

message key

12 декабря 2020 г. 15:45

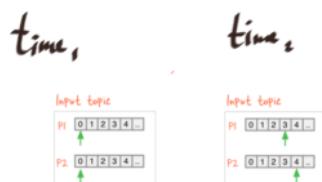
Но этот подход горизонтального масштабирования в кафке работает только при наличии логического ключа, который четко разделяет все операции:

- Stateful stream processing systems like Kafka Streams avoid remote transactions or cross-process coordination. They do this by partitioning the problem over a set of threads or processes using a chosen business key. This provides the key (no pun intended) to scaling these systems horizontally.
- So splitting (i.e., partitioning) the problem by ProductId ensures that all operations for one ProductId will be sequentially executed on the same thread. That means all iPads will be processed on one thread, all iWatches will be processed on one (potentially different) thread, and the two will require no coordination between each other to perform the critical section (Figure 15-5). The resulting operation is atomic (thanks to Kafka's transactions), can be scaled out horizontally, and requires no expensive cross-network coordination. (This is similar to the Map phase in MapReduce systems.)

у каждой партиции много оффсетов (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!

пример с одной consumer group



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

у сообщения может вообще не быть ключа, тогда стратегия будет RoundRobin

- и сообщения просто будут равномерно распределяться между партициями

иметь сообщения с одинаковым ключем -это нормальная практика аналогично тому как с одной и той же записью в БД происходят разные CRUD операций (в кафке это просто отдельные сообщения C R U D)

- например можно сделать компакшион
- например можно последовательно(как бы однопоточно) обработать операции относящиеся к одному и тому же клиенту (клиент как ключ 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 сравнивать с PK записи в таблице БД

key в сообщении может и не быть PK

- а быть только частью PK (а остальная часть PK запихнута в VALUE сообщения)
- те только использоваться для партиционирования

guaranteed order of messages at partition level - порядок сообщений внутри партиции очень важен для того чтобы например, симулировать однопоточную обработку операций над одним и тем же договором (чтобы операции не пришли в перепутанном порядке)

- но порядок сообщений не гарантируется на уровне топика
- It serves as a way to divvy up processing among consumer processes while allowing local state and preserving order within the partition. We call this semantic partitioning.
- упорядочивания на уровне партиции (а не на уровне топика) обычно вполне достаточно, так как обычно требуется упорядочивание только небольших кучек информации
 - (например чтобы упорядочить поставки конкретного покупателя в том порядке в котором он их заказывал,
 - а можно наоборот распараллелить и специально отправить их в разные партиции, может они придут к нему быстрее хоть и в разном порядке
- While it often isn't the case for analytics use cases, most business systems need strong ordering guarantees. Say a customer makes several updates to their Customer Information. The order in which these are processed is going to matter, or else the latest change might be overwritten with an older value.(порядок нужен чтобы устаревшее значение не перезаписало новое- последнее)
- There are a couple of things we need to consider to ensure strong ordering guarantees. The first is that messages that require relative ordering need to be sent to the same partition. This is managed for you, you supply the same key for all messages that require a relative order. So our stream of Customer Information updates would use the CustomerId, as their sharding key, so that all messages for the same Customer would be routed to the same partition, and hence be strongly ordered.

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 consumer 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 keyed, 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 contains a full snapshot of final record values for every record key not just the recently changed keys.

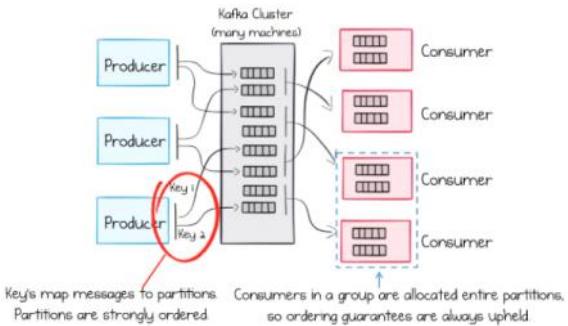
по дефолту в key-hashing, сообщения с одним и тем же ключем будут направляться на одну и ту же партицию

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

хэш функция

номер партиции в которую отправится сообщение == hash(key)% числоパーティций

- поэтому сообщения с одинаковым ключем и отправляются на одну и ту же партицию



Incoming messages:

{foo, message data}
{bar, message data}

Message keys are used to determine which partition the message should go to. These keys are not null.

The bytes of the key are used to calculate the hash.

Partition 0 $\text{hashCode}(\text{fooBytes}) \% 2 = 0$



Once the partition is determined, the message is appended to the appropriate log.

Partition 1



$\text{hashCode}(\text{barBytes}) \% 2 = 1$

Figure 2.8 "foo" is sent to partition 0, and "bar" is sent to partition 1. You obtain the partition by hashing the bytes of the key, modulus the number of partitions.

при изменении числа партиций данные всегда будут ломаться, и если они ранее поступали на первую партицию то после они будут поступать на вторую

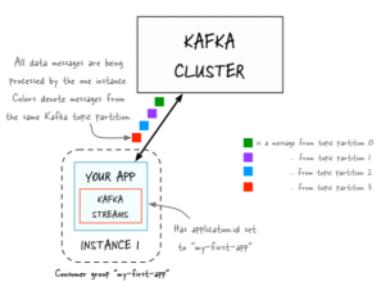
Consumer Groups: Caution When Changing Partitions



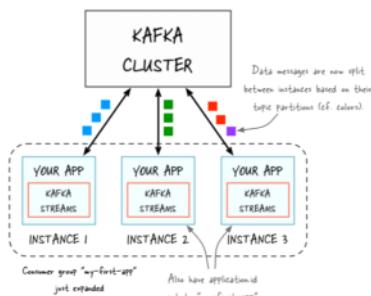
partition change

а теперь изменили
здесь
и комьюнитер будет получать
другие данные

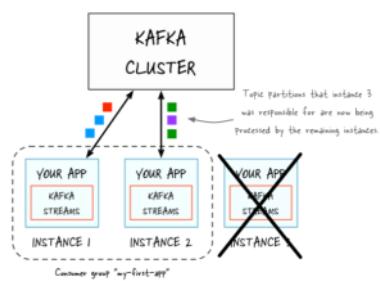
Бап1) 1-consumer



Бап2) 3 consumers



Бап3) удаление комьюнитера



при репартиционировании особенно надо быть осторожным, если коньюнитер является statefull (и выполняет например stateful-операцию агрегации данных с данными ключом)

- если например коньюнитер считал клиентов с заданным client_id, то после репартиционирования клиенты пойдут на другой инстанс-коньюнитера и он начнет считать их заново (а на предыдущем коньюнитере счетчик остановится тк прежние клиенты перестанут туда поступать)
- Recall that semantic partition works on the idea that a message will be sent to the partition determined by the formula $\text{hash}(\text{key}) \% n$, where n is the number of partitions. Changing the n number could change the output of the formula, resulting in messages with the same key being sent to different partitions. This would defeat the purpose of semantic partitioning.
- Example: Using Kafka's default Partitioner, Messages with key K1 were previously written to Partition 0 of a Topic. After repartitioning, new messages with key K1 may now go to a different Partition. Therefore, the Consumer which was reading from Partition 0 may not get those new messages, as they may be read by a new Consumer

одним из решений является - просто завести новый более большой топик и смигрировать в него данные из этого (чем пытаться расширять существующий топик)

- также все продьюсеры настроить на новый топик и старый топик тоже будет сбрасываться в новый топик
- There are strategies to mitigate this problem (e.g., migrating to a larger topic rather than just expanding the existing topic) but the best option is to plan appropriately so that you do not have to resize your topic. Selecting an appropriate number of partitions for your topic is something that will be discussed later in the course.
- In the lower part of the slide the intent to show is that the partitions may after the change contain values with the keys that were assigned prior to the change and new key values, after the change. The downstream consumers might be confused by that (that's

why I put a question mark near to the consumers C1 and C2)...

message key как одно из средств распараллеливания/distribution

The key by default is used to decide into which partition a record is written to. As a consequence all records with identical key go into the same partition. This is important in the downstream processing, since ordering is (only) guaranteed on a partition and not on a topic level!

число партиций зависит от пиковой нагрузки

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

но если надо иметь возможность параллельно получать сообщения от потребителей, а не все время ждать по кафке
потребителей на один патч

- тк степень параллелизма в кафке ограничена числом партиций

число партиций = число нод *2

?? 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 (может несработать особенно если консьюмер отсоединенется явно)

<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.

если какое-то сообщение медленно обрабатывается, то задерживает обработку следующих сообщений этого раздела-партиции

- так как у каждой партиции только один консьюмер, то если он грохнется то понадобится максимально быстрое верни на доставку другому уже работающему консьюмеру (который будет временно назначен этой партиции)

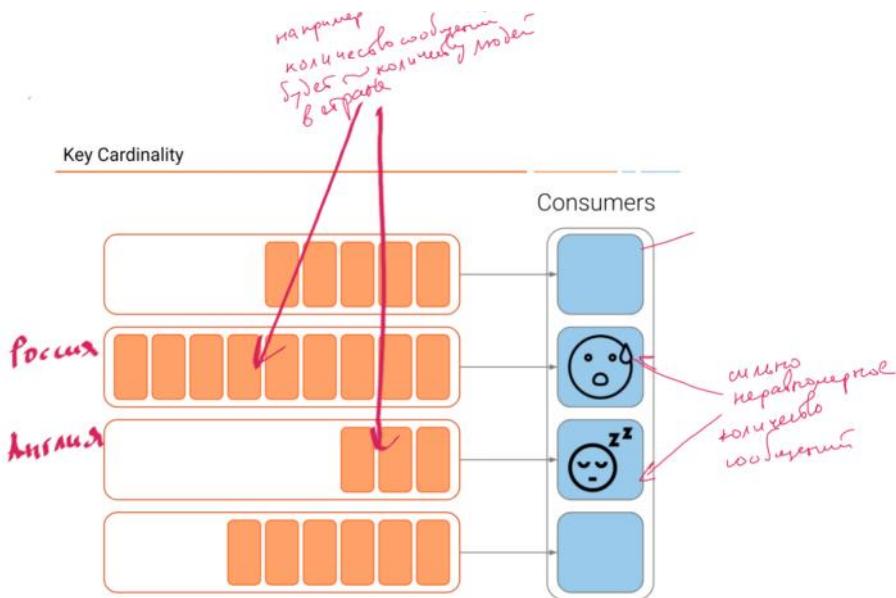
- мы сами задаем количество партиций

- и то какие данные будут в конкретной партиции (тем что мы явно придумываем каким будет ключ)

- **Cardinality** - и если мы например знаем что данные с ключем "россия" будут несизмеримо больше чем с ключем малонаселенной страны то мы должны выровнять перекос (и может быть даже написать свой собственный partitioner)

Cardinality: the number of elements in a set or other grouping, as a property of that grouping.

