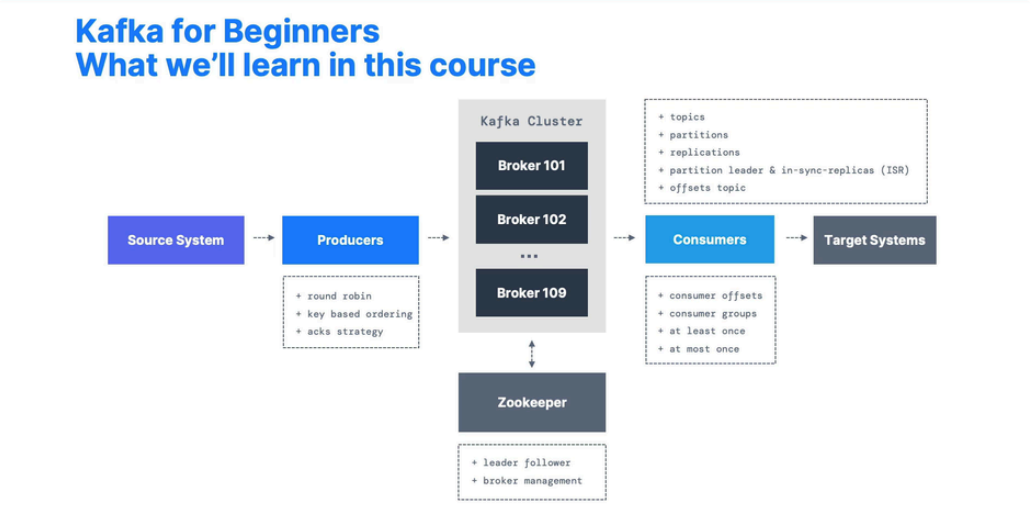
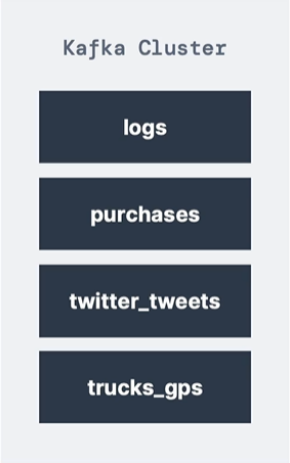
Kafka: Basics

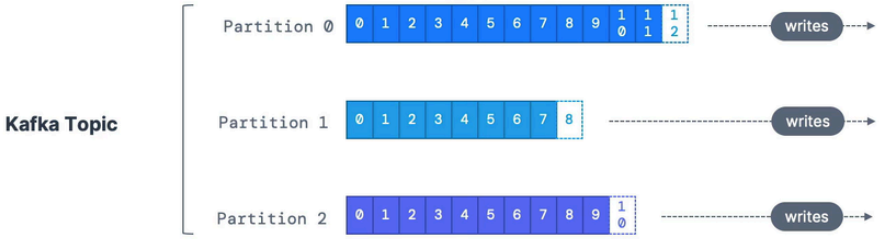


Topics



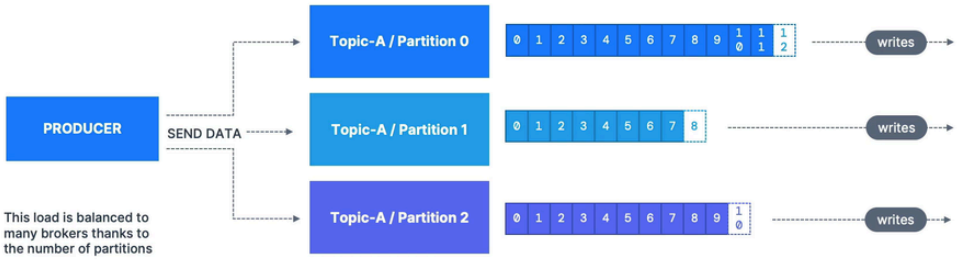
- It is a stream of Data
- A Kafka Cluster can have multiple topics
- It is like a table in a database (one bd can have multiple tables)
- Topic is identified by its name
- Supports any kind of message format - JSON, Avro, text file, binary, etc.
- Sequence of message is called Data Streams
- Topics cannot be queried
- Data are sent to topics via Producers and read via consumers

Partitions


Kafka Topic
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12
writes
Partition 1
0 1 2 3 4 5 6 7 8
writes
Partition 2
0 1 2 3 4 5 6 7 8 9 10
writes

