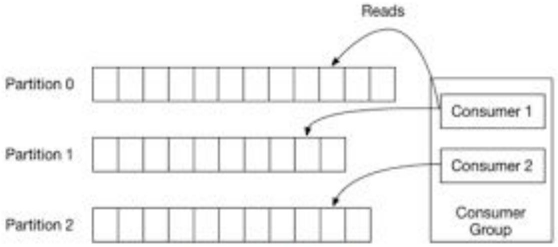


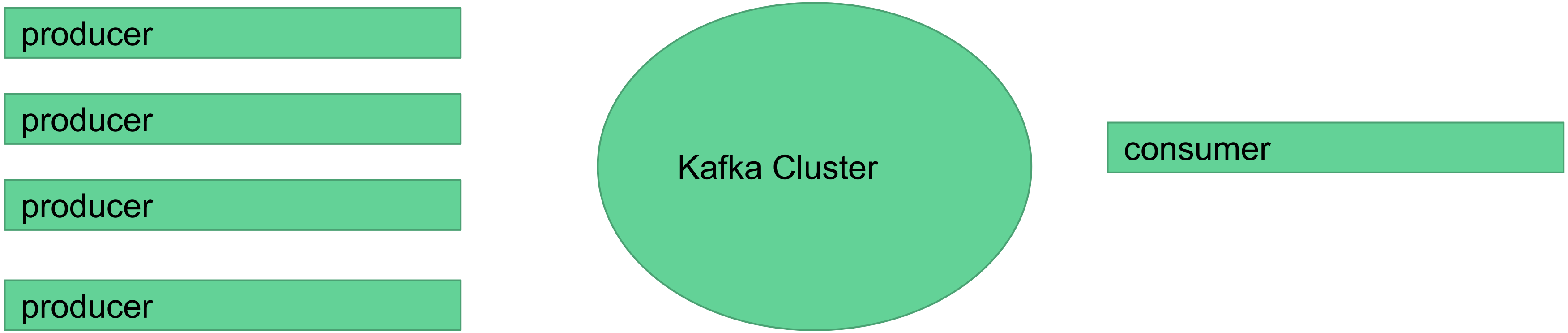
Figure 1: Consumer Group

Kafka Consumers

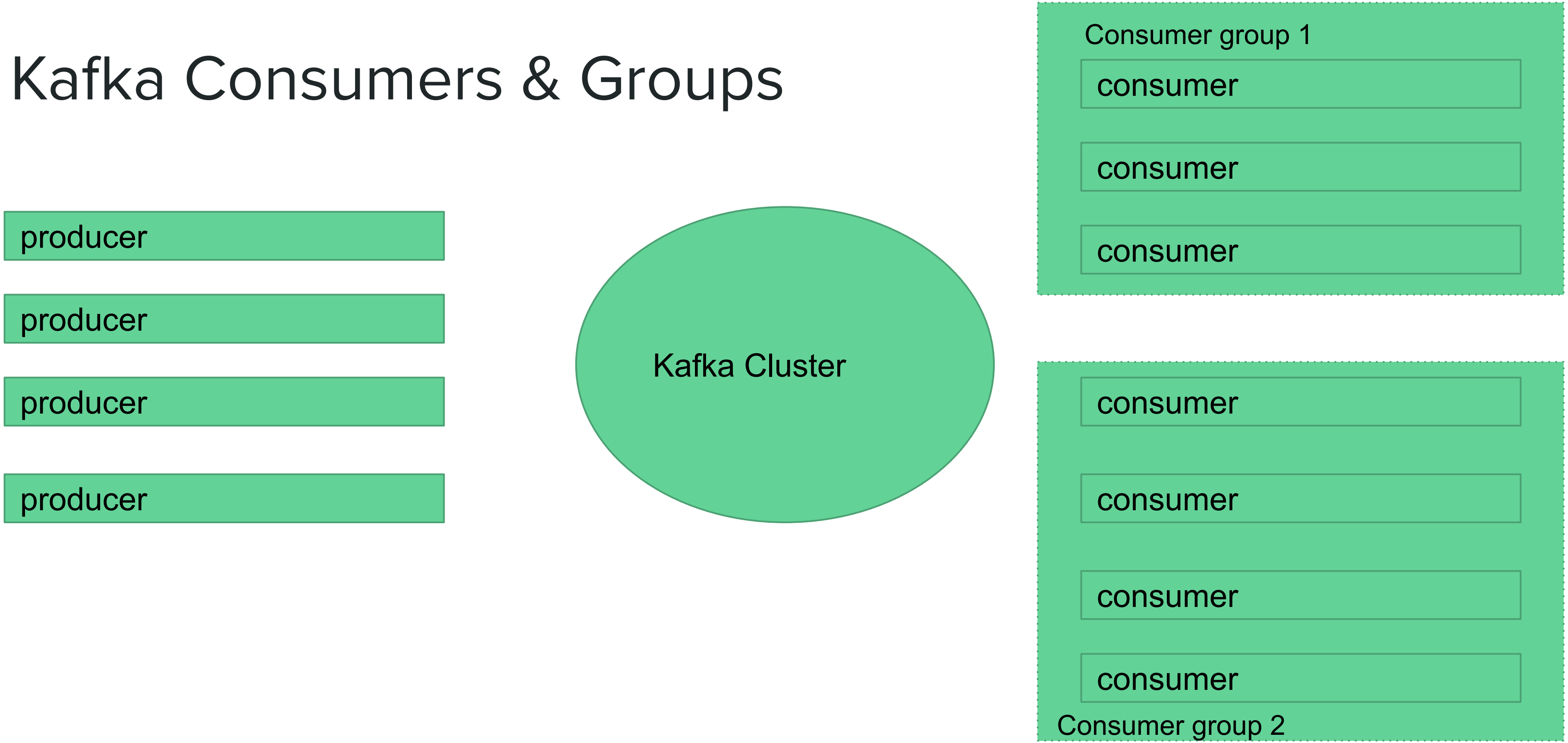
An application needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store.