- Key cardinality affects the amount of work done by the individual consumer in a group. Poor key choice can lead to uneven workloads.
- Keys in Kafka don't have to be simple types like Integer, String, etc. They can be complex objects with multiple fields. For example, in some cases, a compound key can be useful, where part of the key is used for partitioning, and the rest used for grouping, sorting, etc. So, create a key that will evenly distribute groups of records around the partitions.



(a) Данные топика должны быть равномерно распределены междуパーティциями топика

Данные должны быть равномерно распределены по тематическим разделам. Например, если два тематических раздела содержат по 1 миллиону сообщений, это лучше, чем один раздел с 2 миллионами сообщений и ни одного в другом.

(б) Время обработки данных топика (workload) должны быть равномерно распределены междуパーティциями топика

Рабочая нагрузка обработки должна быть равномерно распределена по тематическим разделам. Например, если время обработки сообщений сильно различается, то лучше распределить сообщения с интенсивной обработкой по всемパーティциям, а не хранить эти сообщения в одном разделе.

? consistent hashing (есть мнение что murmur2 является consistent hashing)

<https://simplydistributed.wordpress.com/2016/12/13/kafka-partitioning/>

<https://stackoverflow.com/questions/57131345/zookeeper-kafka-and-consistent-hashing>

<https://www.sderosiaux.com/articles/2017/08/26/the-murmur3-hash-function--hashtables-bloom-filters-hyperloglog/>

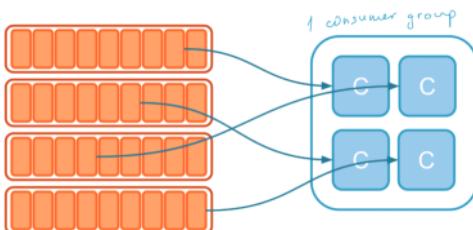
<https://en.wikipedia.org/wiki/MurmurHash>

④ концепция consumer group нужна для масштабирования/распараллеливания потребления/consumerов

(чтобы разные консьюмер-группы запускались параллельно, если нужно производить разные операции над одними и теми же данными)
(а также чтобы консьюмеры в консьюмер группе тоже могли работать параллельно, для того чтобы быстрее обработать данные из топика)

- но консьюмеры тоже можно запускать параллельно
- консьюмеры в группе могут работать параллельно (если у них разные партиции, то они могут параллельно читать из одной партиции)
- те группе инстансов одного микросервиса назначается одна и также consumer group
- consumer group предоставляют нам automatic load balancing and fault tolerance (особенно при подключении/отключении консьюмеров внутри группы)

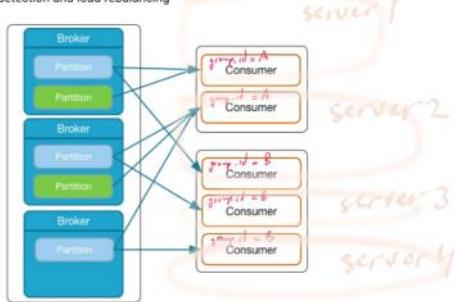
- Members of a consumer group collaborate and as such allow the parallel processing of data. Consumers in a consumer group are instances of the same application with the same application ID.
- As we said before, a consumer group transparently load balances the work among the participating consumer instances. In this image we have 4 consumers that consume a topic with 4 partitions. Thus each consumer consumes records from exactly one partition.
- A partition is always consumed as a whole by a single consumer of a consumer group. A consumer in turn can consume from 0 to many partitions of a given topic.
- Kafka Topics allow the same message to be consumed multiple times by different Consumer Groups.
- A Consumer Group binds together multiple consumers for parallelism. Members of a Consumer Group are subscribed to the same topic(s). The partitions will be divided among the members of the Consumer Group to allow the partitions to be consumed in parallel. This is useful when processing of the consumed data is CPU intensive or the individual Consumers are network-bound.
- Within a Consumer Group, a Consumer can consume from multiple Partitions, but a Partition is only assigned to one Consumer to prevent repeated data.
- Consumer Groups have built-in logic which triggers partition assignment on certain events, e.g., Consumers joining or leaving the group.



для надежности, чтобы одногруппные консьюмеры были на разных нодах (настроить афинити для pod-ов на кубере)

Group Consumption Balances Load

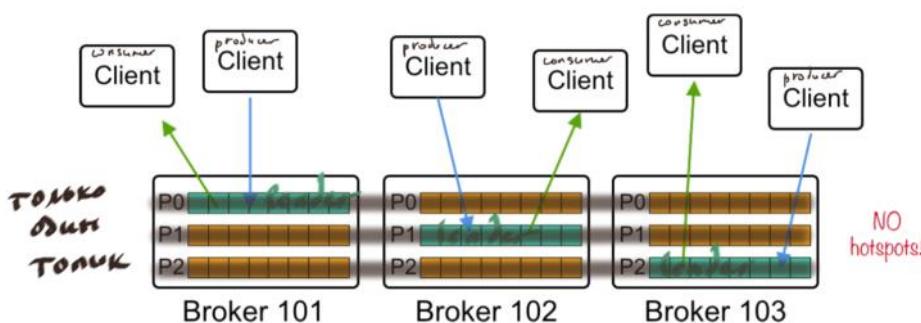
- Multiple Consumers in a Consumer Group
 - The `group.id` property is identical across all Consumers in the group
- Consumers in the group are typically on separate machines
- Automatic failure detection and load rebalancing



2) концепция партиций нужна для масштабирования/распараллеливания (чтобы клиенты равномерно распределялись между нодами)
если бы не былоpartitionирования то все клиенты присосались бы

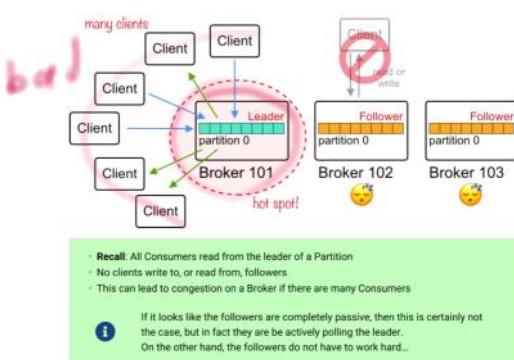
- In a multi-partition topic with replication, each partition will have its own leader. The leader of a partition serves more traffic than the followers since it handles all produce and consume (or fetch) requests. Kafka will spread the leaders across the available brokers to balance the load.
- A Broker is usually both a leader and a follower, depending on which partition you are talking about.
- Both Producers and Consumers need to know which Broker is the leader for the partition they need to contact. This is done through a metadata request. When the client starts, it does not know which Brokers are the leaders of which partitions so it issues a metadata request to a Broker (any Broker). This is why the Controller caches the partition information on all the Brokers - so any Broker can respond to a metadata request. After receiving the updated metadata, the client will know who the leader is for each partition and will communicate with the appropriate Brokers. If a Broker fails, the client will get an error and will refresh its metadata

на рисунке показаны партиции+копии одного и того же топика



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

Scaling using Partitions



global ordering - чтобы все сообщения в топике были упорядочены

- Иногда упорядочивания на основе ключей недостаточно, и требуется глобальное упорядочение. Это часто возникает, когда вы перехватываете устаревшие системы обмена сообщениями, где глобальное упорядочивание предполагалось исходной конструкцией системы. Чтобы поддерживать глобальный порядок, используйте одну тему раздела. Пропускная способность будет ограничена пропускной способностью одной машины, но обычно этого достаточно для случаев использования этого типа.

The keys used to partition the event streams must be invariant if ordering is to be guaranteed

правило: ключи ProductId и OrderId в каждом заказе должны оставаться фиксированными во всех сообщениях, относящихся к этому заказу
Если я хочу удалить/добавить товар то нужно создать новый заказ

- Ключи, используемые для разделения потоков событий, должны быть неизменными, чтобы упорядочение было гарантировано . Таким об разом, в данном конкретном случае это означает, что ключи ProductId и OrderId в каждом заказе должны оставаться фиксированными во всех сообщениях, относящихся к этому заказу. Как правило, это довольно простая вещь для у правления на уровне домена (например, принудительно, если пользователь хотите изменить приобретаемый товар, необходимо создать новый заказ)
- There are limitations to this approach, though. The keys used to partition the event streams must be invariant if ordering is to be guaranteed. So in this particular case it means the keys, ProductId and OrderId, on each order must remain fixed across all messages that relate to that order. Typically, this is a fairly easy thing to manage at a domain level (for example, by enforcing that, should a user want to change the product they are purchasing, a brand new order must be created).