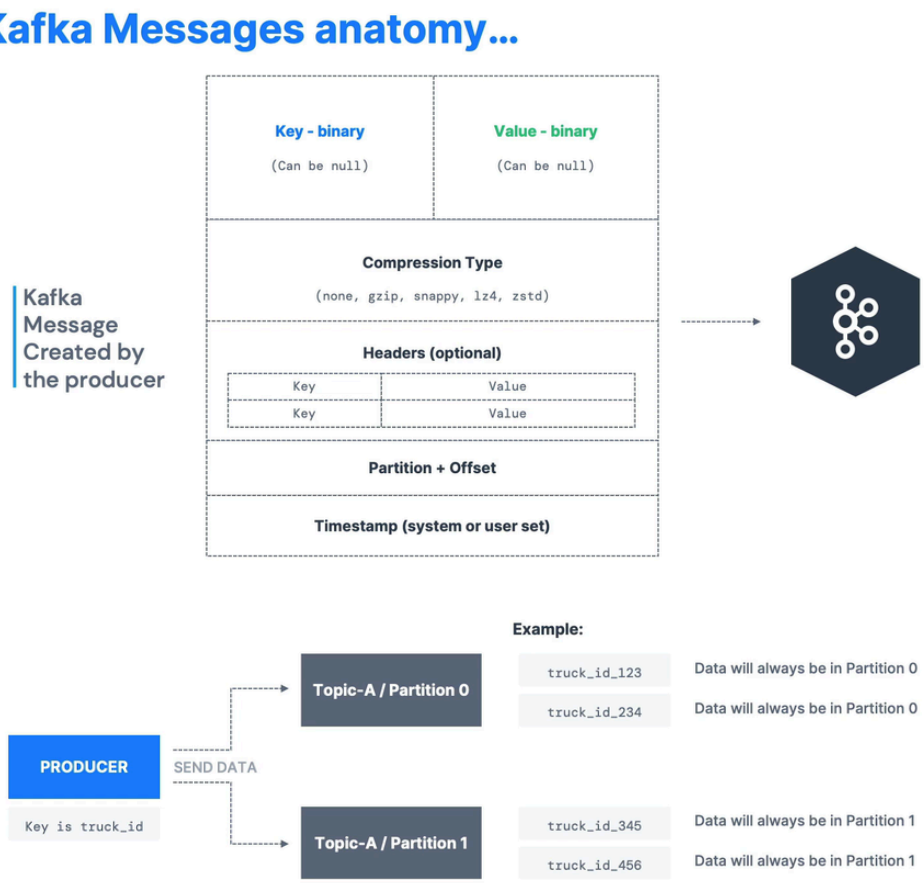
- Topics can be split into any number of partitions
- Messages sent will end up in one of these partitions
- Message in each partitions will be ordered
- Each message is getting an ID in a partitions. This is called an **Offset.**
- Kafka topics are **immutable**, thus data cannot be **deleted** or **updated.**
- Data is kept in Kafka for a limited time. Default is One week. After this it will be deleted.
- Offsets are never reused, even if old message is deleted.
- Data is assigned to a random partition, unless we provide a key.

Producers



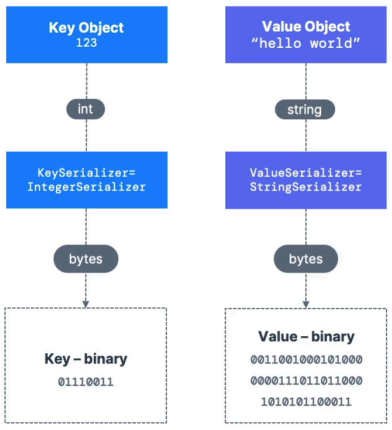
- They write data into Kafka Topics
- They know in partition they need to write to, and which kafka broker has it.
- Producers decide in advance in which partition to write to.
- They also know how to recover automatically if a broker fails
- Load Balancing happens in producers
- Producers can choose to receive acknowledgment of data writes:
  - **acks=0**: Producer won't wait for acknowledgment. (possible data loss)
  - **acks=1**: Producer waits for leader acknowledgement. (limited data loss)
  - **acks=all (-1)**: Producers waits for leader + replica acknowledgment. (no data loss)

Message Keys



- Producers can send keys with message (String, number, binary, etc.)
- If no key is provided, then the data is sent in a round robin message to each partition (P1, then P2, then P3….)
- If we have a key, then all message for that key will go to same partition. Initially the partition is decided based on hashing.
- We send keys typically when we need message ordering for a specific filed (ex: truck_id).

