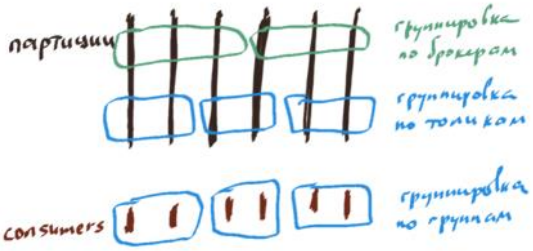
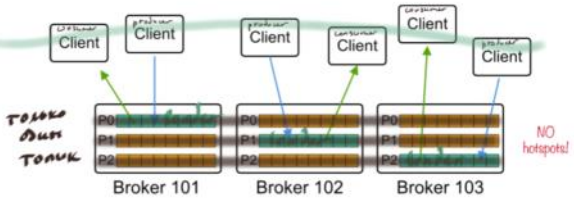
_ОБЩИЕ СУПЕРИЛЛЮСТРАЦИИ

29 ноября 2020 г. 14:06

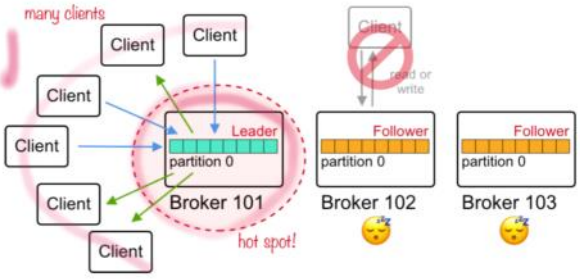


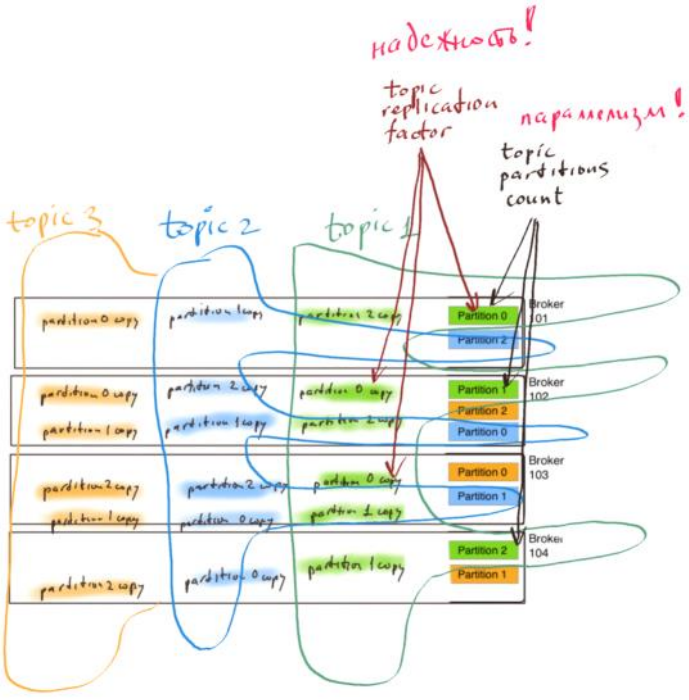
Зачем нужно партиционирование — ШАРДИНГ

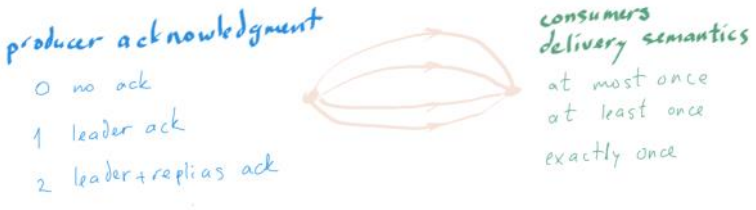
good

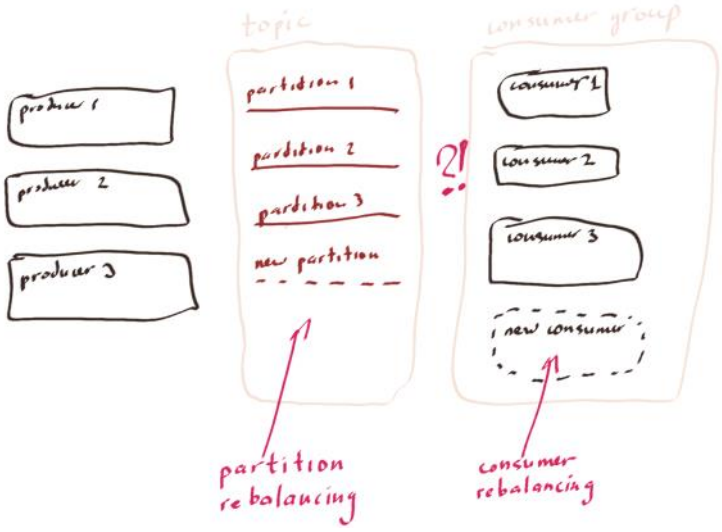


bad



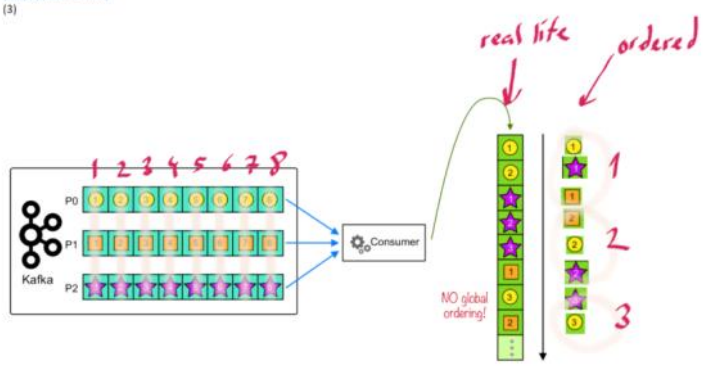


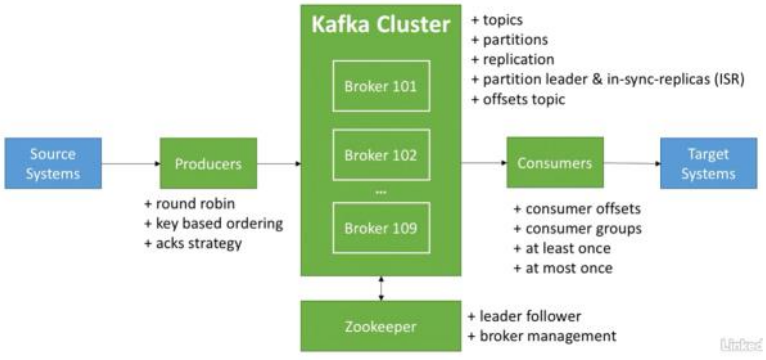




порядок на уровне топика сложно сделать

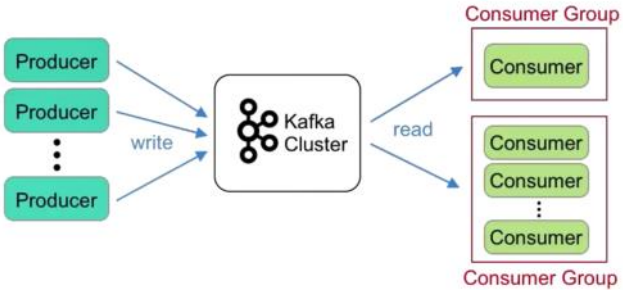
- на практике НЕ встречаются случаи, где порядок нужен **именно на уровне всего топика** а не партиции,
- (1) но если вдруг такое понадобится то нужно просто сделать **топик из всего одной партиции** но это возможно только когда нет большой нагрузки
- (2) можно засосать все данные из топика в свою прогу, затем переупорядочить их и далее направить дальше (но они уже будут упорядоченными)
- (3)



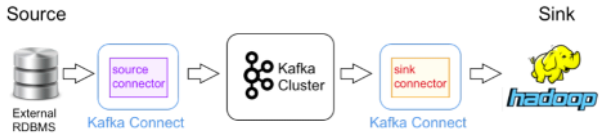


консьюмеры бьются по группам а продьюсеры нет

- Producers and consumers are both so called [redacted]