в примере: один order содержит только один product (а не массив)

- те как бы денормализованные данные
- событием считается выбор одного товара юзером (а не когда он собирает всю корзину)
- это находится в согласии с законом "journal the whole fact as it arrived см " onenote:///C:\Users\trans\Qsync\vova_from_onenote_v1\SYSTEMDESIGN\KAFKA_EVENT%20DRIVEN.one#%20событие%20это%20полное-состояние%20или%20дельта§ion-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={53A24205-3590-4EC8-BBF7-97057F636F61}&end
- Но что происходит, когда пользователь отменяет одну позицию? Простое решение - просто снова записать все это в журнал как еще один агрегированный, но отмененный. Но что, если по какой-то причине заказ недоступен, а все, что мы получаем, - это одна отмененная позиция?
- Эмпирическое правило - записывать то, что вы получаете, поэтому, если поступает только одну позицию товара то запишите только ее одну. Процесс объединения со всеми остальными товарами в заказе можно выполнить потом по чтению из materialized view.
- И наоборот, разбиение событий на подсобытия по мере их появления часто не является хорошей практикой по тем же причинам.
- см также правило "для того чтобы все это заработало нужно собирать данные в реальном времени как только юзер кликает(как в реактивном программеiovании)" onenote: EVENT%20DRIVEN.one#EVENT%20DRIVEN§ion-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={1F7E4313-F7E6-4981-B2FA-C7E28EB79181}&end&base-path=C:\Users\trans\Qsync\vova_from_onenote_v1\SYSTEMDESIGN\KAFKA

```
[{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "enum",
"name": "OrderState",
"symbols": ["CREATED", "VALIDATED", "FAILED", "SHIPPED"]},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "enum",
"name": "Product",
"symbols": ["JUMPERS", "UNDERPANTS", "STOCKINGS"]},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "record",
"name": "Order",
"fields": [
    {"name": "id", "type": "string"},
    {"name": "customerId", "type": "long"},
    {"name": "state", "type": "OrderState"},
    {"name": "product", "type": "Product"},
    {"name": "quantity", "type": "int"},
    {"name": "price", "type": "double"}]},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "record",
"name": "OrderValue",
"fields": [
    {"name": "order", "type": "Order"},
    {"name": "value", "type": "double"}]}]
```

```
public enum Product
{
    GUITARS,
    AMPS,
    BOOKS
}

public class Order
{
    String id;
    String customerId;
    OrderState state;
    Product product;
    int quantity;
    double price;
}
```

message identity

4 января 2021 г. 13:45

См ...

[* optimistic concurrency control](#)

[... log compaction](#)

[event sourcing](#)

[* событие это полное-состояние или дельта](#)

см semantic lock в [* изолированность: аномалии и контрмеры](#)

см trace_id span_id [* Distributed tracing](#)

JWT transparent token [* Authorization OAuth 2.0](#)см колонки в таблице event store [event sourcing](#)

partition == хэширование == параллелизм

ordering is guaranteed, even as services fail and restart (упорядоченность сообщений с заданным ключом гарантирована даже при ребалансе консьюмеров) [ASSIGNMENT STRATEGY / CONSUMER REBALANCING](#)

в сообщении должен быть идентификатор сообщение и номер версии (которая меняется после каждого события Created, Validated...)

```
"Customer":  
  {  
    "CustomerId": "1234"  
    "Source": "ConfluentWebPortal"  
    "Version": "1"  
  }  
  ...
```

- The basic premise of identity is that it should correlate with the real world: an order has an OrderId, a payment has a PaymentId, and so on. If that entity is logically mutable (for example, an order that has several states, Created, Validated, etc., or a customer whose email address might be updated), then it should have a version identifier also

бизнесовый идентификатор сообщения не всегда легко придумать

- Хотя определение идентификатора заказа относительно очевидно, не все потоки событий имеют такое четкое понятие идентичности. Если бы у нас был поток событий, представляющий средний баланс учетной записи на регион в час, мы могли бы придумать подходящий ключ, но вы можете представить, что он был бы намного более хрупким и подверженным ошибкам.

customer_id + product_id

Существует менее распространенный случай, о котором стоит упомянуть, когда ключ (который Kafka использует для упорядочивания) полностью отличается от ключа, который вы хотите удалить. Допустим, вам нужно привязать свои заказы по ProductId . Этот выбор ключа не позволит вам удалять заказы для отдельных клиентов, поэтому описанный простой метод не сработает. Вы по-прежнему можете добиться этого, используя ключ, который является составным из двух: создайте ключ [ProductId] [CustomerId] , затем используйте пользовательский разделитель в производителе, который извлекает ProductId и разделяет только на это значение. Затем вы можете удалять сообщения, используя механизм, рассмотренный ранее, используя пару [ProductId] [CustomerId] в качестве ключа.

заметки Eberhard Wolf

log compaction (делается на основе ключа) ?event_sourcing?event_store

- Kafka представляет собой смесь системы обмена сообщениями и решения для хранения данных.
- Срок хранения по умолчанию для записей составляет семь дней, но его можно изменить. Записи также могут быть сохранены навсегда, где потребители просто хранят свое смещение. Поэтому новый потребитель(экземпляр микросервиса) может обрабатывать все записи, которые когда-либо были записаны производителем, чтобы обновить свое собственное состояние.
- Однако это означает, что Kafka должен хранить все больше и больше данных с течением времени. Однако некоторые записи со временем становятся неактуальными. Если клиент перемещался несколько раз, вы можете сохранить только последнюю информацию о последнем перемещении в виде записи в Kafka
- идея log compaction состоит в том, чтобы выборочно удалять записи, где у нас есть более свежее обновление с тем же первичным ключом. Таким образом м, журнал

гарантированно будет иметь по крайней мере последнее состояние для каждого ключа

- <https://towardsdatascience.com/log-compacted-topics-in-apache-kafka-b1aa1e4665a7>

• Чтобы упростить это описание, Kafka удаляет все старые записи, когда в журнале разделов есть более новая его версия с тем же ключом.

- Log compaction is a possibility to delete superfluous records from a topic. The Kafka documentation explains this feature. To activate log compaction, it has to be switched on when the topic is generated. See also <https://hub.docker.com/r/wurstmeister/kafka/>. Change the example in such a way that log compaction is activated.
- <https://kafka.apache.org/documentation/#compaction>

All records with the same key are removed, except for the last one.

- Поэтому выбор ключа очень важен и должен рассматриваться с точки зрения логики домена, чтобы все соответствующие записи оставались доступными после сжатия журнала
- Сжатие журнала для темы заказа удалит все события с ключом, updated42 кроме самого последнего. В результате в Kafka остается доступным только самое последнее обновление заказа.

вариант с partitioner (номер_заказа+create, номер_заказа + update)

- record key- это идентификатор заказа с расширением created, например 1created
- Просто идентификатор заказа будет недостаточно. В случае сжатия журнала все записи с одинаковым ключом удаляются, кроме последней записи. Для одного заказа могут быть разные записи.
- Одна запись может быть результатом генерации нового заказа 1created, а другие записи могут быть результатами разных обновлений 1updated. Таким образом, ключ должен содержать больше, чем идентификатор заказа, чтобы хранить все записи, принадлежащие заказу, во время сжатия журнала. Когда ключ соответствует идентификатору заказа, после сжатия журнала останется только последняя запись.
- Однако этот подход имеет тот недостаток, что записи, принадлежащие одному заказу, могут оказаться в разных партициях (поэтому нужен custom partitioner) и у разных потребителей, поскольку у них разные ключи. Это означает, что, например, записи для одного и того же заказа могут обрабатываться параллельно, что может вызвать ошибки.

- Экземпляр Кафка обеспечивает связь между microservices

- record key- это идентификатор заказа с расширением created, например 1created

• Просто идентификатор заказа будет недостаточно. В случае сжатия журнала все записи с одинаковым ключом удаляются, кроме последней записи. Для одного заказа могут быть разные записи.

• Одна запись может быть результатом генерации нового заказа 1created, а другие записи могут быть результатами разных обновлений 1updated. Таким образом, ключ должен содержать больше, чем идентификатор заказа, чтобы хранить все записи, принадлежащие заказу, во время сжатия журнала. Когда ключ соответствует идентификатору заказа, после сжатия журнала останется только последняя запись.

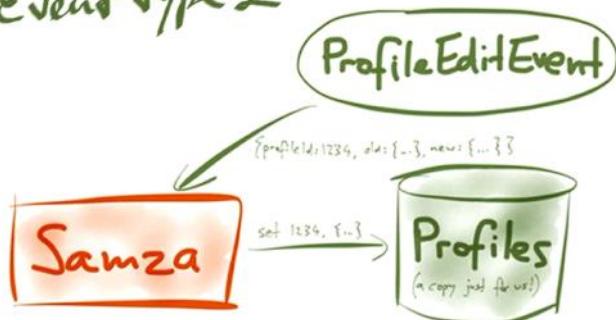
- В экземпляре базы данных каждый микросервис имеет свою собственную отдельную схему базы данных.

• Таким образом, микросервисы полностью независимы от своих схем баз данных. В то же время одного экземпляра базы данных может быть достаточно для запуска всех микросервисов.

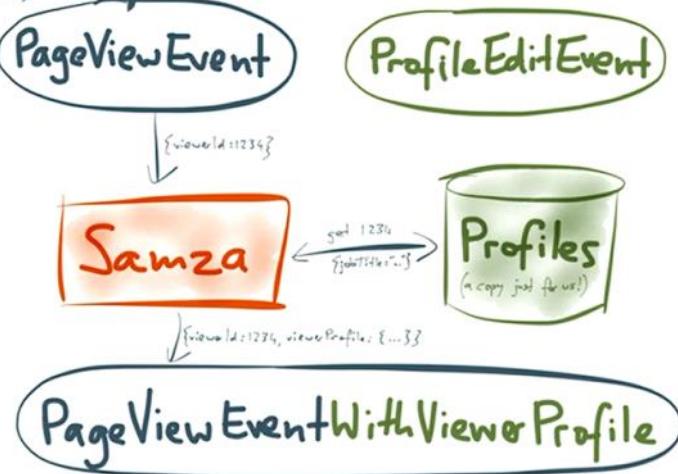
• Альтернативой может быть предоставление каждому микросервису собственного экземпляра базы данных . Однако это увеличит количество Docker-контейнеров и сделает демонстрацию более сложной

пример двух разных событий, которые в итоге должны прийти в одну партицию

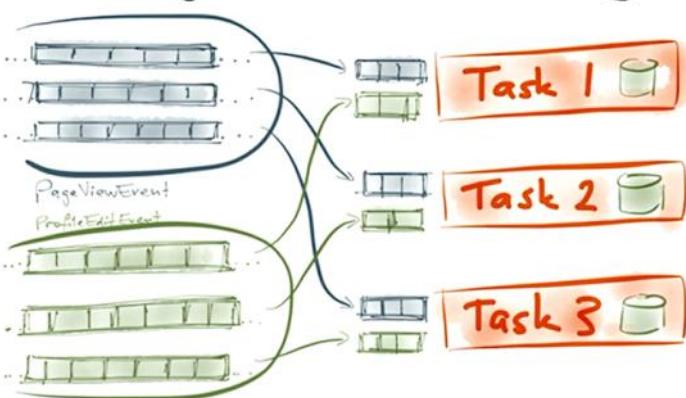
Event type 1



Event type 2



Scaling by co-partitioning



вариант без partitioner (просто номер заказа)