Message Serializer



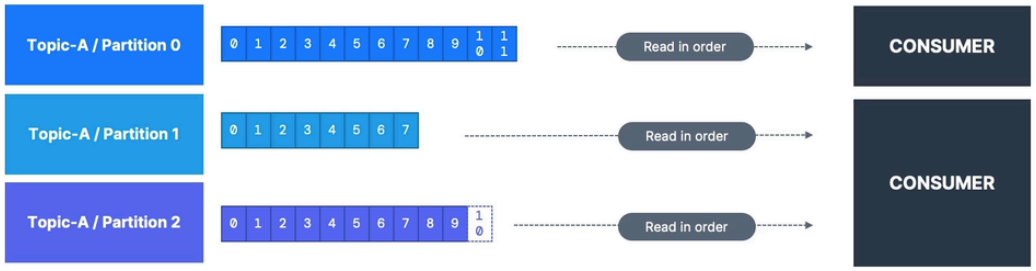
- Kafka accepts bytes as input and sends bytes as output
- A Message Serialization transforms objects/data into bytes
- They are only used on value and key.
- Common Serializers
  - String (incl JSON)
  - Int, Float

- Avro
- Protobuf

<SAFE KAFKA PRODUCER SETTINGS>

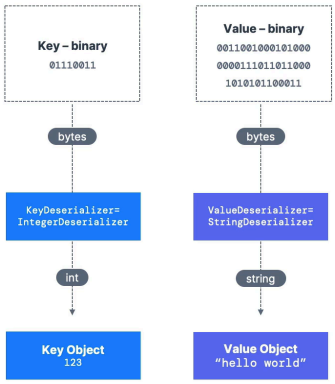
- Kafka 3.0
- acks = all (-1)
- min.insync.replicas = 2 (broker/topic level) - Insures two brokers in ISR at least have the data after an ack.
- enable.Idempotence = true - Duplicates are not introduced due to network retires.
- retries = MAX_INT (producer level) - Retry until delivery.timeout.ms is reached.
- delivery.timeout.ms = 120000 - Fail after retrying for 2 minutes
- max.in.flight.requests.per.connection = 5 - Ensure max performance while keeping message ordering

Consumers



- They read data from topic
- They request data from kafka brokers/servers.
- They automatically know which kafka server to read from
- In case of broker failure, kafka consumers know how to recover from this.
- Data are read in order from low to high offset within each partition.