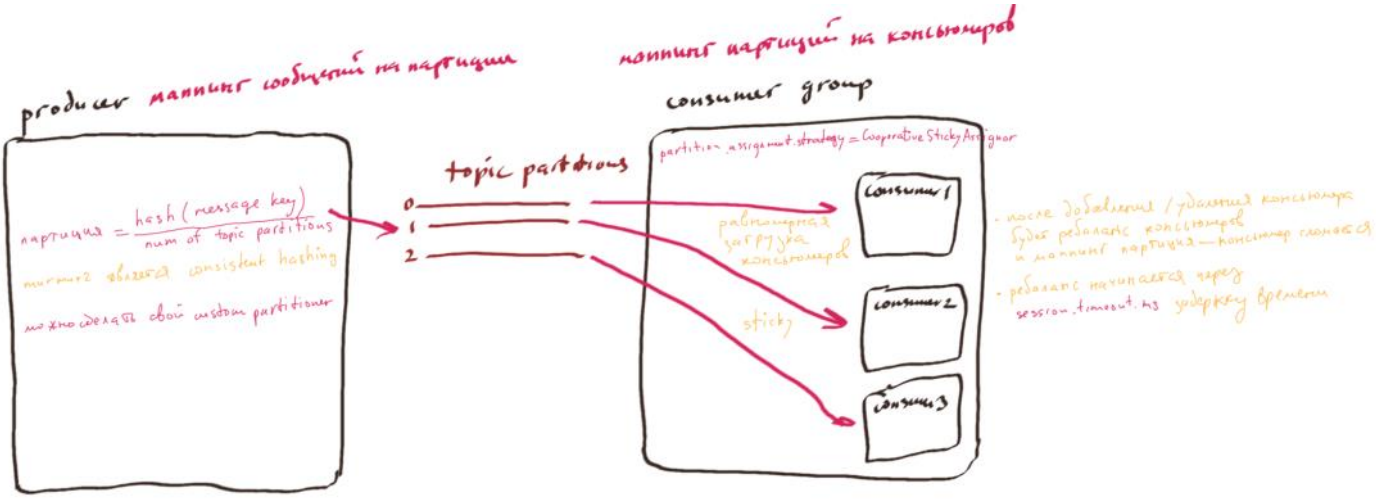


source и sink термины для кафка коннекторов/адаптеров

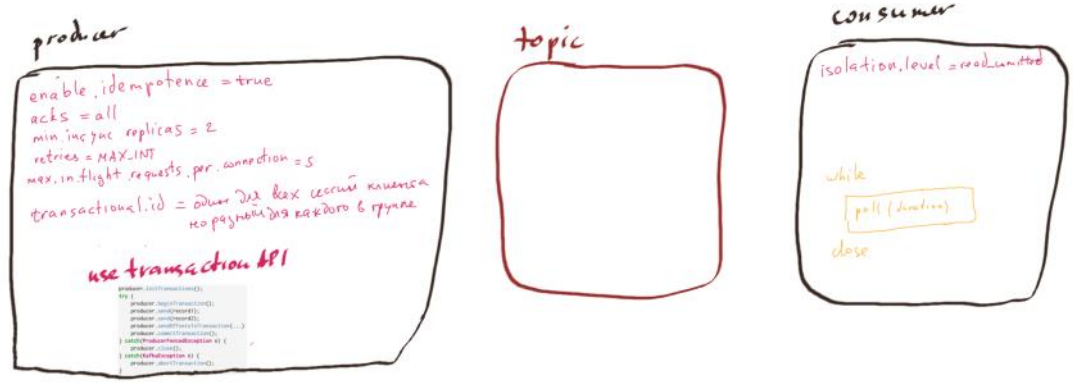


(A) пример жесткой настройки продьюсер-партиция-консьюмер

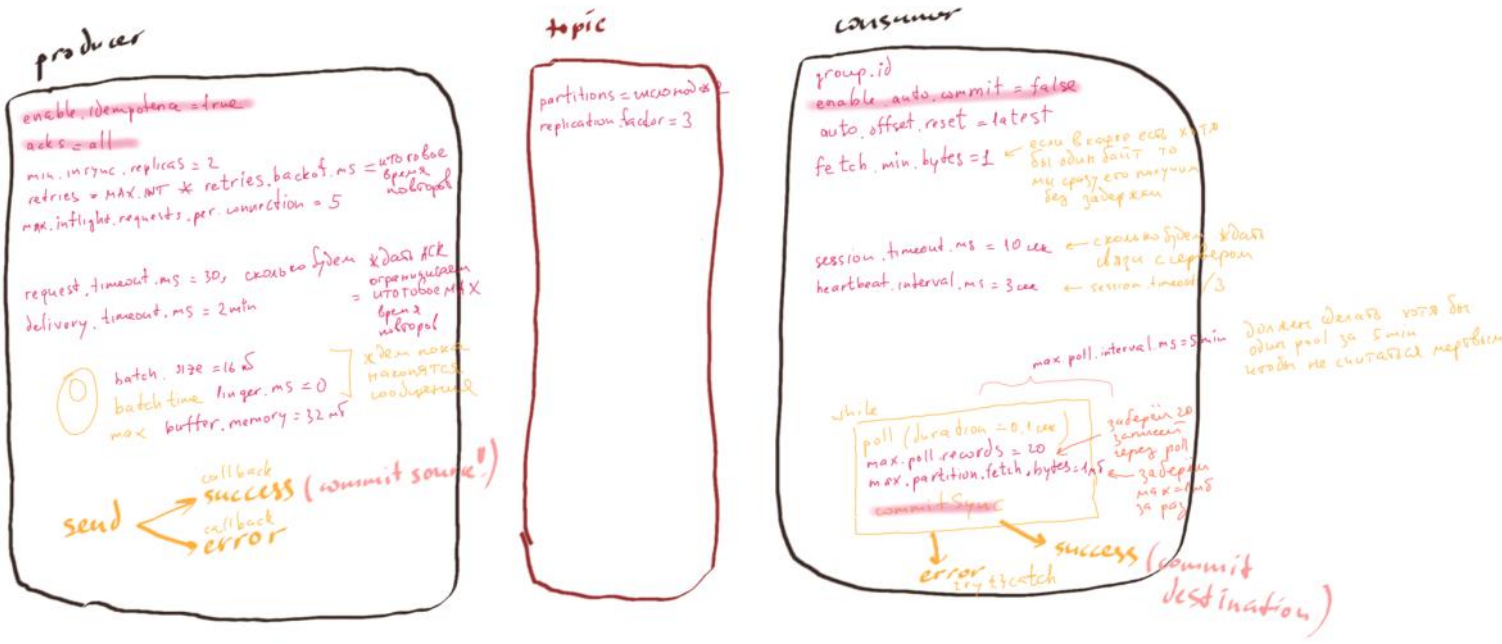
(B) пример default partitioner + auto consumer rebalancing



(1) пример EOS



(2) пример попроще (без строгой консистентности)

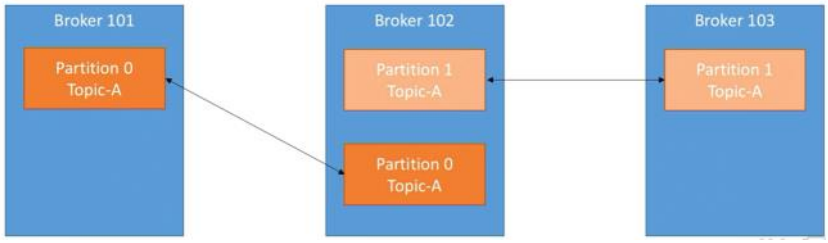

producer
enable.idempotence=true
acks=all
min.insync.replicas=2
retries=MAX.INT * retries.backoff.ms = итоговое время на повторы
max.inflight.requests.per.connection=5
request.timeout.ms=30, сколько будем ждать ACK
delivery.timeout.ms=2min = организация итогового max времени на повторы
batch.size=16кБ
batch.time linger.ms=0
max buffer.memory=32мБ
ждем пока накопятся сообщения
send
callback success (commit source!)
callback error
topic
partitions=кол-во нод
replication.factor=3
consumer
group.id
enable.auto.commit=false
auto.offset.reset=latest
fetch.min.bytes=1 ← если в кафке есть хотя бы один байт то мы сразу его получим без задержки
session.timeout.ms=10сек ← сколько будем ждать ответа с сервером
heartbeat.interval.ms=3сек ← session.timeout/3
max.poll.interval.ms=5min должен делать хотя бы один poll за 5min иначе не считаемся мертвым
while
poll (duration=0.1сек)
max.poll.records=20 ← забираем 20 записей через poll
max.partition.fetch.bytes=1мБ ← забираем max=1мБ за раз
commitSync
error try-catch
success (commit destination)

topic ==table

29 ноября 2020 г. 11:56

topic replication factor ==3

- Topics should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: Topic-A with 2 partitions and replication factor of 2



- Example: we lost Broker 102
- Result: Broker 101 and 103 can still serve the data

Вместо того чтобы пытаться предотвращать сбои, фреймворк учитывает их возможность с помощью репликации блоков данных по кластеру (по умолчанию коэффициент репликации равен 3).