- Возможной альтернативой будет использование только идентификатора заказа в качестве ключа . Чтобы избежать проблемы с сжатием журнала, можно отправить полное состояние заказа вместе с каждой записью , чтобы потребитель мог восстановить свое состояние по данным в Kafka, хотя после сжатия журнала остается только последняя запись для заказа.
 - Однако для этого требуется модель данных, которая содержит все данные, которые нужны всем потребителям. Требуется много усилий для разработки такой модели данных, помимо того, что она сложна и сложна в обслуживании. Это также несколько противоречит шаблону ограниченного контекста, даже если его можно считать опубликованным языком.
-
- Записи могут быть сохранены навсегда , и потребитель может прочитать историю и восстановить ее состояние. Потребитель не обязан хранить данные локально, но может положиться на Kafka.
 - Однако это означает, что вся соответствующая информация должна храниться в записи . События обсуждали преимущества и недостатки этого подхода.
 - Если событие становится неактуальным из-за нового события, данные могут быть удалены с помощью сжатия журнала Kafka .
 - С помощью групп потребителей Kafka может гарантировать, что один и тот же потребитель обрабатывает каждую запись. Это упрощает вопросы, например, когда счет должен быть написан. В этом случае только один потребитель должен написать счет-фактуру. Было бы ошибкой, если бы несколько потребителей создавали несколько счетов одновременно.
 - Только при длительном хранении записей эволюция схемы превращается в проблему (при новой версии схемы)

partitioner

- However, this approach has the disadvantage that records belonging to one order can end up in different partitions and with different consumers because they have different keys. This means that, for example, records for the same order can be processed in parallel, which can cause errors. Однако этот подход имеет тот недостаток, что записи, относящиеся к одному заказу, могут оказаться в разных разделах и у разных потребителей, поскольку они имеют разные ключи. Это означает, что, например, записи для одного и того же заказа могут обрабатываться параллельно, что может привести к ошибкам.
- функция, которая назначает все записи для одного заказа одной партиции. Партиция обрабатывается одним потребителем, и последовательность сообщений внутри партиции гарантируется. Таким образом, гарантируется, что все сообщения, расположенные в одной партиции для одного заказа, обрабатываются одним и тем же потребителем в правильной последовательности.
- Эта функция называется partitioner. Следовательно, можно написать собственный код для распределения записей по партициям. Это позволяет производителю записывать все записи, которые принадлежат друг другу с точки зрения домена, в один и тот же партицию и обрабатывать их одним и тем же потребителем, хотя они имеют разные ключи (например если у них один и тот же префикс 1 created 1 updated).

события - это информация о событии, которое уже произошло

- я отправляю событие другим сервисам уже после того как действие произошло отправить событие после фактического действия .
- producer сначала обрабатывает заказ, а затем информирует другие микросервисы событием заказа. В этом случае, тем не менее, производитель может изменить данные в базе данных и не отправлять событие, если, например, произойдет сбой до отправки события.
- только если само действие всегда откладывается на заданный промежуток времени, хотя событие можно считать произошедшим и отослать как бы ивент еще до действия. Which of the following is NOT a disadvantage of sending an event before the data has been changed? The action gets delayed .

batch mode / пакетный режим отправки сообщений

- приводит к большим задержкам, так как сообщения должны накопиться в пакет. latency is higher; a change can be found in Kafka only after the next batch has been written.
- пакетный режим означает сначала накопить события в локальной БД а затем махом их отправить. to collect the events in a local database and to send them in a batch.
- In this case, writing the changed data and generating the data for the event in the database can take place in one transaction. The transaction can ensure that either the data is changed, and an event is created in the database, or neither takes place.

ASSIGNMENT STRATEGY / CONSUMER REBALANCING

13 декабря 2020 г. 12:16

group coordination protocol / consumer group protocol / consumer group coordination / load balancing (механизм чтобы партии равномерно распределялись между консьюмерами) / fault tolerance (механизм чтобы сообщения в партиях не скапливались)

partition.assignment.strategy задается в конфигурации консьюмера (и относится к распределению внутри консьюмер группы)

- каждая consumer-group может использовать свою собственную стратегию ребаланса

- The Consumer Group is managed by a process called a **Group Coordinator**. Like the Controller, it is a thread from the main Broker software. Unlike the Controller, there are typically multiple Coordinators running in the Cluster at any given time.
- The main tasks of the Group coordinator are handling the consumer offsets, managing the group membership, and coordinating partition assignments and rebalances.
- While the Group Coordinator manages the partition assignment, **it does not actually create the mapping of partitions to Consumers**. To avoid putting unnecessary calculations on the Brokers, the Coordinator elects one of the Consumers in the group to be the Group Leader. Additionally, delegating partition assignment to the Consumer group leader **allows different groups to utilize different partition assignment strategies**. A single Group Coordinator could be coordinating many consumer groups, so performing partition assignment there would require all groups managed by that coordinator to use the same strategy.
- When the Coordinator detects that a rebalance is required, it sends the current list of partitions and Consumers to the Group Leader. The Group Leader generates the map and sends it back to the Coordinator, who then distributes it to the members of the group.

Partition Assignment within a Consumer Group

- Partitions are 'assigned' to Consumers
 - A single Partition is consumed by only one Consumer in any given Consumer Group
 - i.e., you are guaranteed that messages with the same key will go to the same Consumer
 - Unless you change the number of partitions (see later)
 - **partition.assignment.strategy** in the Consumer configuration

1) при ребалансировке особенно надо быть осторожным, если консьюмер является **stateful** (и выполняет например **stateful-operation** агрегации данных с данным ключем)

- если например консьюмер считал клиентов с заданным client_id, то после ребалансировки клиенты пойдут на другой инстанс-консьюмера и он начнет считать их заново (а на предыдущем консьюмере счетчик остановится тк прежние клиенты перестанут туда поступать)
 - state management - даже если новый инстанс получит своих клиентов назад, то надо как то подумать о передаче старого -уже-накопленного-результат-счетчика от умершего инстанса к новорожденному
- Как видите, Потребитель 2 по какой-то причине не работает и либо пропускает опрос, либо запускает тайм-аут сеанса. Координатор группы удаляет его из группы и запускает так называемое ребалансирование. Ребалансировка - это механизм, который пытается равномерно распределить (балансировать) рабочую нагрузку между всеми доступными членами группы потребителей. В этом случае, поскольку Потребитель 2 покинул группу, при ребалансировке его ранее принадлежавшие разделя назначаются другим активным членам группы. Итак, как видите, потеря приложения-потребителя для определенного идентификатора группы не приводит к потере обработки этих разделов темы.

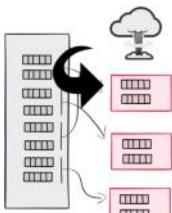
2) ? а если CRUD характер операций на клиенте-договоре: то как повлияет ребалансировка?

- CRUD операции вроде не stateful, но порядок важен
- ordering is guaranteed, even as services fail and restart (см ниже)

ordering is guaranteed, even as services fail and restart (упорядоченность сообщений с заданным ключем гарантирована даже при ребалансе консьюмеров)

- это работает из за того что одна партия может быть назначена только на одного консьюмера
 - и неважно какого, те неважно что один отвалился а другой подзватил, главное что консьюмер один

- This process actually works by assigning whole partitions to different consuming services. A strength of this approach is that a single partition can only ever be assigned to a single service instance (consumer). This is an invariant, implying that ordering is guaranteed, even as services fail and restart. На самом деле этот процесс работает путем назначения целых разделов различным потребляющим службам. Сильной стороной этого подхода является то, что один раздел может быть назначен только одному экземпляру службы (потребителю). Это инвариант, означающий, что порядок гарантирован даже в случае сбоя и перезапуска службы.
- So services inherit both high availability and load balancing, meaning they can scale out, handle unplanned outages or perform rolling restarts without service downtime. In fact Kafka releases are always backwards compatible with the previous version, so you are guaranteed to be able to release a new version without taking your system offline. Таким образом, службы наследуют как высокую доступность, так и балансировку нагрузки, что означает, что они могут масштабироваться, обрабатывать незапланированные отключения или выполнять циклический перезапуск без простой служб. Фактически, выпуски Kafka всегда обратно совместимы с предыдущей версией, поэтому вы гарантированно сможете выпустить новую версию, не отключая свою систему.
- <https://www.confluent.io/blog/apache-kafka-for-service-architectures/>



If an instance of a service dies, data is redirected and ordering guarantees are maintained

- Каждое событие, связанное с конкретным заказом, публикуется в одну и ту же партицию, которая считывается одним экземпляром потребителя. В итоге гарантируется упорядоченная обработка этих сообщений
- гарантировать, что каждое сообщение будет обработано лишь один раз и в правильном порядке
- Each event for a particular order is published to the same shard, which is read by a single consumer instance. As a result, these messages are guaranteed to be processed in order.

kafka rebalance protocol

- Если потребитель покидает группу после контролируемого завершения работы или выходит из строя, все его разделы будут автоматически переназначены другим потребителям. Таким же образом, если потребитель (повторно) присоединится к существующей группе, тогда все разделы также будут перебалансированы между членами группы.
- rebalance mechanism is actually structured around two protocols : Group Membership Protocol and Client Embedded Protocol.