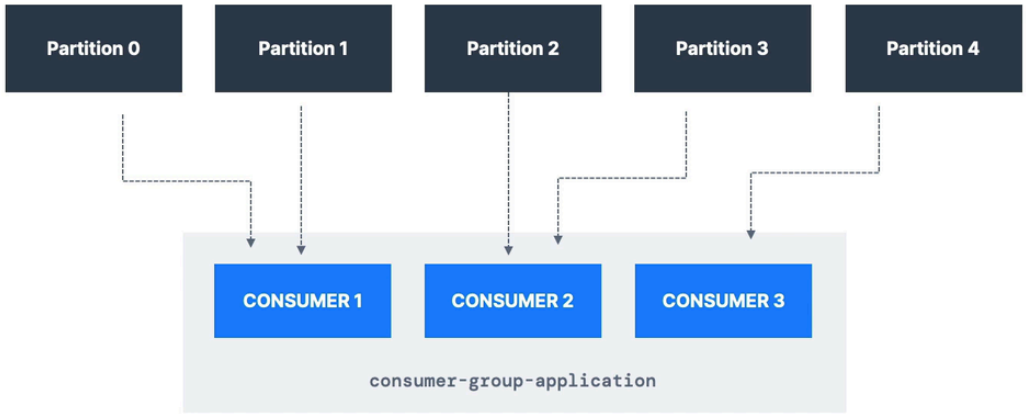
Message De-serializer



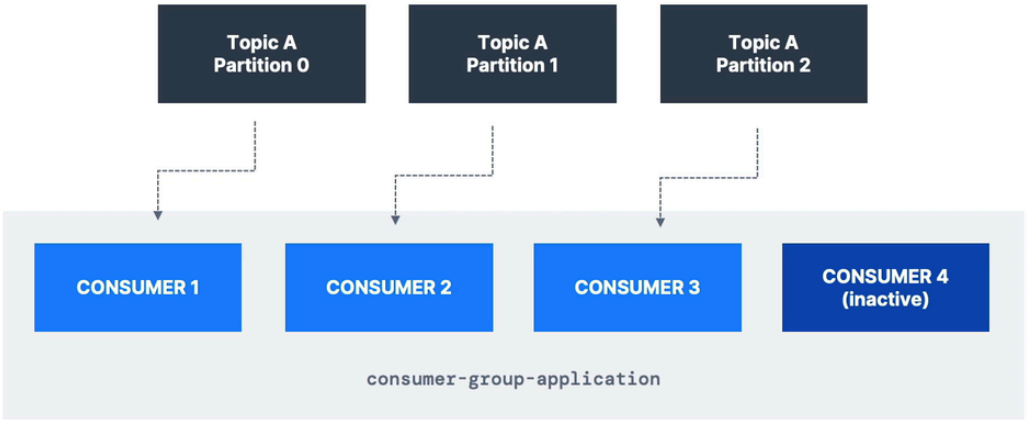
- They transform bytes into object/data
- They are used on key and values of the message
- They need to know what type of message it is to use the correct de-serializer
- Common De-serializers
  - String (incl JSON)
  - Int, Float
  - Avro
  - Protobuf

- Consumer needs to know in advance what is the expected format of the key and value.
- The serializer/de-serializer type must not change during a topic lifecycle. Instead create a new topic.

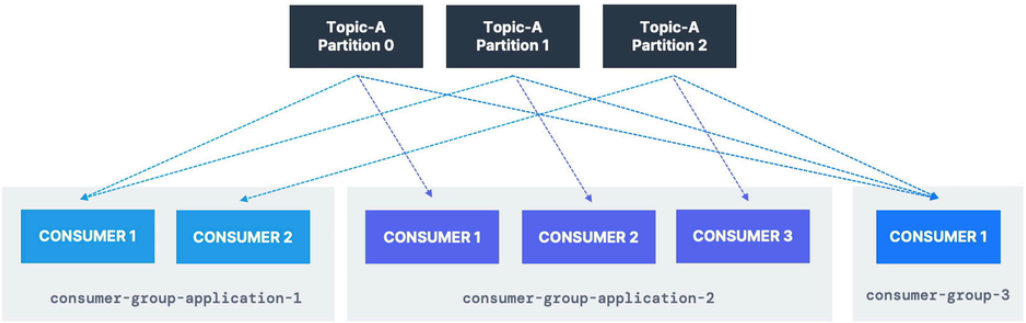
Consumer Groups



- All consumers in an application read data as a consumer group.
- Each consumer within a group reads from exclusive partitions.

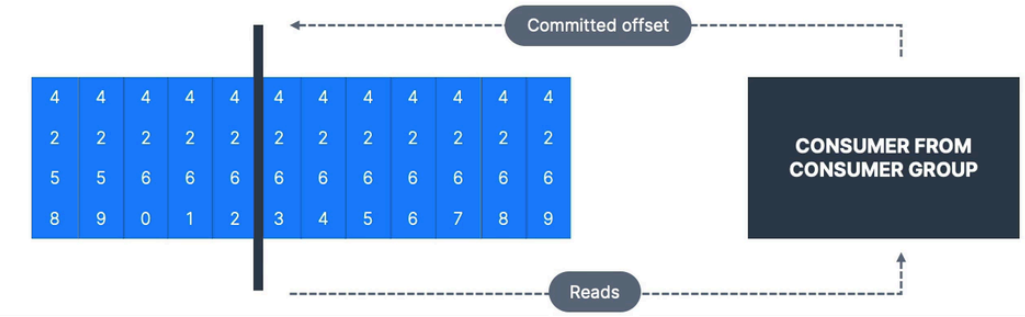


- If number of consumers in a group are more than the partitions, then the extra consumers remain inactive.