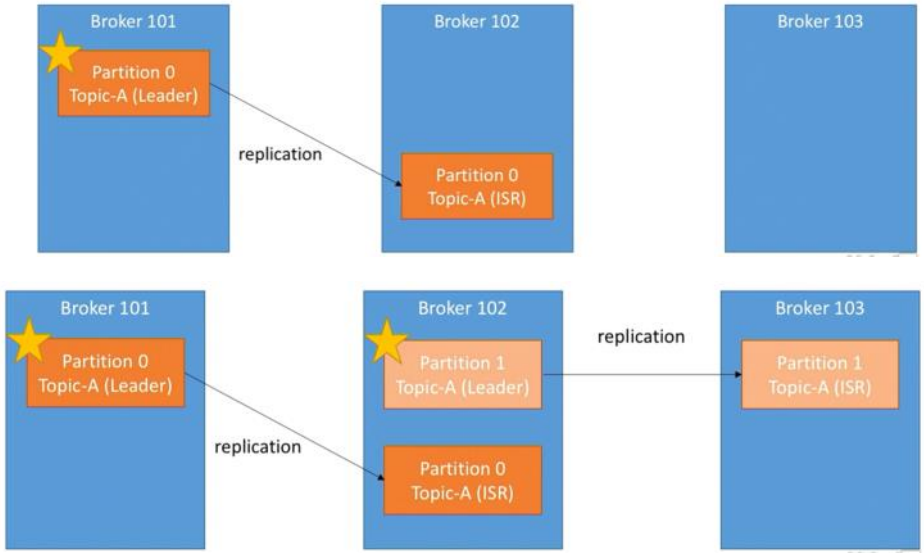
- Благодаря репликации блоков данных на различных серверах больше не нужно бояться, что отказ дисков или даже сервера в целом приведет к простою в работе. -
- Репликация данных критически важна для обеспечения отказоустойчивости распределенных приложений, которая, в свою очередь, необходима для их успешной работы

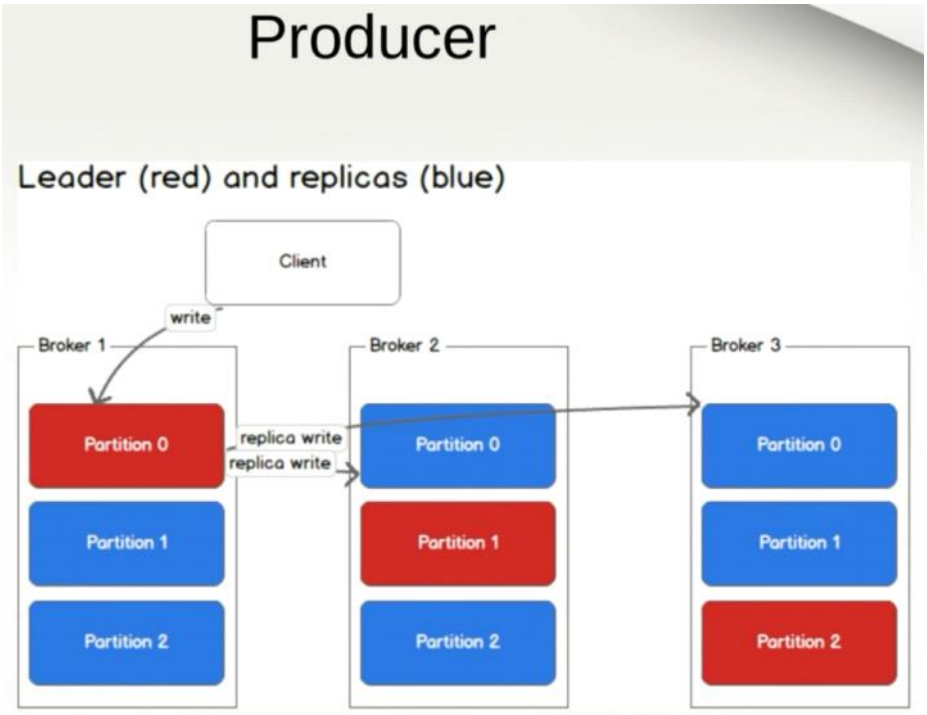
у партиций топика - только одна лидер-мастер партиция - которая сначала принимает данные (и уже затем реплицирует на другие)

это обеспечивается зукипером

- В Kafka существует понятие ведущего и ведомого брокеров: для каждой секции топика один из брокеров выбирается в качестве ведущего (leader) для остальных брокеров — ведомых (followers). Одна из главных задач ведущего брокера — назначение ведомых брокеров для репликации (replication) секций топика. Аналогично тому как Kafka распределяет секции топика по кластеру, Kafka также реплицирует секции по машинам.

- **At any time only ONE broker can be a leader for a given partition**
- **Only that leader can receive and serve data for a partition**
- The other brokers will synchronize the data
- Therefore each partition has one leader and multiple ISR (in-sync replica)





?replication factor топика говорит о том на скольких брокерах должен находится топик
? partition count

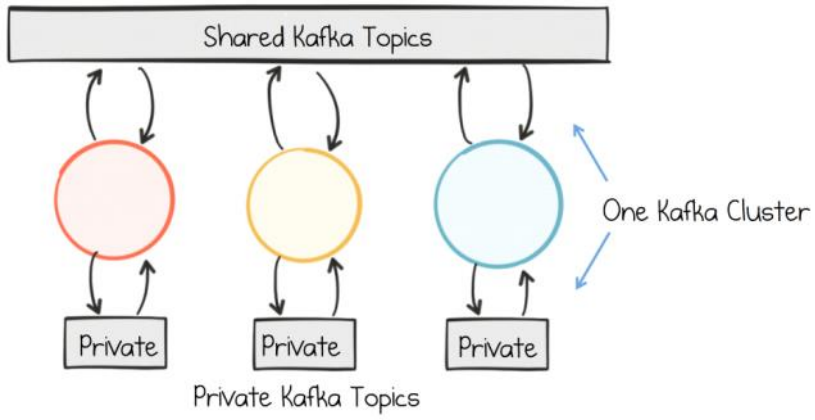
топик = по бизнес смыслу содержит схожие объекты одного DDD-агрегата

Topics are logical containers in Kafka and (usually) contain records or data of same type. Partitions are used to parallelized work with topics. Thus one topic has 1…n partitions.