'stop-the-world' rebalance - остановка всех консьюмеров на время ребалансировки
(происходит для всех стратегий кроме CooperativeStickyAssignor)

- Однако у подхода к ребалансировке по умолчанию есть недостаток. Каждый потребитель полностью отказывается от назначения тематических разделов, и никакой обработки не происходит до тех пор, пока тематические разделы не будут переназначены, что иногда называется перебалансировкой «остановки мира». Чтобы усугубить проблему, в зависимости от используемого экземпляра ConsumerPartitionAssignor потребителям просто переназначаются те же тематические разделы, которыми они владели до перебалансировки, в результате чего на самом деле нет необходимости приостанавливать работу над этими разделами.
- Эта реализация протокола перебалансировки называется **eager rebalancing** потому, что в ней отдается приоритет важности обеспечения того, чтобы ни один из потребителей в группе сам не претендовал на владение одним и тем же тематическим разделом. Владение одним и тем же разделом темы двумя потребителями в одной группе приведет к неопределенному поведению.
- также известны как **штормы перебалансировки** - это может привести кластер в состояние последовательной перебалансировки, и кластеру Connect может потребоваться несколько минут для стабилизации.

group coordinator находится на сервере кафки, консьюмеры сообщают ему о том что они живы через "heartbeat" (это сделано только для обеспечения ребаланса в случае смерти)

- при добавлении консьюмера в консьюмер-группу он на самом деле добавляется в group-coordinator

heartbeat.interval.ms - каждый консьюмер стучит серверу что он жив каждые три секунды

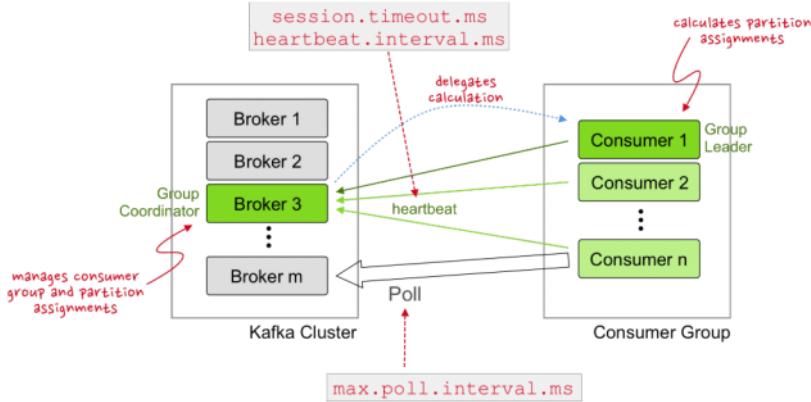
session.timeout.ms - если в течение этого времени не было ни одного heartbeat то производится ребаланс-консьюмеров

max.poll.interval.ms если консьюмер не читал сообщения из кафки в течение 5 минут, то он считается

мертвым (поэтому клиент в цикле должен постоянно читать из кафки в цикле через операцию pool(duration))

- это промежуток между двумя poll() вызовами
- это нужно потому что если background-тред по отправке хартбита в либе будет исправно работать, а основной поток проги зависнет
- ? наверное это время можно сделать раза в три больше чем pool(Duration)+обработка_в_цикле ?
- ребаланс не может занимать больше времени чем max.poll.interval.ms (включая session.timeout.ms) [ie session.timeout.ms должен быть меньше чем max.poll.interval.ms]

Consumer Liveness



Here is an image that depicts how the liveness of a consumer in a consumer group is determined.

- Each consumer sends a periodic liveness signal to the group coordinator (broker) on a dedicated thread
- If no liveness signal is received for more than session.timeout.ms milliseconds (default: 10s) then the consumer is considered to be dead and the group coordinator triggers a partition reassignment
- The calculation of the partition assignment to the remaining consumer group member is delegated to the **group leader**
- The group coordinator then communicates the new partition assignments to each consumer of the group
- If the liveness thread of a consumer is still working but its main thread, where the polling for data is happening, "hangs" then there is another timeout time called max.poll.interval.ms (default: 5min) after exceeded the corresponding consumer is considered dead and the group coordinator triggers a partition reassignment. The parameter heartbeat.interval.ms (default: 3s) in turn defines how long the interval between two successive heartbeat signals are.

??ребаланс сразу планируется как только присоединяется/явно_отсоединяется консьюмер, но затем выдерживается пауза session.timeout.ms и только после этого начинает реально

ИСПОЛНЯТСЯ

- те время нам надо выставлять, чтобы новый инстанс микросервиса успел подняться (но это не страхует от некоторых неправильных сообщений которые могут проскочить в "несвои" микросервисы)
- но увеличение этого времени также влияет на более медленное подхватывание новых консьюмеров
- Once a rebalance has begun, the coordinator starts a timer which is set to expire after the group's session timeout

Consumer Rebalancing

Rebalancing on:

- Consumer **leaves** group
- Consumer **joins** group
- Consumer **changes topic subscription**
- Number of partitions changes



Consumption is **stopped** during rebalance

- **Question:** could adding a Consumer to a Consumer Group cause Partition assignment to change?

- Consumer rebalances are initiated when
 - A Consumer leaves the Consumer group (either by failing to send a timely heartbeat or by explicitly requesting to leave)
 - **A new Consumer joins the Consumer Group**
 - A Consumer changes its Topic subscription
 - The Consumer Group notices a change to the Topic metadata for any subscribed Topic (e.g. an increase in the number of Partitions)
- Rebalancing does not occur if
 - A Consumer calls **pause**
- During rebalance, consumption is paused

Answer: Yes!

Once a rebalance has begun, the coordinator starts a timer which is set to expire after the group's session timeout. Each member in the previous generation detects that it needs to rejoin with its periodic heartbeats sent to the coordinator. The coordinator uses the **REBALANCE_IN_PROGRESS** error code in the heartbeat response so the Consumer knows to rejoin.

[? как сделать liveness probe в кубере](#)

java-консьюмер может отловить ребаланс

The Case For and Against Rebalancing

- Usually Rebalancing is **desired**
- But → Rebalancing is like **fresh start**
- Consumers manage rebalancing with:

```
public interface ConsumerRebalanceListener {  
    void onPartitionsAssigned(Collection<TopicPartition> partitions)  
    void onPartitionsRevoked(Collection<TopicPartition> partitions)  
}
```

- Typically, Partition rebalancing is a good thing
 - Allows you to add more Consumers to a Consumer Group dynamically, without having to restart all the other Consumers in the group
 - Automatically handles situations where a Consumer in the Consumer Group fails
- However a rebalance is like a fresh restart
 - Consumers may or may not get a different set of Partitions
 - * If your Consumer is relying on getting all data from a particular Partition, this could be a problem
 - A previously-paused Partition is no longer paused
- Managing rebalances where Partition assignment matters
 - Option 1: only have a single Consumer for the entire topic
 - Option 2: provide a `ConsumerRebalanceListener` when calling `subscribe()`
 - * Implement `onPartitionsRevoked` and `onPartitionsAssigned` methods
 - Soft option 3: Sticky partition assignment strategy (KIP-54) does not guarantee that assignments do not change. Assignments can and do change, because the goals are, in priority order:
 - * Topic partitions are still distributed as evenly as possible
 - * Topic partitions stay with their previously assigned consumers as much as possible

 Transactions will not automatically work across rebalances in a consumer group. Kafka Streams has special functionality to support this, but it's not built-in at this level

1) range 2) roundRobin

15 декабря 2020 г. 23:16

(1 и 2) partitions are assigned **automatically** to consumers and are rebalanced automatically when consumers are added or removed from the group

- вот этот вот ребаланс нужен как восстановление от сбоев так как кафка сама автоматически определяет что консьюмер отвалился
- не гарантирует assignment preservation т.e. после ребаланса консьюмеры могут получить другие партиции
 - An important note about Range and RoundRobin: Neither strategy guarantees that Consumers will retain the same Partitions after a reassignment. In other words, if a Consumer 1 is assigned to Partition A-0 right now, Partition A-0 may be assigned to another Consumer if a reassignment were to happen. Most Consumer applications are not locality-dependent enough to require that Consumer-Partition assignments be static.
- 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>

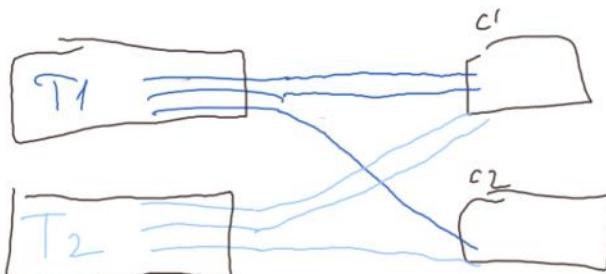
Range

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.

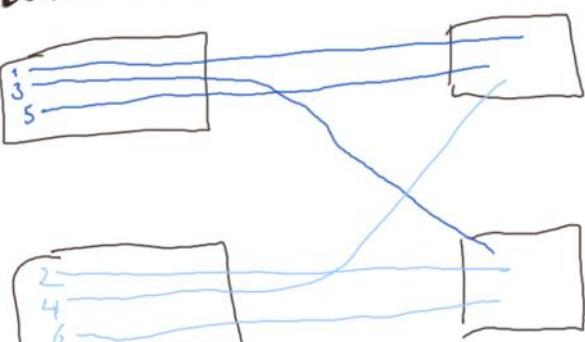
RoundRobin

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).

Бип1) range (default)

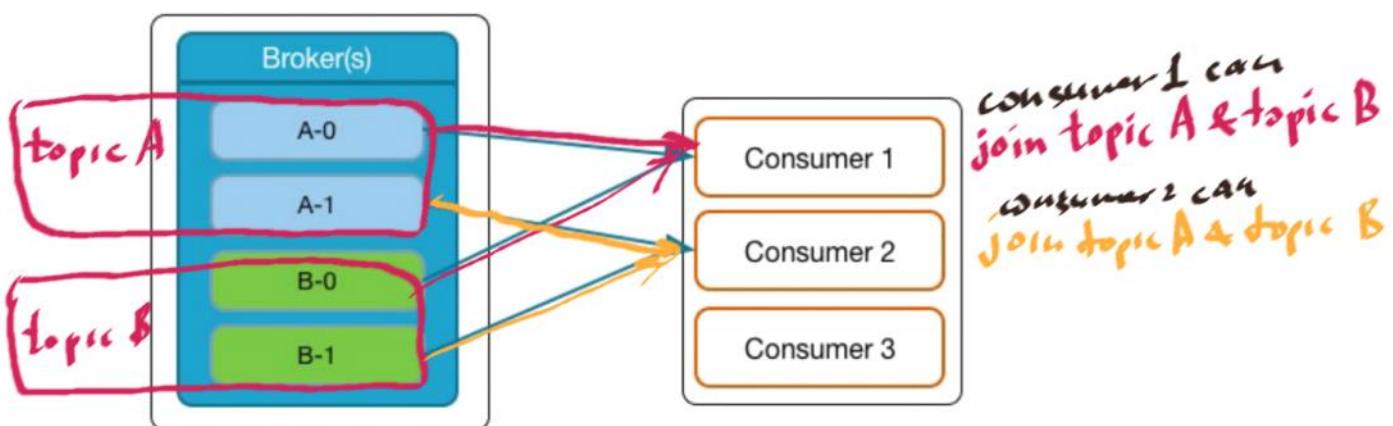


Бип2) round robin



co-partitioning - одна из причин по которой удобен **range partitioner**: для того чтобы клиент мог сдюжинить топики с одинаковым количеством партиций