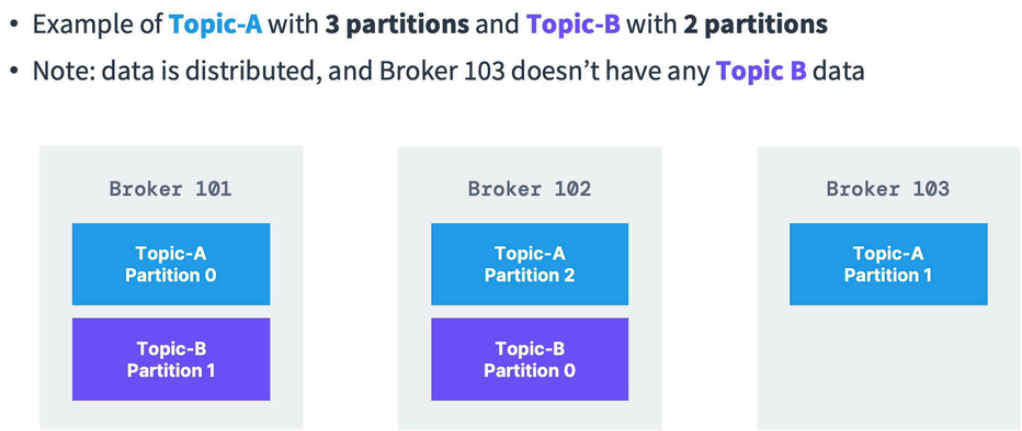
- We can have multiple consumer group on the same topic.
- To create distinct consumer groups, use the consumer property group.id.

Consumer Offsets



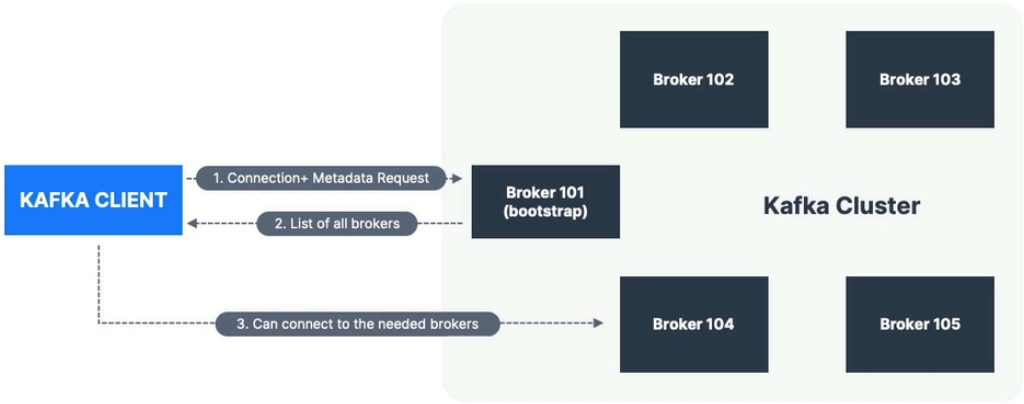
- The offset is crucial to determine, till where a consumer has consumed the data in case of a crash.
- When a consumer in a group processes the data received from kafka, it periodically commits the offsets.
- The kafka broker writes to __consumer_offsets (not the group itself)
- If the consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets.
- Java Consumers by default automatically commits offsets (at least once)
- Three delivery semantics are available for manual commits
  - At Least Once (usually preferred)
    - Offsets are committed after message is processed
    - If processing goes wrong, message will be read again
    - This causes duplicate processing of message.
    - Here make sure processing is idempotent (i.e. processing the massage again must not impact the system)
  - At Most Once
    - Offsets are committed as soon as messages are received.
    - If processing goes wrong, some messages will be lost, as they are not read again.
  - Exactly Once
    - When we read from a topic and write to another topic. Use Transactional API
    - When reading from a kafka topic and writing to external System. Use Idempotent Consumer.

Brokers

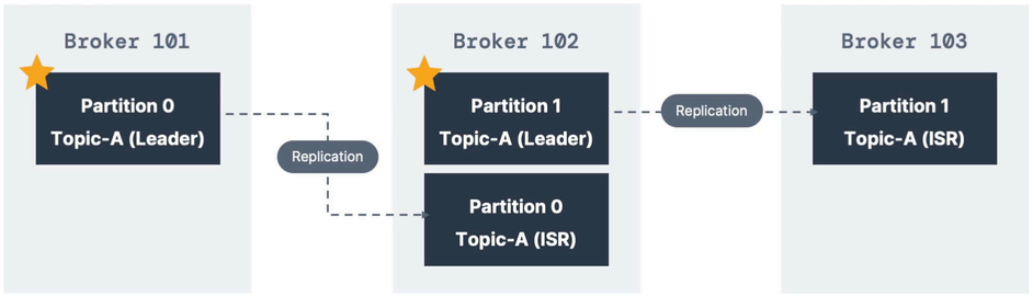


- Brokers can also be referred as servers in Kafka.
- A cluster consists of multiple brokers.
- It is identified with an ID (integer)
- Each broker contains certain topic partition
- After connecting to just one broker, we will get connected to the entire cluster.
- We can have as many brokers as we want.