Segregate Public and Private Topics (на уровне kafka security: authorization)

- инкапсуляция топиков
- одни топики для обмена ивентами между микросервисами, а другие топики для внутренне работы микросервиса(куда не должны лазить другие микросервисы)

- When using Kafka for Event Sourcing or Stream Processing, in the same cluster through which different services communicate, we typically want to segregate private, internal topics from shared, business topics. При использовании Kafka для поиска событий или потоковой обработки в том же кластере, через который взаимодействуют различные службы, мы обычно хотим отделить частные внутренние темы от общих бизнес-тем.
- Some teams prefer to do this by convention, but a stricter segregation can be applied using the authorization interface. Essentially you assign read/write permissions, for your internal topics, only to the services that own them. This can be implemented through simple runtime validation, or alternatively fully secured via TLS or SASL. Некоторые команды предпочитают делать это по соглашению, но с помощью интерфейса авторизации можно применить более строгую сегрегацию . По сути, вы назначаете разрешения на чтение / запись для своих внутренних тем только службам, которым они принадлежат. Это может быть реализовано посредством простой проверки во время выполнения или, в качестве альтернативы, полностью защищено с помощью TLS или SASL.



partition == хэширование == параллелизм

29 ноября 2020 г. 12:08

partition implies grouping,

- Термин «секционирование» подразумевает группировку, но я имею в виду группировку не по одинаковым ключам, а по ключам с одним хеш-кодом.
- Для разбиения данных на секции по ключу можно применить следующую формулу
    - `int partition = key.hashCode() % numberOfPartitions;`

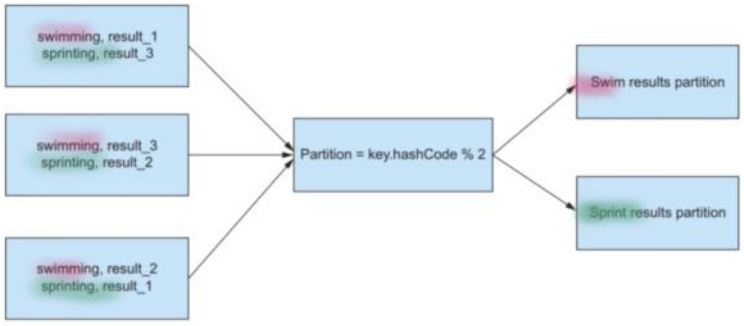
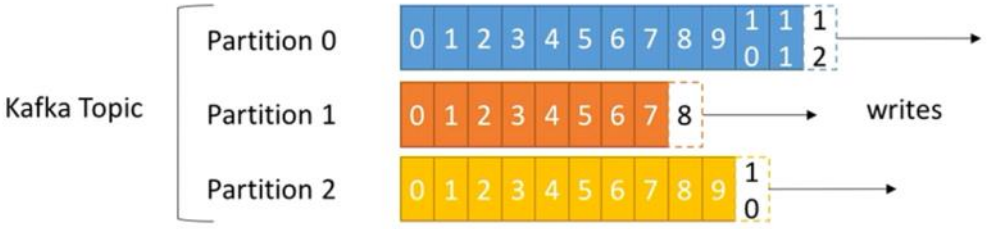


Figure 1.2 Grouping records by key on partitions. Even though the records start out on separate servers, they end up in the appropriate partitions.

offset имеет значение только для **конкретной** партиции



- Offset only have a meaning for a specific partition.
    - E.g. offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1

у каждой партиции много офсетов (по одному на каждую читающую consumer group, на самом деле получается что по одному на каждого консьюмера(своего типа функциональности))

- Kafka stores the offsets at which a consumer group has been reading
- The offsets committed live in a Kafka topic named __consumer_offsets
- When a consumer in a group has processed data received from Kafka, it should be committing the offsets
- If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets!

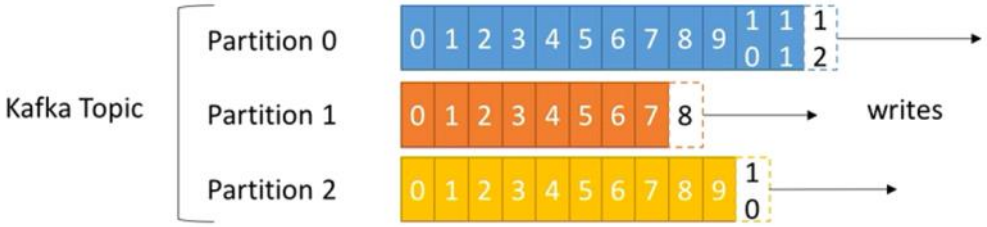
если инстанс микросервиса (в консьюмер группе где у каждого консьюмера своя партиция) умер , то ново-поднявшийся инстанс подхватит offset умершего микросервиса

..

??как настроить консьюмер на конкретную партицию

порядок гарантирован только внутри **конкретной** партиции ~~(а не топика)~~
на одну партицию должен быть один продсьюсер(порядок не гарантирован если в одну партицию будут писать несколько продьюсеров)

-

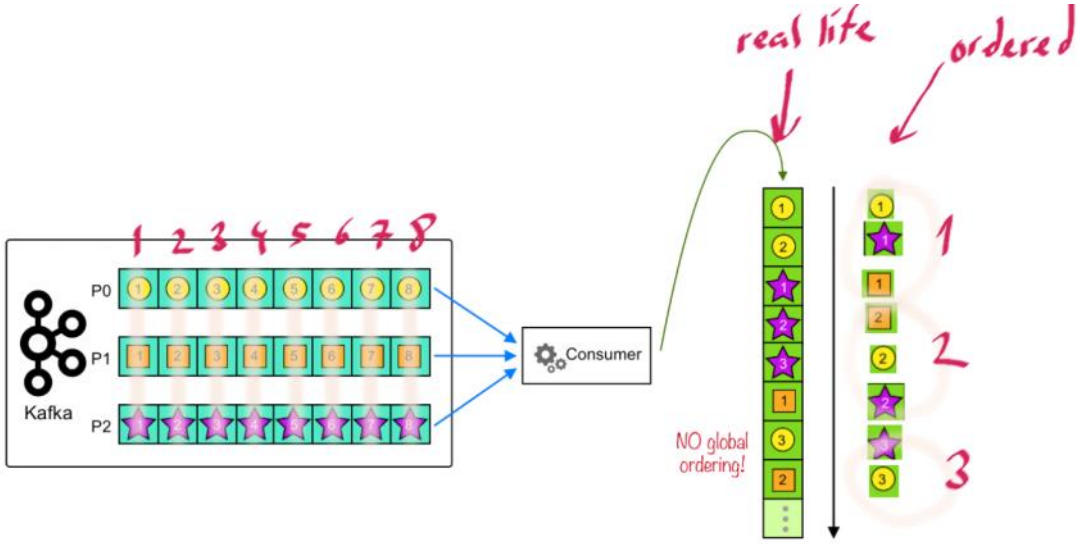


- Offset only have a meaning for a specific partition.
  - E.g. offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1
- Order is guaranteed only within a partition (not across partitions)

порядок на уровне топика сложно сделать

- на практике НЕ встречаются случаи, где порядок нужен **именно на уровне всего топика** а не партиции,
- (1) но если вдруг такое понадобится то нужно просто сделать **топик из всего одной партиции** но это возможно только когда нет большой нагрузки
- (2) можно засосать все данные из топика в свою прогу, затем переупорядочить их и далее направить дальше (но они уже будут упорядоченными)

- Иногда упорядочивания на основе ключей недостаточно, и требуется глобальное упорядочение. Это часто возникает, когда вы перех одите с устаревших систем обмена сообщениями, где глобальное упорядочение предполагалось исходной системой. Для поддержания глобально го порядка используйте **single partition topic .** Пропускная способность будет ограничена пропускной способностью одной машины, но обычно этого достаточно для большинства случаев использования этого типа.
- Сохранение упорядочения, ограниченного шардом, значительно упрощает работу, поскольку нет необходимости поддерживать и совмес тно использовать глобальное упорядочение во всем кластере.



- **Question:** how can you preserve message order if the application requires it?

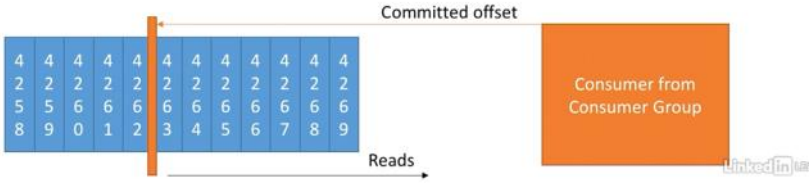
- If there are multiple Partitions, you **will not get total ordering across all messages** when reading data

Answer:

- Be selective when choosing the message key
  - Messages with the same key are delivered to the Consumer in order
- Use a single Producer
  - There are no guarantees that different producers writing to the same topic with the same key will send in order, due to batching, CPU, etc
- Ensure that the application calling the Kafka Producer is preserving message order as well
  - Send synchronously to the producer, e.g. wait till the first message is received before sending the second
- Make the applications responsible for ordering outside of Kafka
  - Have producers include information that can be used by the consumers to reorder the messages after receipt

смысл оффсетов в том, чтобы упавший косьюмер мог прочитать свои данные с начала (те со своего офсета)

- Kafka stores the offsets at which a consumer group has been reading
- The offsets committed live in a Kafka topic named __consumer_offsets
- When a consumer in a group has processed data received from Kafka, it should be committing the offsets
- If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets!