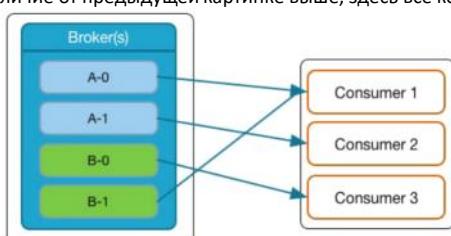
- это стратегия нужна только если нам нужно учитывать взаимное влияние топиков
- тк хэш-функция одинаковая то если в двух топиках используется один и тот же ключ (например номер клиента) то один и тот же клиент попадет в 1-ую партицию первого топика и также в 1-ую партицию второго топика,
- In Range, the Partition assignment assigns matching partitions to the same Consumer. In this example, there are two 2-partition Topics and three Consumers. The first partition from each Topic is assigned to one Consumer, the second partition from each is assigned to another Consumer, repeating until there are no more Partitions to assign. Since we have more Consumers than Partitions in any Topic, one Consumer will be idle.
- The Range strategy is useful for "co-partitioning", which is particularly useful for Topics with keyed messages. Imagine that these two Topics are using the same key - for example, a userid. Topic A is tracking search results for specific user IDs; Topic B is tracking search clicks for the same set of user IDs. By using the same user IDs for the key in both Topics, messages with the same key would land in the same numbered Partition in both Topics (assuming both topics had the same number of Partitions) and so will land in the same Consumer.



roundRobin partitioner делает более равномерную загрузку консьюмеров (подходит только если мы из обрабатываем записи разных топиков независимо друг от друга)

- The RoundRobin strategy is much simpler. Partitions are assigned one at a time to Consumers in a rotating fashion until all the Partitions have been assigned. This provides much more balanced loading of Consumers than Range.

См в отличие от предыдущей картинке выше, здесь все консьюмеры заняты



3) standalone consumer

15 декабря 2020 г. 23:17

(3) standalone consumer (Use a Consumer Without a Group) например когда конкретный консьюмер должен читать только определенную партицию в топике (не гарантирует порядок сообщений)

- 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 все эти автоматический ребаланс и прочее (тк если я хочу скейлить независимо друг от друга кафку и консьюмеры)
- <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"); ①

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
            partition.partition()));
    consumer.assign(partitions); ②

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

        for (ConsumerRecord<String, String> record : 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

① 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.

② 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.

6.2.8. Minimizing the impact of rebalances

The **rebalancing** of a partition between active consumers in a group is the time it takes for:

- Consumers to commit their offsets
- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

Clearly, the process increases the downtime of a service, particularly when it happens repeatedly during a rolling restart of a consumer group cluster.

In this situation, you can use the concept of *static membership* to reduce the number of rebalances. **Rebalancing** assigns topic partitions evenly among consumer group members.

6.2.8. Minimizing the impact of rebalances

The rebalancing of a partition between active consumers in a group is the time it takes for:

- Consumers to commit their offsets
- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

Clearly, the process increases the downtime of a service, particularly when it happens repeatedly during a rolling restart of a consumer group cluster.

In this situation, you can use the concept of static membership to reduce the number of rebalances. Rebalancing assigns topic partitions evenly among consumer group members. Static membership uses persistence so that a consumer instance is recognized during a restart after a session timeout.

The consumer group coordinator can identify a new consumer instance using a unique id that is specified using the `group.instance.id` property. During a restart, the consumer is assigned a new member id, but as a static member it continues with the same instance id, and the same assignment of topic partitions is made.

If the consumer application does not make a call to poll at least every `max.poll.interval.ms` milliseconds, the consumer is considered to be failed, causing a rebalance. If the application cannot process all the records returned from poll in time, you can avoid a rebalance by using the `max.poll.interval.ms` property to specify the interval in milliseconds between polls for new messages from a consumer. Or you can use the `max.poll.records` property to set a maximum limit on the number of records returned from the consumer buffer, allowing your application to process fewer records within the `max.poll.interval.ms` limit.

```
# ...
group.instance.id=UNIQUE-ID 1
max.poll.interval.ms=300000 2
max.poll.records=500 3
# ...
```

- ➊ The unique instance id ensures that a new consumer instance receives the same assignment of topic partitions.
- ➋ Set the interval to check the consumer is continuing to process messages.
- ➌ Sets the number of processed records returned from the consumer.

назначение вручную партиции консьюмеру - как альтернатива репартиционированию и перебалансировке

- Хотя Kafka и выравнивает нагрузку по топикам/секциям между всеми потребителями, назначение топиков и секций носит недетерминистский характер: нельзя предугадать, какие пары топиков/секций получит каждый из потребителей.
- В классе KafkaConsumer имеются методы, с помощью которых можно подписаться на конкретные топик и секцию

```
TopicPartition fooTopicPartition_0 = new TopicPartition("foo", 0);
TopicPartition barTopicPartition_0 = new TopicPartition("bar", 0);

consumer.assign(Arrays.asList(fooTopicPartition_0, barTopicPartition_0));
```

в случае отказа одной из машин не произойдет автоматического переназначения секций топиков, даже для потребителей с одним идентификатором группы.

- Для изменения назначений понадобится выполнить еще один вызов метода `consumer.assign`

group.id для таких консьюмеров не имеет значения, поэтому его лучше делать уникальным

- для фиксации используется группа, указанная в настройках потребителя, но, поскольку все потребители будут функционировать сами по себе, имеет смысл задать для каждого из них свой уникальный идентификатор группы.

продьюсер явно задает партицию

onenote:///C:/Users/trans/Qsync/vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA \ key%20hashing.one#custom%20partitioner%20альтернатива§ion-id={946F12D7-F08C-4162-99EC-1F1AF34560D5}&page-id={D547067B-DDC0-40D4-B8DB-6A309A47B677}&end

- В текущей версии Kafka 2.0 в классе ProducerRecord шесть перегруженных конструкторов. См. документацию Kafka: <https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html>.

Листинг 2.4. Указываем секцию вручную

```
► AtomicInteger partitionIndex = new AtomicInteger(0);

    int currentPartition = Math.abs(partitionIndex.getAndIncrement()) %
        numberPartitions;
    ProducerRecord<String, String> record =
        new ProducerRecord<>("topic", currentPartition, "key", "value");

Создание переменной экземпляра
класса AtomicInteger
```

Получаем текущую секцию
и используем ее в качестве параметра

4) static membership

15 декабря 2020 г. 23:19

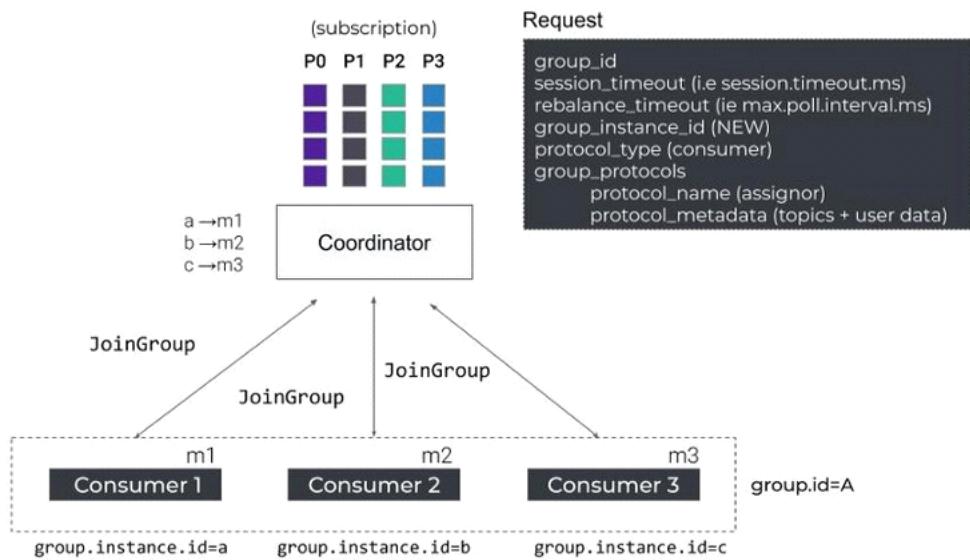
(4) static membership вариант вообще не делать перебалансировку (а при восстановлении консьюмера дать ему ранее выделенные ресурсы)

- . В идеале группа клиентов Kafka могла бы компенсировать эту временную потерю ресурсов без выполнения полной перебалансировки. Как только узел возвращается, ему немедленно будут назначены ранее выделенные ресурсы.
- <https://cwiki.apache.org/confluence/display/KAFKA/KIP-345%20A+Introduce+static+membership+protocol+to+reduce+consumer+rebalances>
- недостаток в том что консьюмер должен знать свой номер (но в кубернетесе решается через statefulset и клиент возьмет номер из имени своего пода)
- session.timeout.ms упавшего консьюмера - только после этого будет перебалансировка
- При использовании статического членства рекомендуется увеличить свойство session.timeout.ms клиента до достаточно большого размера, чтобы координатор брокера не запускал перебалансировку слишком часто
- недостаток в том что партиция все это время недоступна (и из нее никто не потребляет) - disadvantage of increasing the unavailability of partitions because the coordinating broker may only detect a failing consumer after a few minutes (depending on session.timeout.ms).
- достоинство в том что нет "stop-the-world"

Static Membership

To reduce consumer rebalances due to transient failures, Apache Kafka 2.3 introduces the concept of Static Membership with the [KIP-345](#).