App => create a consumer object => subscribe to a topic => start receiving messages => validating them and writing the results.

what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them



Kafka Consumers & Groups



When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic.

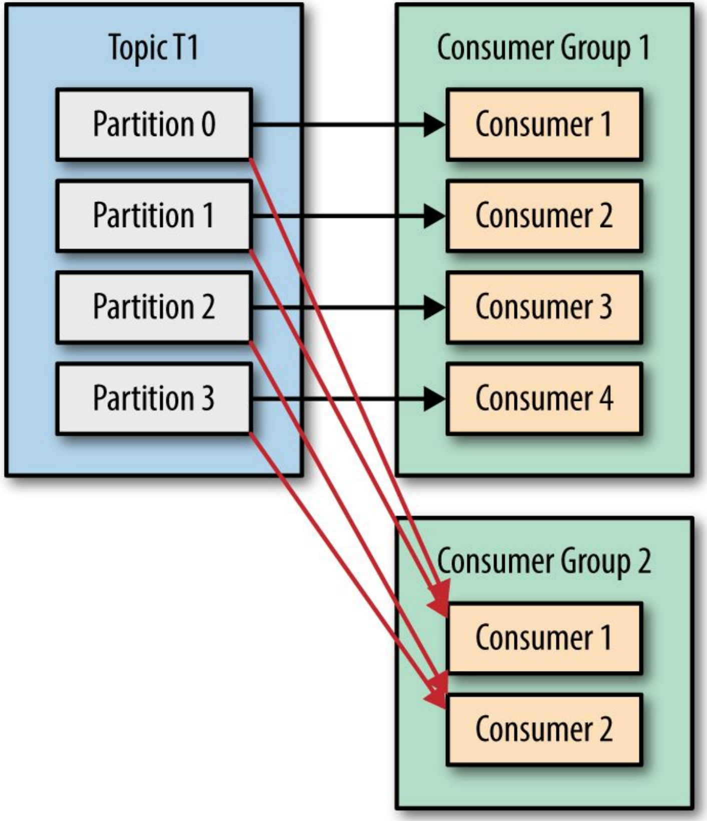
Kafka Concepts: Consumers and Consumer Groups

A Consumer Group based application may run on several nodes, and when they start up they coordinate with each other in order to split up the work.

Each Consumer node can read a partition/list of partitions (no other members in the group will be able to read from them) and one can dimension the Consumers to match number of partitions. A partition can only be worked on by one consumer in a consumer group at a time.

If the number of Consumer Group nodes is more than the number of partitions, the excess nodes remain idle. This might be desirable to handle failover.