мы сами задаем количество партиций

- и то какие данные будут в конкретной партиции

те то по какому условию будут нарезаться общие данные в ту или иную партицию

тк на самом деле это простой БД шардинг, то можно нарезать по буквам алфавита например

хотя смысл партиций в распараллеливании работы, но мы можем разбивать не хаотично а по бизнес смыслу (например чтобы все опреации относящиеся к одному клиенту попадали в одну партицию и их потом обрабатывал один и тот же потребитель)

To parallelize work and thus increase the throughput Kafka can split a single topic into many partitions. The messages of the topic will then be split between the partitions. The default algorithm used to decide to which topic partition a message goes uses the hash code of the message key. A partition is handled in its entirety by a single Kafka broker. A partition can be viewed as a "log"

иметь сообщения с одинаковым ключем -это нормальная практика

аналогично тому как с одной и той же записью в БД происходят разные CRUD операции (в кафке это просто отдельные сообщения C R U D)

- например можно сделать компакшн
- например можно последовательно(как бы однопоточно) обработать операции относящиеся к одному и тому же клиенту (клиент как клю ч а операция и данные как значение сообщения) тк кафка гарантирует порядок сообщений в одной партиции

- те кафка топик(а на самом деле compacted kafka table) можно представить аналогией - таблицей БД
- те ключ кафки можно сравнить с PK записи в таблице БД

микросервисы могут использовать компакшн для быстрого восстановления своего состояния (те для поднятия нового инстанса, который должен прочитать все записи event sourcing/event store)

- Compacted logs are useful for restoring state after a crash or system failure. Kafka log compaction also allows downstream co nsumers to restore their state from a log compacted topic.
- They are useful for in-memory services, persistent data stores, reloading a cache, etc. An important use case of data streams is to log changes to k eyed, mutable data changes to a database table or changes to object in in-memory microservice.
- Log compaction is a granular retention mechanism that retains the last update for each key. A log compacted topic log contain s a full snapshot of final record values for every record key not just the recently changed keys.

партиционирование в кафке это шардинг в БД

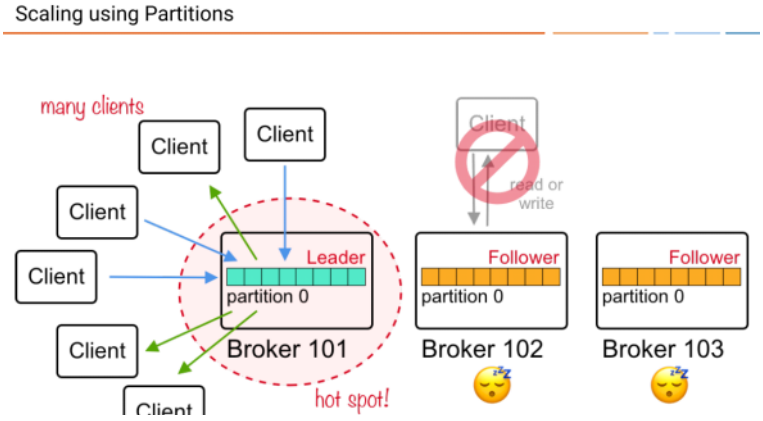
те шардировать можно аналогично как в мире БД те по буквам алфавита или географически

по дефолту в key-hashing, сообщения с одним и тем же ключем будут направляться на одну и туже партицию

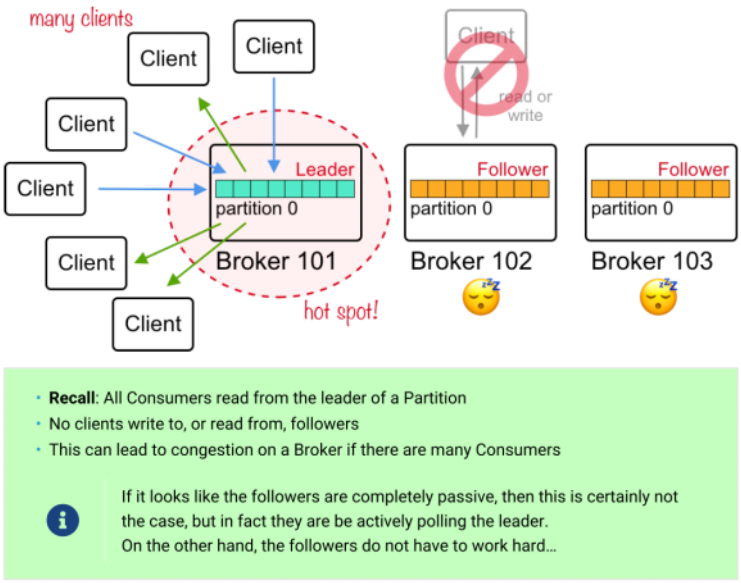
- но если сделать ребаланс то текущая схема сломается и выстроется новая но тоже устойчивая схема

message key как средство распараллеливания/distribution

если бы не было партиционирования то все клиенты присосались бы к одной ноде (так как из реплик не читаем)



Scaling using Partitions



- **Recall**: All Consumers read from the leader of a Partition
- No clients write to, or read from, followers
- This can lead to congestion on a Broker if there are many Consumers

If it looks like the followers are completely passive, then this is certainly not the case, but in fact they are be actively polling the leader.
On the other hand, the followers do not have to work hard...

Важно понимать, что партиция на самом деле является единицей параллелизма для производителей и потребителей Kafka

- На стороне производителя разделы позволяют писать сообщения параллельно. Если сообщение публикуется с ключом, то по умолчанию производитель хеширует данный ключ, чтобы определить целевой раздел. Это гарантирует, что все сообщения с одним и тем же ключом будут отправлены в о дин и тот же раздел. Кроме того, у потребителя будет гарантия доставки сообщений для этого раздела.
- Со стороны потребителя количество разделов для темы ограничивает максимальное количество активных потребителей в группе потре бителей. Группа потребителей - это механизм, предоставляемый Kafka для группирования нескольких клиентов -потребителей в одну логическую группу, чтобы сбалансировать нагрузку на потребление разделов.Kafka гарантирует, что раздел темы будет назначен только одному потребителю в группе

broker == кафка сервер

29 ноября 2020 г. 12:11

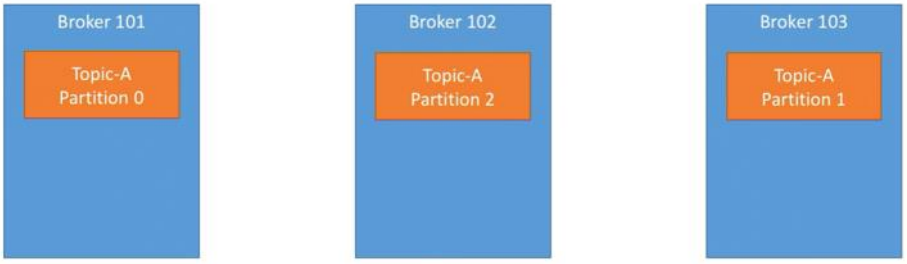
каждый брокер идентифицируется своим ID

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker is identified with its ID (integer)
- Each broker contains certain topic partitions
- After connecting to any broker (called a bootstrap broker), you will be connected to the entire cluster
- A good number to get started is 3 brokers, but some big clusters have over 100 brokers
- In these examples we choose to number brokers starting at 100 (arbitrary)

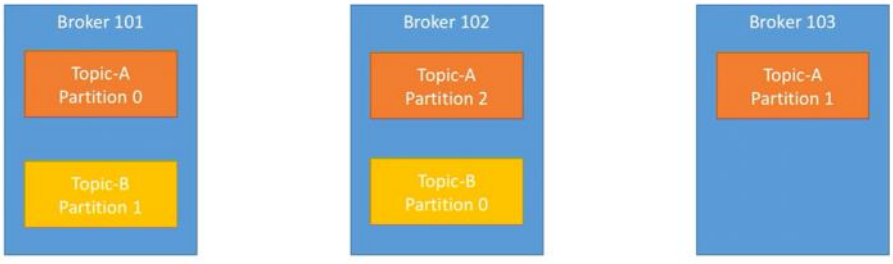
брокер хранит партиции

- **конкретная** партиция привязана к **конкретной** ноде, и чтобы это изменить надо сделать reassign