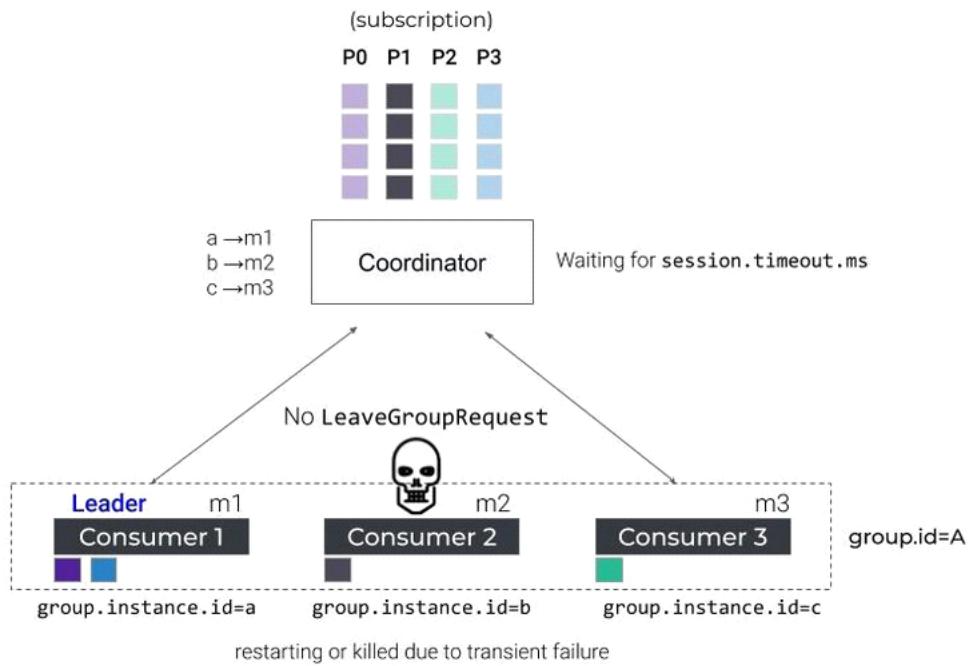
The main idea behind static membership is that each consumer instance is attached to a unique identifier configured with `group.instance.id`. The membership protocol has been extended so that ids are propagated to the broker coordinator through the `JoinGroup` request.



group.instance.id=a group.instance.id=b group.instance.id=c

39

If a consumer is restarted or killed due to a transient failure, the broker coordinator will not inform other consumers that a rebalance is necessary until `session.timeout.ms` is reached. One reason for that is that consumers will not send `LeaveGroup` request when they are stopped.



When the consumer will finally rejoin the group, the broker coordinator will return the cached assignment back to it, without doing any rebalance.

5) sticky

15 декабря 2020 г. 23:20

(5) sticky (consistent hashing)

- ребаланс все равно происходит (отвалившегося консьюмера никто ждать не будет пока его заменят на новый) но алгоритм стремится свести изменения к минимуму
- гарантирует assignment preservation
 - o If your application requires Partition assignments to be preserved across reassigned, use the Sticky strategy. Sticky is RoundRobin with assignment preservation.
- Preserves existing Partition assignments to Consumers during reassignment to reduce overhead
- Kafka Consumers retain pre-fetched messages for Partitions assigned to them before a reassignment
- Reduces the need to cleanup local Partition state between rebalances
- In certain circumstances the round robin assignor, which produces better assignments compared to range assignor, fails to produce an optimal and balanced assignment of topic partitions to consumers. KIP-49 touches on some of these circumstances. In addition, when a reassignment occurs, none of the existing strategies consider what topic partition assignments were before reassignment, as if they are about to perform a fresh assignment. Preserving the existing assignments could reduce some of the overheads of a reassignment. For example, Kafka consumers retain pre-fetched messages for partitions assigned to them before a reassignment. Therefore, preserving the partition assignment could save on the number of messages delivered to consumers after a reassignment. Another advantage would be reducing the need to cleanup local partition state between rebalances.

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-54>

[++Sticky+Partition+Assignment+Strategy](#)

<https://www.confluent.de/blog/5-things-every-kafka-developer-should-know/>

<https://www.confluent.de/blog/incremental-cooperative-rebalancing-in-kafka/>

<https://medium.com/streamthoughts/understanding-kafka-partition-assignment-strategies-and-how-to-write-your-own-custom-assignor-ebeda1fc06f3>

<https://www.confluent.de/blog/apache-kafka-producer-improvements-sticky-partitioner/>

<https://www.confluent.io/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/>

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%203A+Kafka+Consumer+Incremental+Rebalance+Protocol>

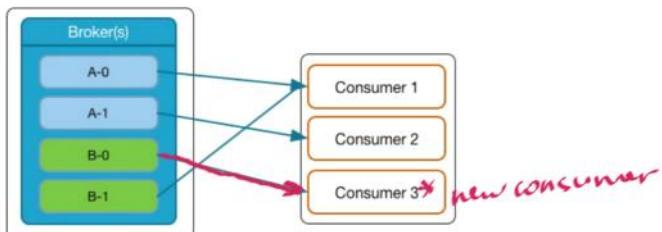
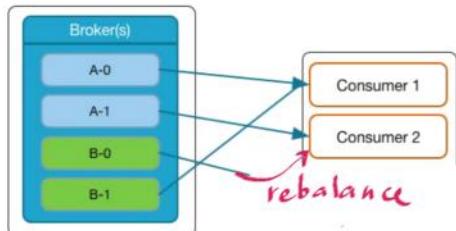
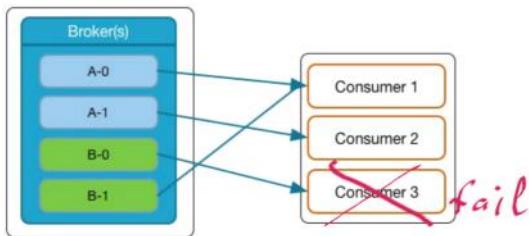
<https://medium.com/streamthoughts/apache-kafka-rebalance-protocol-or-the-magic-behind-your-streams-applications-e94baf68e4f2>

<https://stackoverflow.com/questions/56561378/how-to-introduce-delay-in-rebalancing-in-case-of-kafka-consumer-group>

<https://cwiki.apache.org/confluence/display/KAFKA/Incremental+Cooperative+Rebalancing%203A+Support+and+Policies>

<https://cwiki.apache.org/confluence/display/KAFKA/KIP-345%203A+Introduce+static+membership+protocol+to+reduce+consumer+rebalances>

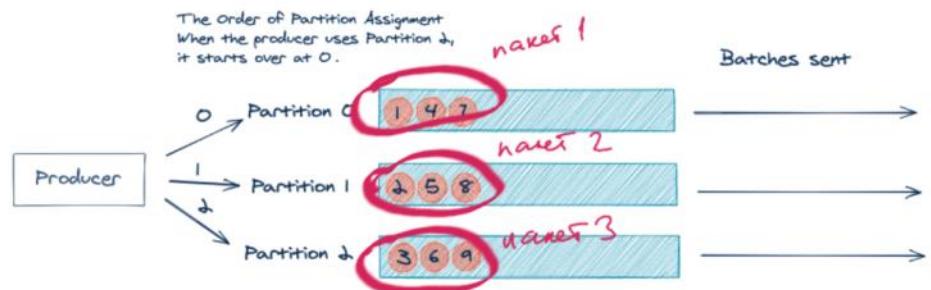
```
# sticky strategy
kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --
topic test --property print.key=true --group second --
consumer.config vovan.properties
# в файле vovan.properties написать
partition.assignment.strategy=org.apache.kafka.clients.consumer.StickyAssignor
```



★ также эта sticky-стратегия уменьшает количество пакетов отправляемых по сети

- Стоит отметить, что липкий разделитель по-прежнему обеспечивает равномерное распределение записей. Равномерное распределение происходит с течением времени, поскольку секционер отправляет пакет в каждый раздел. Вы можете думать об этом как о циклическом подходе «на каждый пакет» или «в конечном итоге даже»

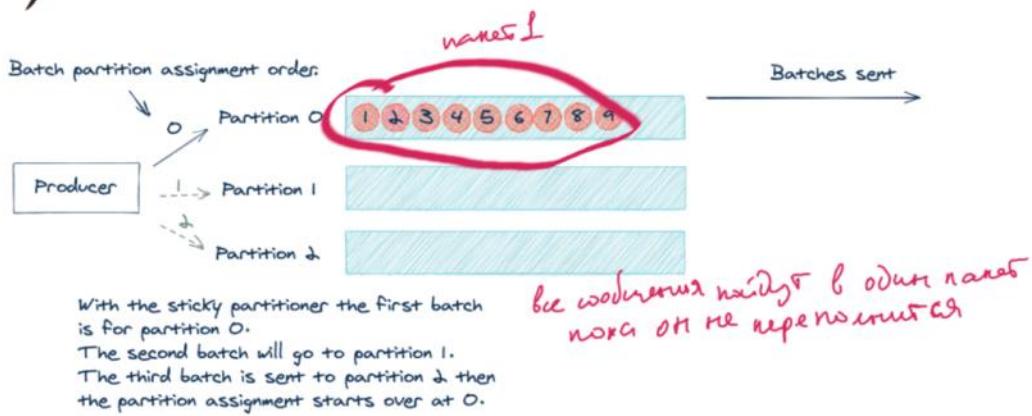
bap1) round robin \rightarrow coördineren key hashing



Nine records are produced at the same time.