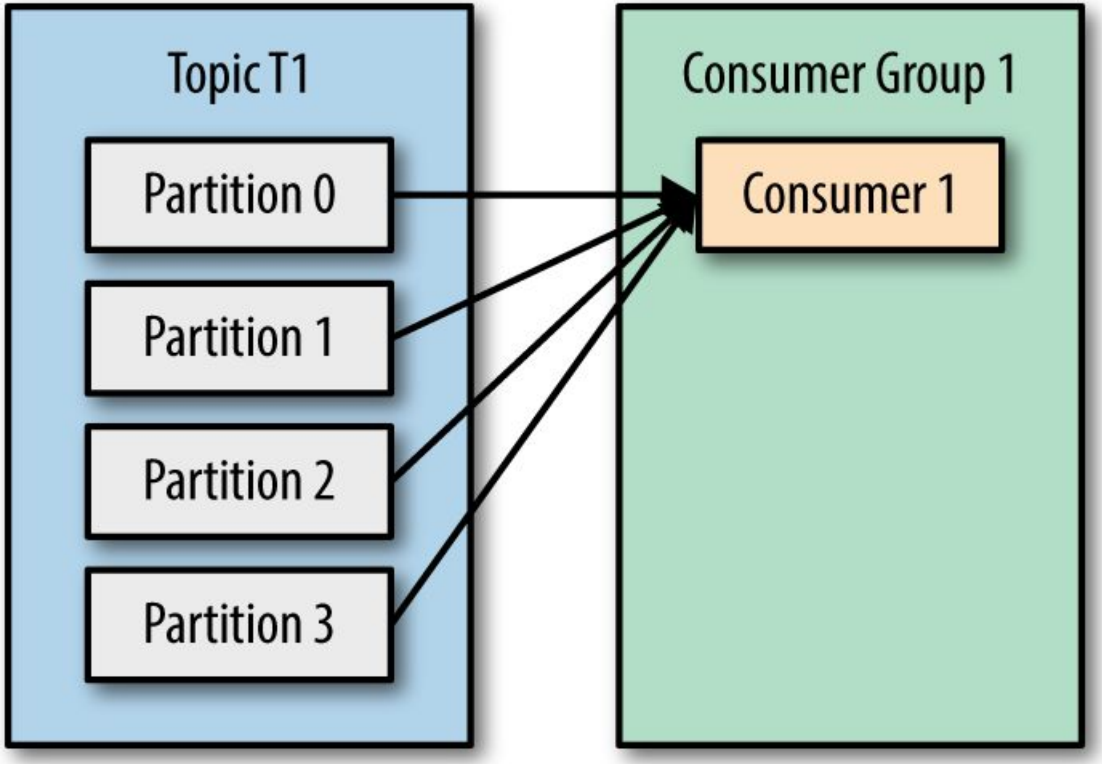
If there are more partitions than Consumer Group nodes, then some nodes will be reading more than one partition.



Consumer group - 1 consumer

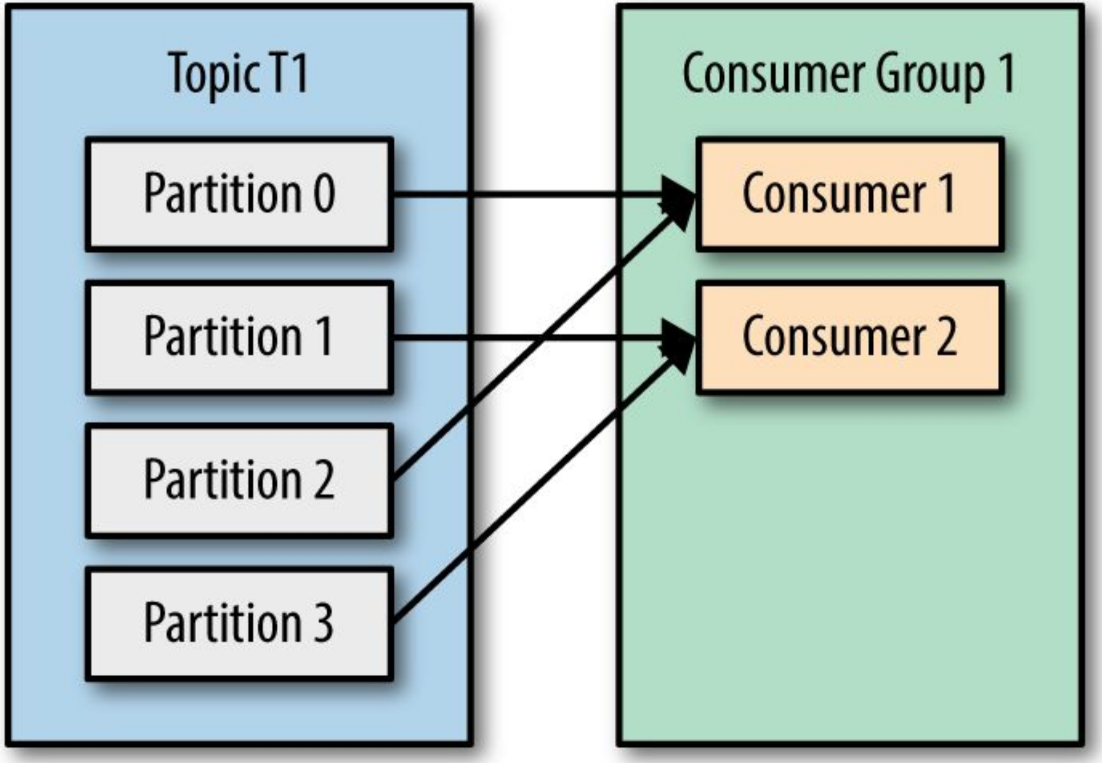
The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group.

This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases.