- Example of **Topic-A** with **3 partitions**



- Example of **Topic-A** with **3 partitions**
- Example of **Topic-B** with **2 partitions**

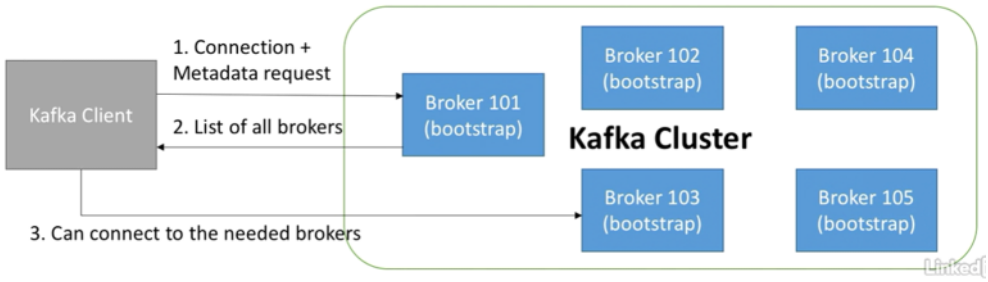


- Note: Data is distributed and Broker 103 doesn't have any Topic B data

bootstrap server (kafka broker discovery) - присоеденившись к одному брокеру ты конектишся ко всему кластеру

- это обеспечивается зукипером
- те можно коннектиться к любой ноде кафки

- Every Kafka broker is also called a “bootstrap server”
- That means that **you only need to connect to one broker**, and you will be connected to the entire cluster.
- Each broker knows about all brokers, topics and partitions (metadata)



в итоге надо настроить кафку так чтобы сообщения были **равномерно распределены** по кафке

кафка стремится сделать так чтобы партиции раскидывались по разным брокерам

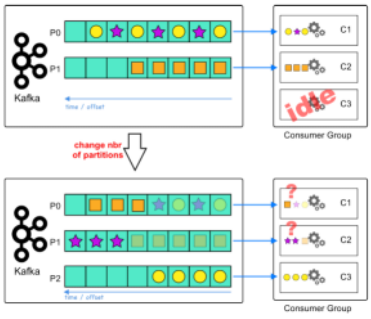
- ребалансировка автоматическая или ручная

число партиций зависит от пиковой нагрузки

больше нагрузка -> больше кафка нод -> больше партиций -> больше потребителей

- но всетки надо иметь возможность масштабировать партиции независимо от потребителей, а не все время делать по одному консьюмеру на одну партицию

число партиций = число нод *2

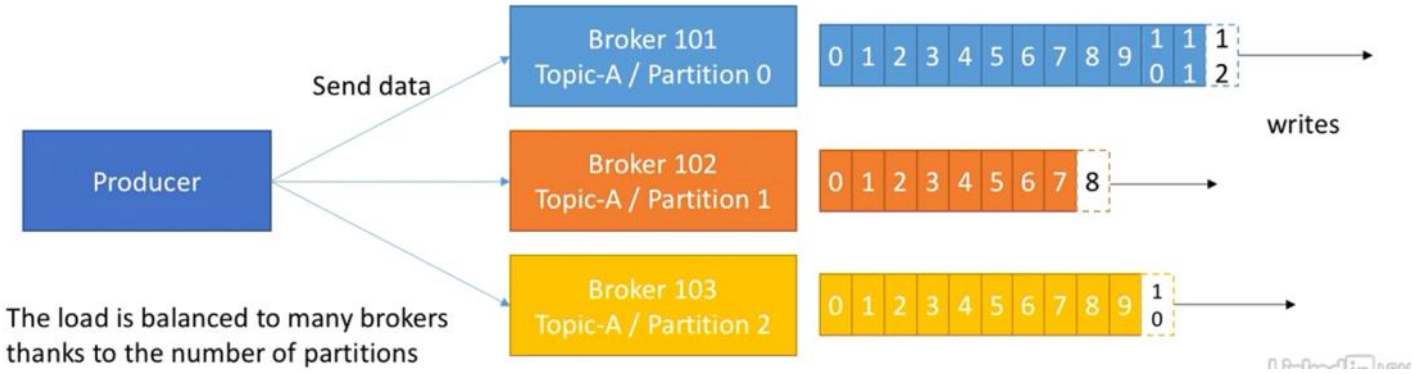


producer

29 ноября 2020 г. 12:28

продьюсеры не заботятся о брокерах и партициях

- Producers write data to topics (which is made of partitions)
- Producers automatically know to which broker and partition to write to
- In case of Broker failures, Producers will automatically recover



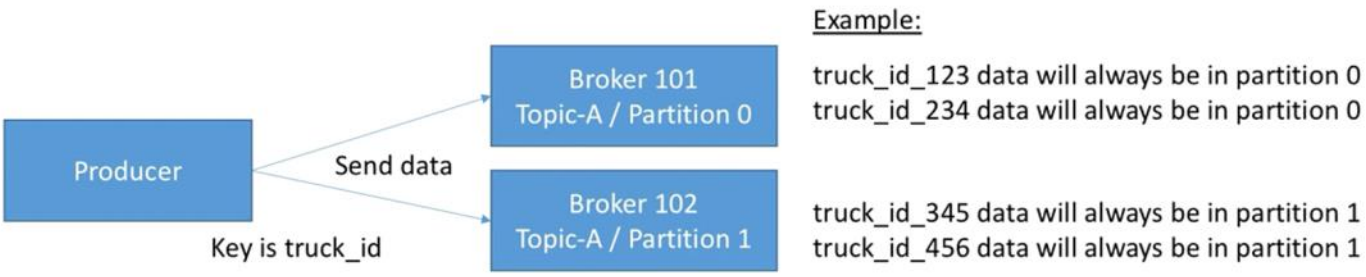
producer acknowledgment - уровень надежности

- Producers can choose to receive acknowledgment of data writes:
  - acks=0: Producer won't wait for acknowledgment (possible data loss)
  - acks=1: Producer will wait for leader acknowledgment (limited data loss)
  - acks=all: Leader + replicas acknowledgment (no data loss)

партиции можно сопоставить ключам сообщений

- key hashing
- key to particions mechanism

- Producers can choose to send a **key** with the message (string, number, etc..)
- If key=null, data is sent round robin (broker 101 then 102 then 103…)
- If a key is sent, then all messages for that key will always go to the same partition
- A key is basically sent if you need message ordering for a specific field (ex: truck_id)



ProducerRecord - можно даже сообщение явно направить в нужную партицию (минуя автоматическое назначение по хэшу ключа сообщения)

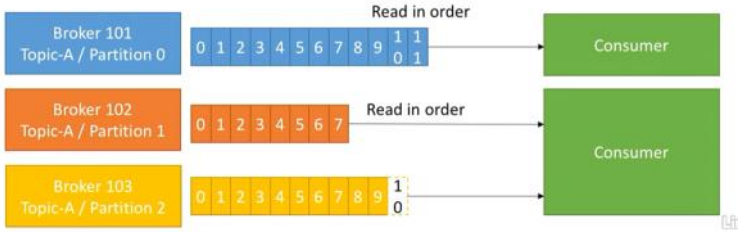
- The partition can be explicitly provided in the ProducerRecord object via the overloaded ProducerRecord constructor. In this case, the producer always uses this partition.