- Each Kafka broker is called a “bootstrap server”

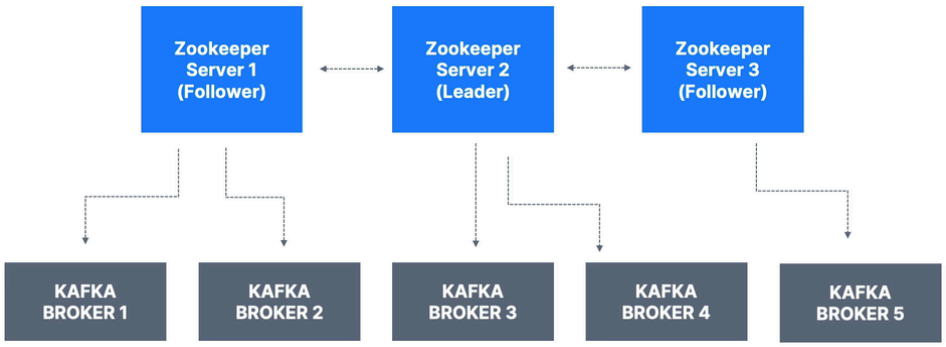


Topic Replication Factor



- Topics should have a replication factor > 1
- Due to this, if a broker is down, then another broker can serve the data.
- At any time, only one broker can be a leader for a given partition.
- Producers can only send data to the broker that is leader of a partition.
- The other brokers will replicate the data.
- ISR: In-sync replica. This happens when the data is replicated fast enough.
- Thus, each partition has one leader and multiple ISR.
- Producers only write to the leader broker for a partition.
- Consumers can read from the closest partition replica. This reduces latency.
- When a broker goes down the new broker becomes a leader for the partition.
- If we have a replication factor of 3. then we can withstand 2 broker loss.
- **Topic Data Durability = (Replication Factor - 1)**

Zookeeper



- It manages the brokers.
- In case a leader goes down, it performs election to select a new leader.
- Send notification to kafka, in case of changes. (e.g. new topic, broker dies, broker comes back up, delete topics, etc.)
- Kafka 2.x cannot work without zookeeper.
- Kafka 3.x can work without zookeeper. (uses Kafka Raft instead) (KIP-500) (KRaft)
- Kafka 4.x will not have zookeeper. It will only have KRaft
- Zookeeper operates with odd number of servers (1,3,5,7). Usually the max is 7.
- Zookeeper has a leader (writes) the rest of the server are followers (reads).
- It is less secure.

Kafka KRaft

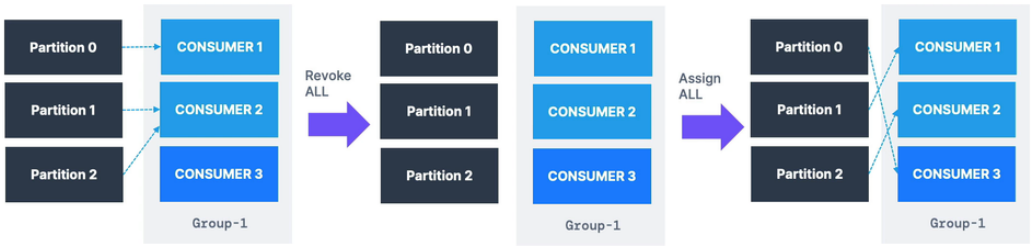
- KIP-500
- Zookeeper has scaling issue when Kafka clusters > 100,000 partitions.
- KRaft helps scaling to millions of partitions.
- Improve stability. Easier to monitor, support and administer.
- Single security model for whole system.
- Single process to start kafka.
- faster controller shutdown and recovery time.