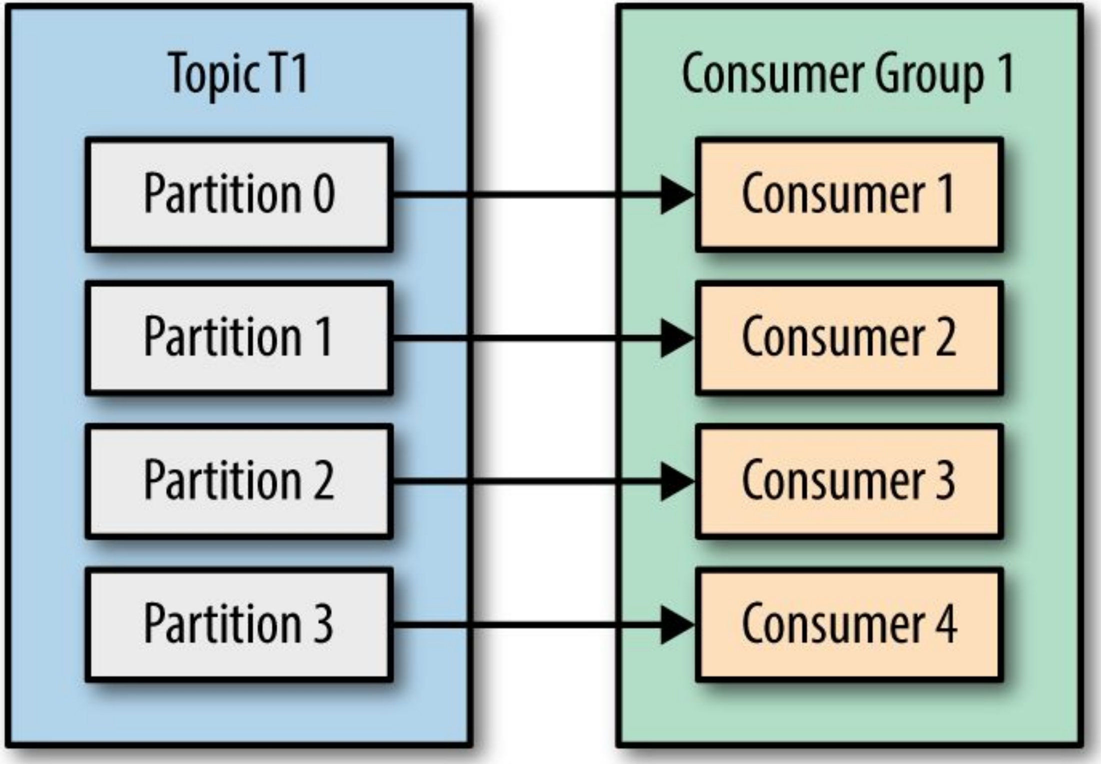
Source: Kafka, the definitive guide book

Consumer group - 2 consumers



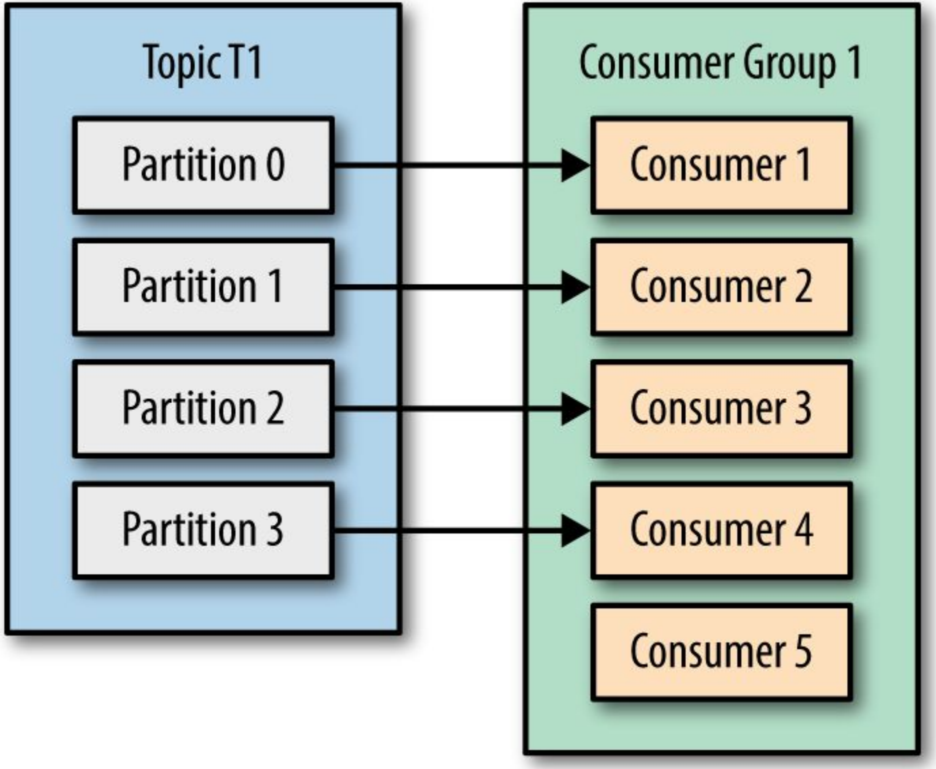
Source: Kafka, the definitive guide book