consumer

29 ноября 2020 г. 12:36

консьюмер сопоставлен конкретному брокеру

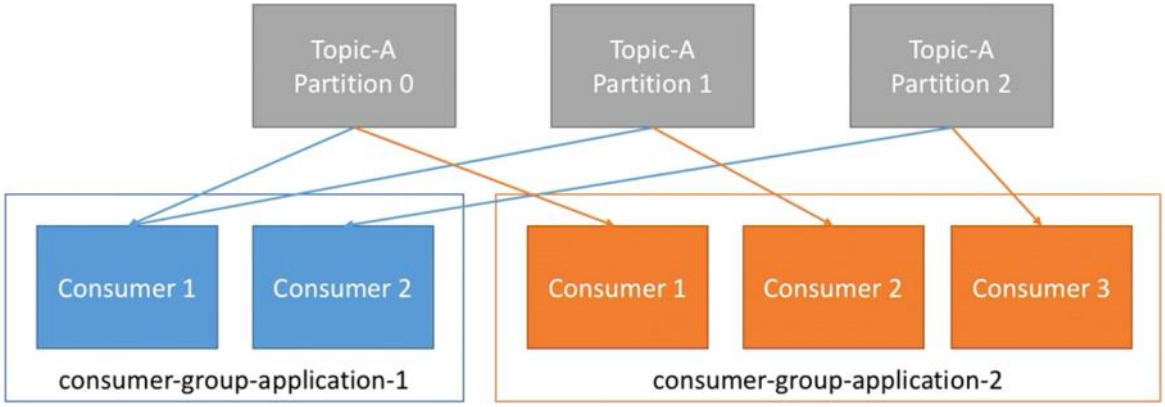
- Consumers read data from a topic (identified by name)
- Consumers know which broker to read from
- In case of broker failures, consumers know how to recover
- Data is read in order **within each partitions**



consumer group - можно жестко сопоставить партицию консьюмеру (внутри группы)

-

- Consumers read data in consumer groups
- Each consumer within a group reads from exclusive partitions
- If you have more consumers than partitions, some consumers will be inactive



?? те если консьюмеров столько же сколько партиций, то каждый консьюмер автоматически получит одну только его партицию (?? проблема только во временном падении консьюмера тогда его сообщения будут **сразу** перенаправлены на другой живой консьюмер, а не будет ждать пока поднимется его консьюмер)

?? а может свойство выставить чтобы ребаланс был медленным
https://jaceklaskowski.gitbooks.io/apache-kafka/content/kafka-consumer-internals-ConsumerCoordinator.html
https://learning.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html

SESSION.TIMEOUT.MS

The amount of time a consumer can be out of contact with the brokers while still considered alive defaults to 10 seconds. If more than `session.timeout.ms` passes without the consumer sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`. `heartbeat.interval.ms` controls how frequently the `KafkaConsumer poll()` method will send a heartbeat to the group coordinator, whereas `session.timeout.ms` controls how long a consumer can go without sending a heartbeat. Therefore, those two properties are typically modified together—`heartbeat.interval.ms` must be lower than `session.timeout.ms`, and is usually set to one-third of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than the default will allow consumer groups to detect and recover from failure sooner, but may also cause unwanted rebalances as a result of consumers taking longer to complete the poll loop or garbage collection. Setting `session.timeout.ms` higher will reduce the chance of accidental rebalance, but also means it will take longer to detect a real failure.

(1) partitions are assigned **automatically** to consumers and are rebalanced **automatically** when consumers are added or removed from the group

- RoundRobin не гарантирует порядка сообщений, а Range-гарантирует

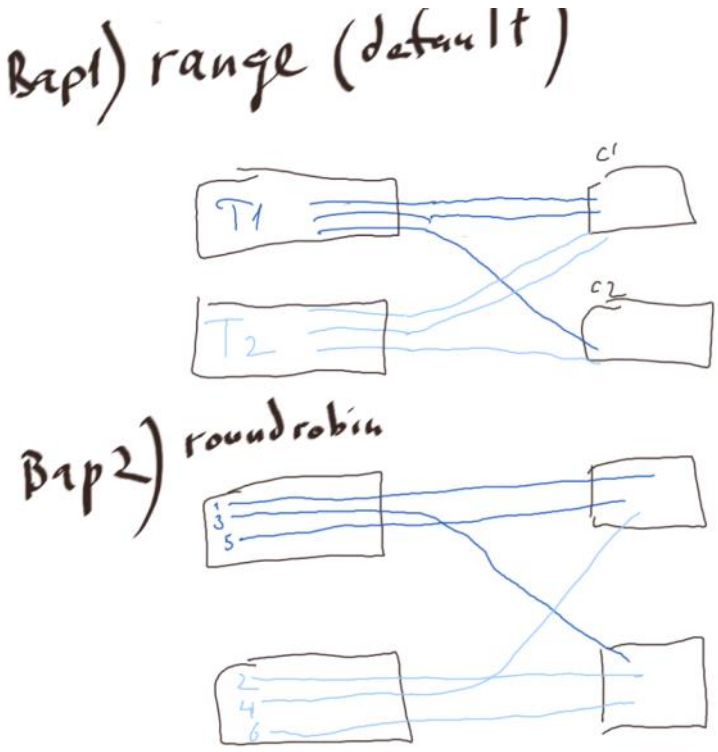
partition.assignment.strategy = org.apache.kafka.clients.consumer.RoundRobinAssignor
https://learning.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html

Range

Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2, and each of the topics has three partitions, then C1 will be assigned partitions 0 and 1 from topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has an uneven number of partitions and the assignment is done for each topic independently, the first consumer ends up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

RoundRobin

Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described previously used RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1 and partition 1 from topic T2. C2 would have partition 1 from topic T1 and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most 1 partition difference).



(2) standalone consumer (Use a Consumer Without a Group) например когда конкретный консьюмер должен читать только определенную партицию в топике

- тогда консьюмер не подписывается на топик а сразу присваивается партиции
- тогда я вроде как вручную должен поддерживать одинаковое число партиций и консьюмеров
- ??? а может не использовать это, а использовать всетки автоматический ребалан и прочее

- https://learning.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html
- single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic.
- When you know exactly which partitions the consumer should read, you don't subscribe to a topic—instead, you assign yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group), or assign itself partitions, but not both at the same time.

Standalone Consumer: Why and How to Use a Consumer Without a Group

So far, we have discussed consumer groups, which are where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case, there is no reason for groups or rebalances—just assign the consumer-specific topic and/or partitions, consume messages, and commit offsets on occasion.

When you know exactly which partitions the consumer should read, you don't *subscribe* to a topic—instead, you *assign* yourself a few partitions. A consumer can either subscribe to topics (and be part of a consumer group), or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, Strin                = consumer.poll(1000);

        for (ConsumerRecord<String, S                ord: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

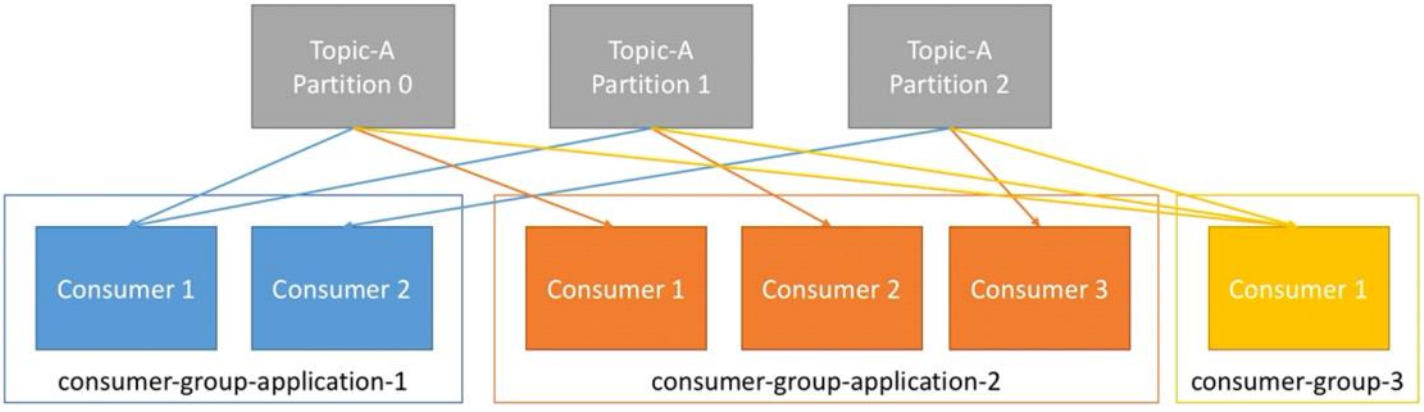
Copy
Add
Highlight
Add
Note

❶ We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.