Partition Rebalancing

- Moving of partition between consumers is termed as a rebalance.
- Reassignment of partitions occur when a consumer leaves or joins a group
- Also happens when an admin adds a new partition to the group

- There are two ways this rebalance could happen
  - Eager Rebalance (default)
  - Cooperative Rebalance (Incremental Rebalance)

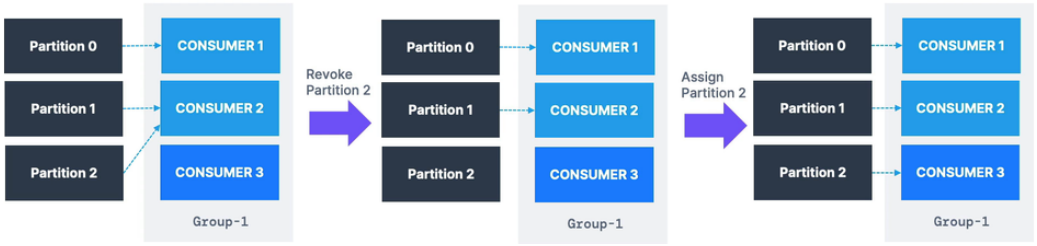
Eager Rebalance



- This is the default rebalance technique

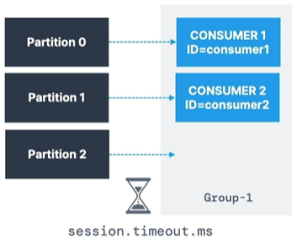
- When a new consumer joins or leaves a group, all the existing consumers give up their membership of the partitions
- They then rejoin the consumer group and get a new partition assignment.
- Due to this, for a short period of time, the entire consumer group stops processing. This is called the “Stop the World Event“
- The consumers will not necessarily get back the same partition as they used to have.
- Assigners:-

Cooperative Rebalance (Incremental Rebalance)



- Reassigns a small subset of partitions from one consumer to another
- Other consumers that don’t have reassigned partitions can still process data uninterrupted
- It goes through several iterations to find a stable assignment
- This helps in avoiding Stop the World events.
- Kafka Connect and Kafka Streams have Cooperative Rebalance turned on by default.

Static Group Membership



- Usually, when a consumer leaves the group, its partition is revoked and reassigned.
- When the consumer joins back, it will have a new member ID, and a new partition is assigned to it.
- Alternatively, to make a consumer a static member, you can specify a group.instance.id.
- Upon leaving a group, the consumer has sessions.timeout.ms to join back the group.
- If they join back, they will get back its partition, or else, a rebalance occurs.
- This is helpful in Kubernetes, or when a consumer maintains a local state or cache.

Assigners

- RangeAssignor: Assigns partition on a per topic basis. Leads to imbalance.
- RoundRobin: Assigns partition in round-robin fashion across all topics. Optimal balance.
- StickyAssignor: Balanced like round-robin, then minimizes partition movements when consumer joins/leaves the group.
- CooperativeStickyAssignor: Identical to StickyAssignor, but supports cooperative rebalance, so that consumers can keep consuming from topic.
- New default assignor for Kafka 3.x is a list of assignor
  - [RangeAssignor, CooperativeStickyAssignor]
  - Uses RangeAssignor be default, but allows upgrading to the CooperativeStickyAssignor.
  - This is done by removing RangeAssignor from the list.
- StreamsPartitionAssignor

Message Compression

- Producers usually send text based data - e.g. JSON.
- Thus its important to apply compression to the data.
- Compressions can be enabled at producer level.
- Does not require config changes in the broker or consumer.
- compression.type can be
  - none (default)
  - gzip
  - lz4
  - snappy
- Its more effective when bigger batch of messages are being sent to Kafka
- Advantages
  - Smaller producer request size
  - Faster to transfer data over network
  - Less latency
  - Better throughput
  - Better disk utilization in Kafka
  - Message stored on disk are smaller
- Disadvantages
  - producers must commit some CPU cycles to compression
  - Consumers must commit some CPU cycles to decompression
- Compression can also be applied at Broker or topic level.
- compression.type = producer (default)
  - The broker would take the compressed batch from the producer client and would write to the topic's log without recompressing the data.
- compression.type = none
  - All batches are decompressed by the broker
- compression.type = lz4
  - If it matches the producer setting, data is stored on the disk as it is.
  - If it's different compression setting, batches are decompressed by the broker, then recompressed using the specified compression algorithm.
- Broker side compression utilizes additional CPU cycles.