Consumer group - 4 consumers



Source: Kafka, the definitive guide book

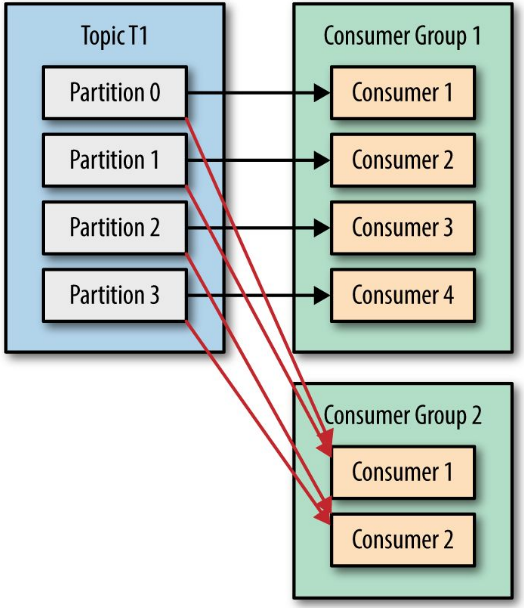
Consumer group - 5 consumers



Source: Kafka, the definitive guide book

Consumer Groups

To make sure an application gets all the messages in a topic, ensure the application has its own consumer group.



Source: Kafka, the definitive guide book

Kafka concepts: Group Coordinator, Group Leader

Consumer group co-ordinator is one of the cluster brokers, **broker which receives heartbeats(or polling for message) from all consumers of a consumer group.** Every consumer group has a **group coordinator**. If consumer stops sending heartbeats ,coordinator will trigger rebalance of partitions in the group.

The group leader is one of consumers in consumer group. When a consumer wants to join a consumer group, it sends a JoinGroup request to the group coordinator. **The first consumer to join the group becomes the group leader.**

The **group leader** **receives a list of all consumers in the group from the group coordinator** (this will include all consumers that sent a heartbeat recently and are therefore considered alive) and it is responsible for assigning a subset of partitions to each consumer.

After deciding on the partition assignment, the **consumer group leader sends the list of assignments to the GroupCoordinator which sends this information to each consumers (so Its main job is to mediate partition assignment when new members arrive, old members depart, and when topic metadata changes).** Each consumer only sees his own assignment - the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Kafka Concepts: Offsets in Consumer groups

When a group is first initialized, the consumers typically begin reading from either the earliest or latest offset in each partition. The messages in each partition log are then read sequentially. As the consumer makes progress, it **commits the offsets of messages it has successfully processed (offsets commited live in a kafka topic named _consumer_offsets_)**. For example, in the figure below, the consumer's position is at offset 6 and its last committed offset is at offset 1.

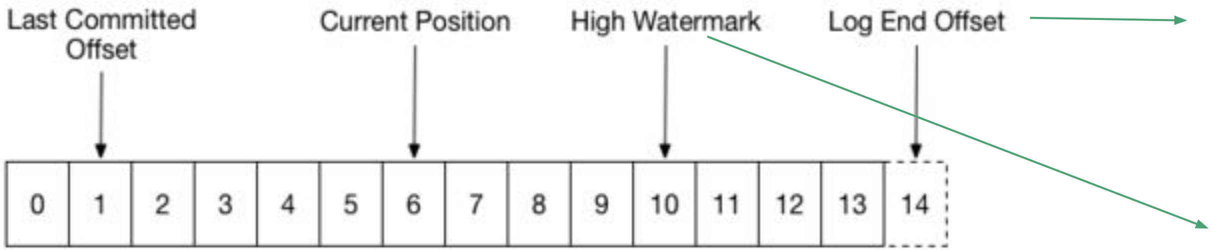


Figure 2: The Consumer's Position in the Log

Offset of the last message written in the log.

Offset of the last message successfully copied to all replicas

When a partition gets reassigned to another consumer in the group, the initial position is set to the last committed offset. If the consumer in the example above suddenly crashed, then the group member taking over the partition would begin consumption from offset 1. In that case, it would have to reprocess the messages up to the crashed consumer's position of 6.