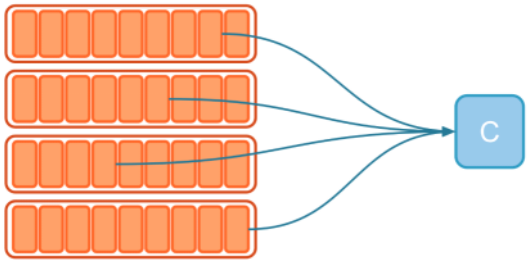
❷ Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply by bouncing the application whenever partitions are added.

пример 1-консьюмера в группе (тк он один то он потребляет все партиции)



Consuming from Kafka - Single Consumer



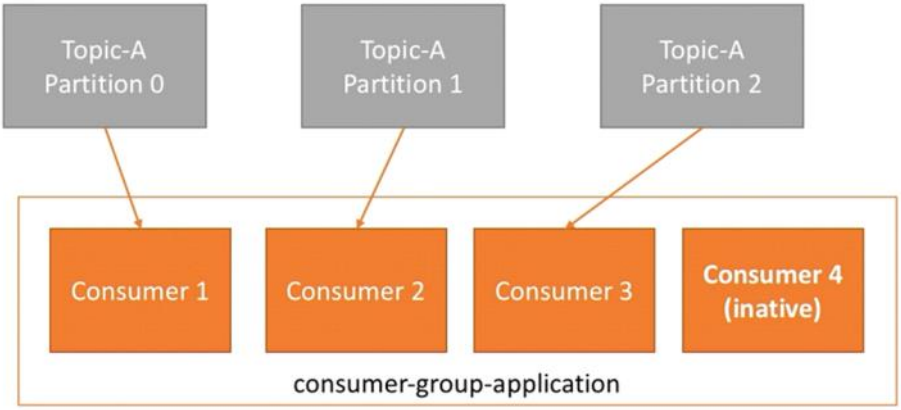
- group coordinator
- consumer coordinator

- *Note: Consumers will automatically use a GroupCoordinator and a ConsumerCoordinator to assign a consumers to a partition*

пример консьюмеров в группе где их больше чем партиций (то один лишний консьюмер за бортом)

- это является самым большим ограничением для масштабирования, то дальше уже масштабировать независимые консьюмеры не можем и нужно создавать больше партиций
- не могут два консьюмера в одной и той же группе читать одну и ту же партицию тк тогда получится что одна и таже инфа обработается два раза
- The number of useful Consumers in a Consumer Group is constrained by the number of Partitions on the Topic. Example: If you have a Topic with three partitions, and ten Consumers in a Consumer Group reading that Topic, only three Consumers will receive data. One for each of the three Partitions.
  - If there are more consumers than partitions, the additional consumers will sit idle. The idea of a hot-standby consumer is not required but can prevent performance differentials during client failures.

- If you have more consumers than partitions, some consumers will be inactive



consumers delivery semantics

- Consumers choose when to commit offsets.
- There are 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 (it won't be read again).
- At least once (usually preferred):
  - 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 your processing is idempotent (i.e. processing again the messages won't impact your systems)
- Exactly once:
  - Can be achieved for Kafka => Kafka workflows using Kafka Streams API
  - For Kafka => External System workflows, use an idempotent consumer.

концепция consumer group нужна для масштабирования/распараллеливания потребления

- но консьюмеры тоже можно запускать параллельно

- одна группа может быть сопоставлена одному топику (но внутри каждый из консьюмеров возьмет свою партицию)
- те группе инстансов одного микросервиса назначается одна и таже consumer group
- коньсьюмеры в одной констюмер-группе не делают другую-обработку на тех же самых данных, они делают ту же саму обработку но на других данных (те все консьюмеры в одной консьюмер группе имеют один и тот же java-код)

Members of a consumer group collaborate and as such allow the parallel processing of data. Consumers in a consumer group are instances of the same application with the same application ID.

концепция пиртиций нужна для масштабирования/распараллеливания

если бы не было партиционирования то все клиенты присосались бы

-

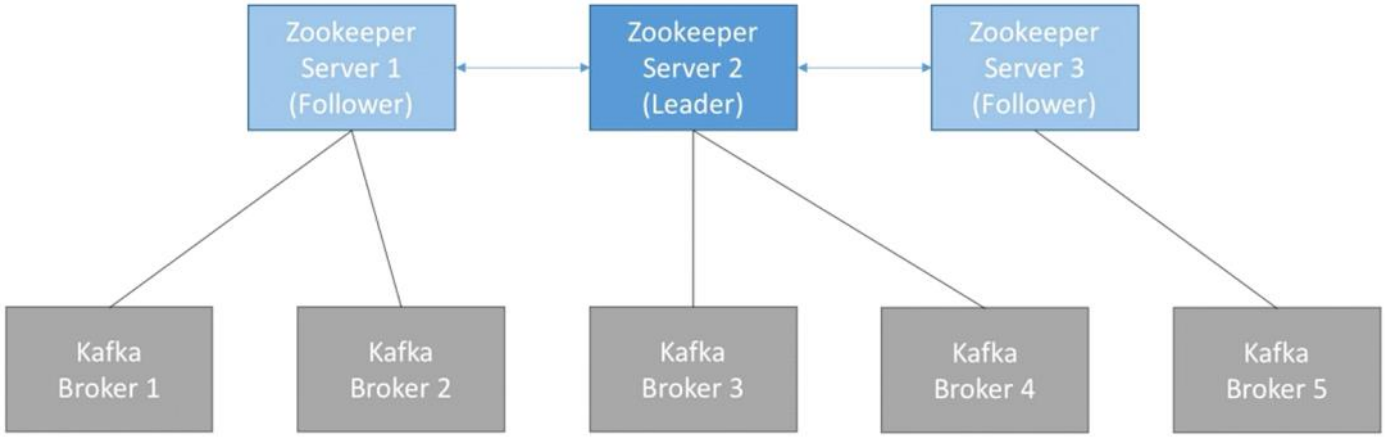
концепция consumer group = группа это как бы один логический потребитель

Любой набор потребителей, настроенный с одним и тем же group.id, образует одного логического потребителя, называемого группой потребителей. Каждый потребитель в группе отвечает за потребление из одного или нескольких разделов подписанной темы (ов).

zookeeper

29 ноября 2020 г. 15:28

- Zookeeper manages brokers (keeps a list of them)
- Zookeeper helps in performing leader election for partitions
- Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc.…)
- **Kafka can't work without Zookeeper**
- Zookeeper by design operates with an odd number of servers (3, 5, 7)
- Zookeeper has a leader (handle writes) the rest of the servers are followers (handle reads)
- (Zookeeper does NOT store consumer offsets with Kafka > v0.10)



If a Broker registration in ZK goes away, that indicates a Broker failure

- контроллер смотрит hearbeat от нод на ZK
- (А что если сам контрлллер упадет?) другие ноды через ZK мониторят самого контрллера (тк контроллер сам отправляет свой heartbeat на ZK)

Managing Broker Failures

- One Broker in cluster is assigned role of **Controller**
  - Detects Broker failure/restart via ZooKeeper
- Controller action on Broker failure
  - Selects a new leader and updates the ISR list
  - Persists the new leader and ISR list to ZooKeeper
  - Sends the new leader and ISR list changes to all Brokers
- Controller action on Broker restart
  - Sends leader and ISR list information to the restarted Broker
- Current Controller fails → another Broker is assigned role of **Controller**

The Controller is a thread off the main broker software which maintains the list of partition information, such as leaders and ISR lists. The Controller runs on exactly one Broker in the cluster at any point in time.

The Controller monitors the health of every Broker by monitoring their interaction with ZooKeeper. If a Broker registration in ZK goes away, that indicates a Broker failure. The Controller elects new leaders for partitions whose leaders are lost due to failures and then broadcasts the metadata changes to all the Brokers so that clients can contact any Broker for metadata updates. The Controller also stores this data in ZooKeeper so that if the Controller itself fails, the new Controller elected by ZooKeeper will have a set of data to start with.

If the Controller fails, ZooKeeper will assign the role to the next Broker to check in with ZooKeeper after the failure.

()

см также в книге клепмана

22 февраля 2021 г. 12:27

Multiple consumers

consumer acknowledgments (with load balancing)