Kafka Concepts: Delivery semantics for consumers

Consumers choose when to commit the offsets and has 3 delivery semantics

- **At most once**
    - Offsets are committed as soon as the message is received
    - If the processing goes wrong, the message will be lost
- **At least once**
    - Offsets are committed after the message is processed
    - If the processing goes wrong the message will be read again
    - This can result in duplicate processing of messages (make sure the processing in the consumer is idempotent - processing again the message won't impact your systems)

Kafka auto-commit

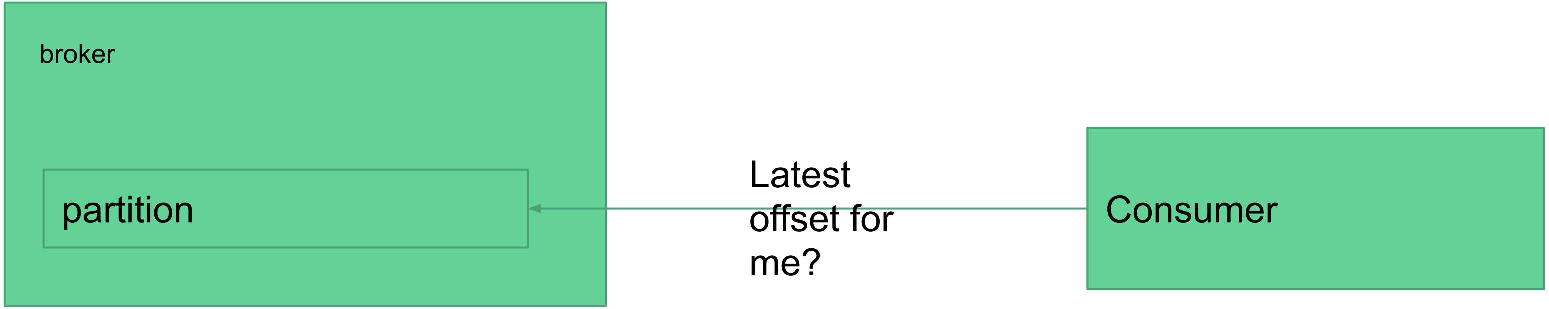
The Offset is a piece of metadata, an integer value that continually increases for each message that is received in a partition. Each message will have a unique Offset value in a partition.

When a Consumer reads the messages from the Partition it lets Kafka know the Offset of the last consumed message. This Offset is stored in a Topic named _consumer_offsets, in doing this a consumer can stop and restart without forgetting which messages it has consumed. Two properties:

Enable.auto.commit (default true)

Auto.commit.interval.ms (default 5000 - for Java Kafka Consumer). So by default every 5 seconds a Consumer is going to commit its Offset to Kafka or every time data is fetched from the specified Topic it will commit the latest Offset.

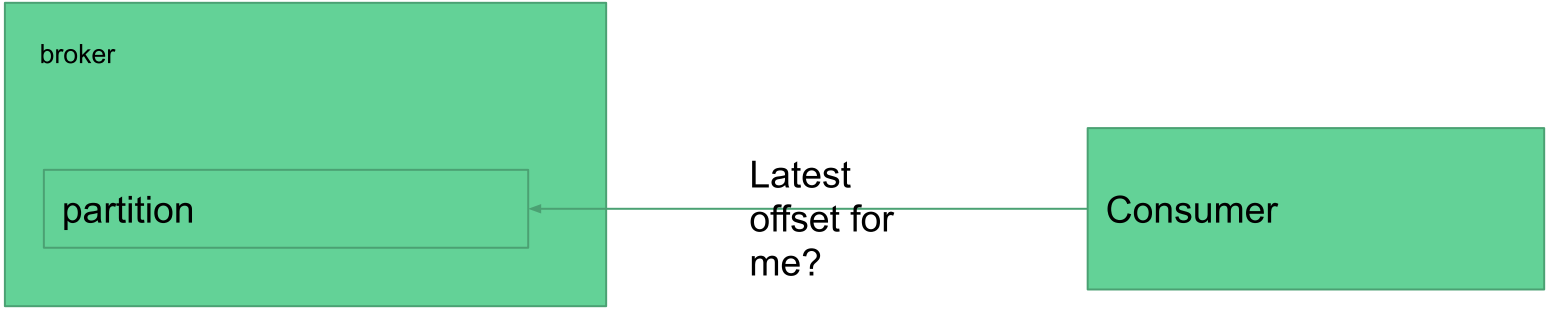
Auto-commiting messages



Step 1: latest offset requested by consumer, offset 30 is returned by broker
Step 2: auto-commit, ack is sent to broker, latest available offset will be 31
Step 3. Before processing message consumer crashes
Step 4: next read from consumer will bring message from offset 31

Commit messages manually in consumer app

enable.auto.commit: false



Step 1: latest offset requested by consumer, offset 30 is returned by broker
Step 2: auto-commit = off, ack is sent to broker after consumer processes message
Step 3. If before processing message consumer crashes => no ack sent to broker, so next read from consumer will bring message from offset 30

Kafka concepts: Group Coordinator, Group Leader

You should always close the consumer when you are finished with it. Not only does this clean up any sockets in use, it ensures that the consumer can alert the coordinator about its departure from the group.

Consumers will read from each partition only the committed data (see next slide).

Consuming data in Kafka topics - will not trigger deletion of the data. We will come back later to Data Retention in Kafka.

Kafka Concepts: Offsets in Consumer groups

Consumers will only see the offsets until High Watermark, not the Log End Offset. This prevents the consumer from reading unreplicated data which could later be lost.

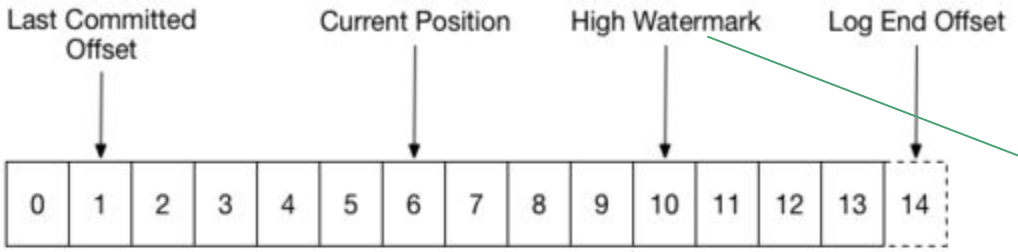


Figure 2: The Consumer's Position in the Log

Offset of the last message written in the log.

Offset of the last message successfully copied to all replicas

Let’s use Kafka console consumer/groups

Data retention & Compacted topics

How long does Kafka retains data

Two options:

- to **discard messages older than some threshold** (which can be a few weeks if you have enough disk space, giving you plenty of time to recover from failures) or **when a topic grows bigger than a certain size**, Called delete - retention option.
- or to **keep only the newest message(s) for a given key and discard older messages with the same key**. Called compaction - retention option.

When using time-based retention (older than some threshold), when a segment of the log expires, offsets within that segment or before become unavailable.

Note: Kafka also has a tool: kafka-delete-records.sh tool - you can delete records from beginning of the topic until a specified offset. Not individual messages.

Data Retention - storing only the last values

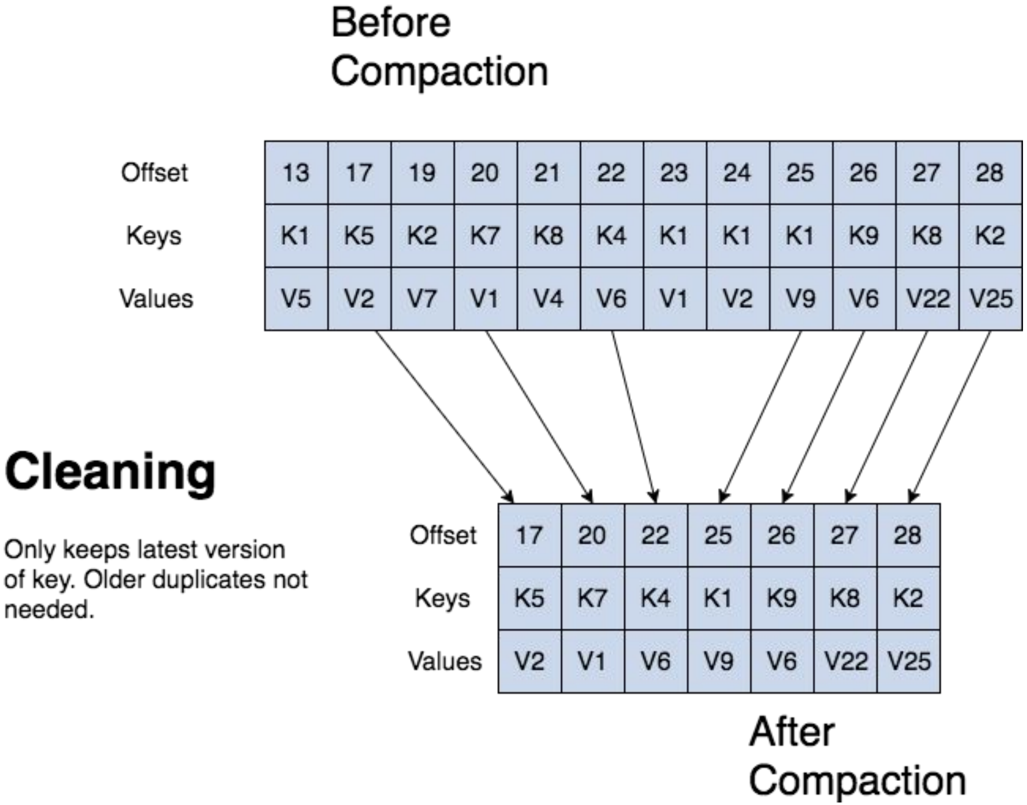
Possible use cases:

- you use Kafka to store shipping addresses for your customers. In that case, it makes more sense to store the **last address for each customer rather than data for just the last week or year.** This way, you don't have to worry about old addresses and you still retain the address for customers who haven't moved in a while.
- an application that uses Kafka to store its current state. Every time the state changes, the application writes the new state into Kafka. When recovering from a crash, the application reads those messages from Kafka to recover its latest state. In this case, it only cares about the **latest state before the crash**, not all the changes that occurred while it was running.

Kafka supports such use cases by allowing the retention policy on a topic to be either delete: which deletes events older than retention time Or to compact: which only stores the most recent value(s) for each key in the topic. Obviously, setting the policy to compact only makes sense on topics for which applications produce events that contain both a key and a value. If the topic contains null keys, compaction will fail.

Log compaction

Log compaction is a granular retention mechanism that **retains the last update(s) for each key.** A compacted topic log contains a full snapshot of final record values for every record key not just the recently changed keys.



Before Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

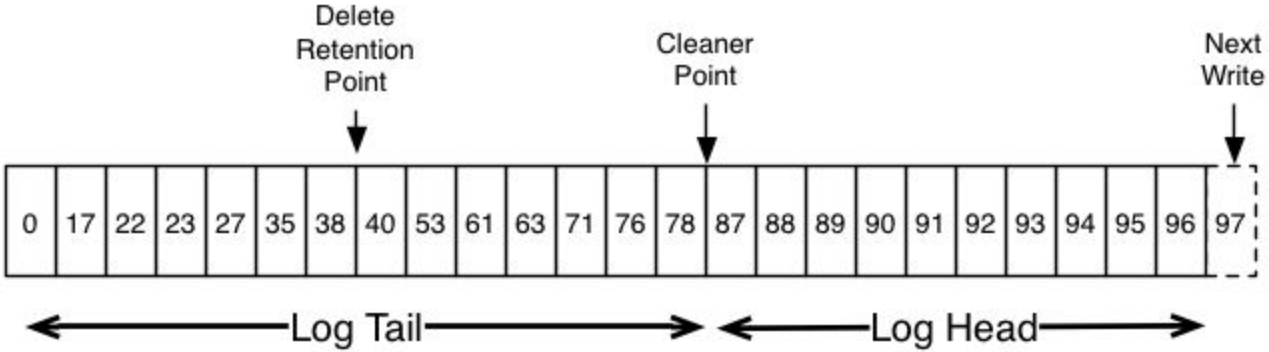
Cleaning

Only keeps latest version of key. Older duplicates not needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After Compaction

Log Compaction



Log compaction adds an option for handling only the tail of the log. The log head is not touched by compaction.

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "**delete retention point**" .

Log Compaction - you can delete messages

Kafka log compaction also allows for deletes. **A message with a key and a null payload acts like a tombstone, a delete marker for that key.** This delete marker will cause any prior message with that key to be removed.

Tombstones get cleared after a period.

Log compaction periodically runs in the background by recopying log segments. Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers.

When does Kafka delete a message

“The simplest way to remove messages from Kafka is to simply let them expire.”

By default, if your topic comes with the default configuration, the **data is deleted after a week (default) by Kafka from the receive timestamp**. See log.retention.hours.

Admin API (Confluent) - that allows to delete data explicitly if they are older than some specified time or offset.

Log compaction provides an alternative such that it maintains the most recent entry for each unique key, rather than maintaining only recent log entries. Your topic needs to be log compacted (specify cleanup.policy=compact in your topic configuration), in this case from time to time Kafka will delete all the messages that share the same key, except the most recent one(s).

Log compaction allows as well records to be marked for removal and then later deleted when the compaction process runs.

Note: Kafka also has a tool: kafka-delete-records.sh tool - you can delete records from an offset onwards. Not individual messages.

Exercise compaction

Log Compaction guarantees

- any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets.
- **Min.compaction.lag.ms**: provides a lower bound on how long each message will remain in the (uncompacted) head
- Max.compaction.lag.ms: can be used to guarantee the maximum delay between the time a message is written and the time the message becomes eligible for compaction.
- Ordering of messages is always maintained. Compaction will never reorder messages, just remove some.
- The offset for a message never changes. It is the permanent identifier for a position in the log.
- **Any consumer progressing from the start of the log will see at least the final state of all records in the order they were written.**
- all delete markers for deleted records will be seen, provided the consumer reaches the head of the log in a time period less than the topic's delete.retention.ms setting (the default is 24 hours).

Preserving messages ordering by the producer

In general, messages are written to the broker in the same order that they are received by the producer client. However, if you enable message retries by setting retries to a value larger than 0 (which is the default), then message reordering may occur since the retry may occur after a following write succeeded. To enable retries without reordering, you can set max.in.flight.requests.per.connection to 1 to ensure that only one request can be sent to the broker at a time. Without retries enabled, the broker will preserve the order of writes it receives, but there could be gaps due to individual send failures.

Kafka can change automatically the partitions/topic?

No - the number of partitions per topic needs to be manually increased.