The producer assigns partitions in a round robin fashion indicated by the number on the record.
The producer sends three batches to the broker as a result.

bap2) sticky \rightarrow coördineren key hashing



6) CooperativeStickyAssignor (cooperative rebalancing)

15 декабря 2020 г. 23:21

(6) CooperativeStickyAssignor (cooperative rebalancing)

в файле `vovan.properties` написать `# только для кафки 2.6`

```
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor  
sticky
```

- Any member that revoked partitions then rejoins the group, triggering a second rebalance so that its revoked partitions can be assigned. Until then, these partitions are unowned and unassigned. The synchronization barrier hasn't been dropped at all; it turns out that it just needed to be moved.
- Любой член, отозвавший разделы, затем снова присоединяется к группе, вызывая вторую перебалансировку, чтобы можно было назначить его отозванные разделы. До тех пор эти разделы не принадлежат владельцам и не назначены. Барьер синхронизации вообще не был снят; оказывается, его просто нужно было переместить.
- So the sticky aspect is just as important as the cooperative in the new assignor. And thanks to the owned partitions encoded in the subscription, being sticky is as easy as ever.

stop-the-world problem

- При кооперативном подходе потребители не отказываются автоматически от владения всеми тематическими разделами в начале перебалансировки. Вместо этого все участники кодируют свое текущее назначение и персылают информацию лидеру группы. Затем руководитель группы определяет, в каких разделах необходимо сменить владельца, вместо того, чтобы создавать совершенно новое назначение с нуля
- решает проблему "stop-the-world" остановки всех потребителей на время перебалансировки
- Суть в том, что отказ от подхода «остановки мира» к ребалансировке и остановка только задействованных тематических разделов означает менее дорогостоящую перебалансировку, что сокращает общее время для завершения ребалансировки.

incremental

- такая перебалансировка называется инкрементальной так как клиенты вскоре сходятся к состоянию сбалансированной нагрузки после нескольких последовательных перебалансировок. Небольшое количество последовательных раундов ребалансировки может быть использовано для того, чтобы группа клиентов Kafka достигла желаемого состояния сбалансированных ресурсов. Кроме того, вы можете настроить grace period , чтобы позволить уходящему участнику вернуться и восстановить свои ранее назначенные ресурсы.

cooperative

- такая перебалансировка называется кооперативной потому что каждый процесс в группе должен добровольно высвободить ресурсы, которые необходимо перераспределить. Затем эти ресурсы становятся доступными для перепланирования, учитывая, что клиент, которого попросили освободить их, сделал это вовремя.

group.initial.rebalance.delay.ms ?? (broker level config)

- I want to give some time to my consumer to restart so that unnecessary rebalance doesn't happen
 - The amount of time the group coordinator will wait for more consumers to join a new group before performing the first rebalance. A longer delay means potentially fewer rebalances, but increases the time until processing begins.
 - This config only works if you create a new consumer group, and the delay is only considered when the first consumer joins an empty group
-
- GroupCoordinator брокера распределяет все секции топиков первому запускаемому экземпляру приложения Kafka Streams. При запуске следующего экземпляра происходит перебалансировка, в результате которой текущие назначения секций топиков сбрасываются и секции топиков распределяются заново по обоим экземплярам Kafka Streams. Этот процесс повторяется до тех пор, пока не запустятся все экземпляры приложения Kafka Streams с одним идентификатором приложения
 - Для приложения Kafka Streams такое поведение — норма. Но обработка записей приостанавливается на время перебалансировки, до ее завершения; следовательно, желательно при запуске приложения ограничить по возможности число перебалансировок.
 - В версии Kafka 0.11.0 появилась новая настройка брокеров — `group.initial.rebalance.delay.ms`. Эта настройка откладывает начальную перебалансировку потребителя при его

присоединении к группе на указанное в group.initial.rebalance.delay.ms время (в миллисекундах). Значение данной настройки по умолчанию равно 3 секундам. По мере присоединения к группе других потребителей перебалансировка откладывается на заданный промежуток времени (вплоть до достижения предела, задаваемого параметром max.poll.interval.ms). Это полезно для Kafka Streams, поскольку перебалансировка при запуске новых экземпляров откладывается до момента включения их всех в работу (если они запускаются по очереди). Например, если запустить четыре экземпляра приложения, задав подходящую задержку перебалансировки, то будет выполнена одна перебалансировка после запуска всех четырех экземпляров, а значит, вы сможете быстрее начать обрабатывать данные

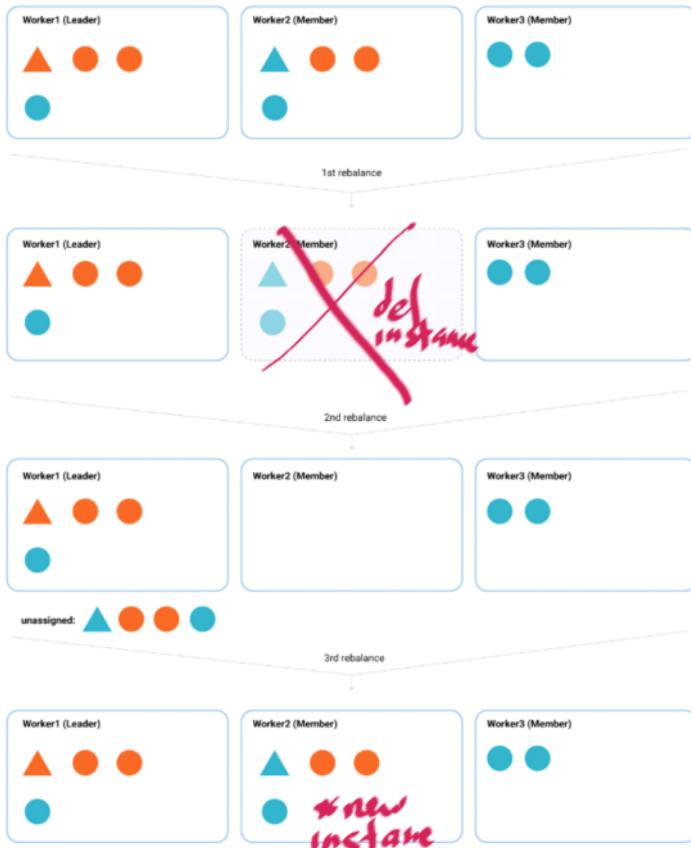
scheduled.rebalance.max.delay.ms (?только для kafka connect?) - отложить ребаланс на время - чтобы дать новому консьюмеру подняться

- Кроме того, вы можете настроить grace period , чтобы позволить уходящему участнику вернуться и восстановить свои ранее назначенные ресурсы.
- If not reassigning immediately, set ScheduledRebalanceTimeout appropriately to defer the actual movement in the hopes the member will reappear. In this case there should be no RevokeTopicPartitions
- http://mail-archives.apache.org/mod_mbox/kafka-commits/201905.mbox/%3C155806478876.1237.7369403125243358398@gitbox.apache.org%3E
- <https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol>
- <https://cwiki.apache.org/confluence/display/KAFKA/KIP-441%3A+Smooth+Scaling+Out+for+Kafka+Streams>
- Почему нет такой настройки для KAFKA CONSUMER: Note that one minor difference compared with KIP-415 is that we do not introduce the scheduledDelay in the protocol, but instead the consumer will trigger rebalance immediately. This is because the consumer protocol would applies to all consumers (including streams) and hence should be kept simple, and also because KIP-345 is being developed in parallel which is aimed for tackling the scaling out / rolling bounce scenarios already.

<https://cwiki.apache.org/confluence/display/KAFKA/Incremental+Cooperative+Rebalancing%3A+Support+and+Policies>

- a) new subscriptions
 - This case can be detected based on the subscriptions and assignments. If a topic is completely missing from previous assignments, assume it is this case.
 - These should be resolved immediately, so just continue with immediate assignment.
- b) a member left the group
 - If we didn't detect a new subscription, assume it is due to this case.
 - However, if we detect a new member, we want to utilize it immediately to resolve the imbalance. This could happen if the process manages to get restarted within the time it takes to do the rebalance. This process rejoins the group but its list of previously AssignedTopicPartitions is empty due to the restart. For a stable leader, they can just compare the set of member IDs to the last generation. For a new leader we can use a heuristic such as looking for a member that has no previous assignments (assuming there's enough members that they *should* have had an assignment).
 - If not reassigning immediately, set ScheduledRebalanceTimeout appropriately to defer the actual movement in the hopes the member will reappear. In this case there should be no RevokeTopicPartitions.

<https://medium.com/streamthoughts/apache-kafka-rebalance-protocol-or-the-magic-behind-your-streams-applications-e94baf68e4f2>



schedule rebalance time delay
 перегруппировка не происходит сразу а с задержкой чтобы успеть подготовка нового участника

probing.rebalance.interval.ms (only for kafka streams)

- The maximum time to wait before triggering a rebalance to probe for warmup replicas that have restored enough to be considered caught up. Streams will only assign stateful active tasks to instances that are caught up and within the acceptable.recovery.lag, if any exist. Probing rebalances are used to query the latest total lag of warmup replicas and transition them to active tasks if ready. They will continue to be triggered as long as there are warmup tasks, and until the assignment is balanced. Must be at least 1 minute.
- <https://atsc.com.sg/docs/edp/3-configuration/3-6-kafka-streams-configs/>
- <https://kafka.apache.org/26/documentationstreams/developer-guide/config-streams>
- <https://medium.com/streamthoughts/apache-kafka-rebalance-protocol-or-the-magic-behind-your-streams-applications-e94baf68e4f2>

heartbeat.interval.ms **session.timeout.ms** and the **max.poll.interval.ms** - это если консьюмер отвалился случайно

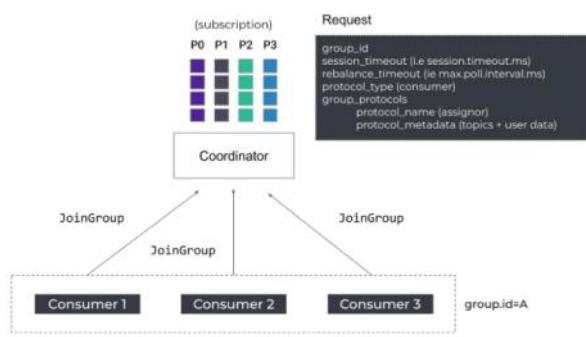
- session.timeout.ms должно быть в три раза больше чем heartbeat.interval.ms = 30000 (30сек)
- session.timeout.ms=100000 должно быть между group.min.session.timeout.ms and group.max.session.timeout.ms (по дефолту 6 сек - 30 мин)
- Currently broker accepts a config value called **rebalance timeout** which is provided by consumer **max.poll.intervals**. The reason we set it to poll interval is because consumer could only send request within the call of poll() and we want to wait sufficient time for the join group request. When reaching rebalance timeout, the group will move towards COMPLETING_REBALANCE stage and remove unjoined members. This is actually conflicting with the design of static membership, because those temporarily unavailable members will potentially reattempt the join group and trigger extra rebalances. Internally we would optimize this logic by having rebalance timeout only in charge of stopping PREPARE_REBALANCE stage, without removing non-responsive members immediately. There would not be a full rebalance if the lagging consumer sends a JoinGroupRequest within the session timeout.
- So in summary, the member will only be removed due to session timeout. We shall remove it from both in-memory static `group.instance.id` map and member list.

Присоединиться к группе

Когда потребитель запускается, он отправляет первый `FindCoordinator` запрос на получение координатора брокера Kafka,

Присоединиться к группе

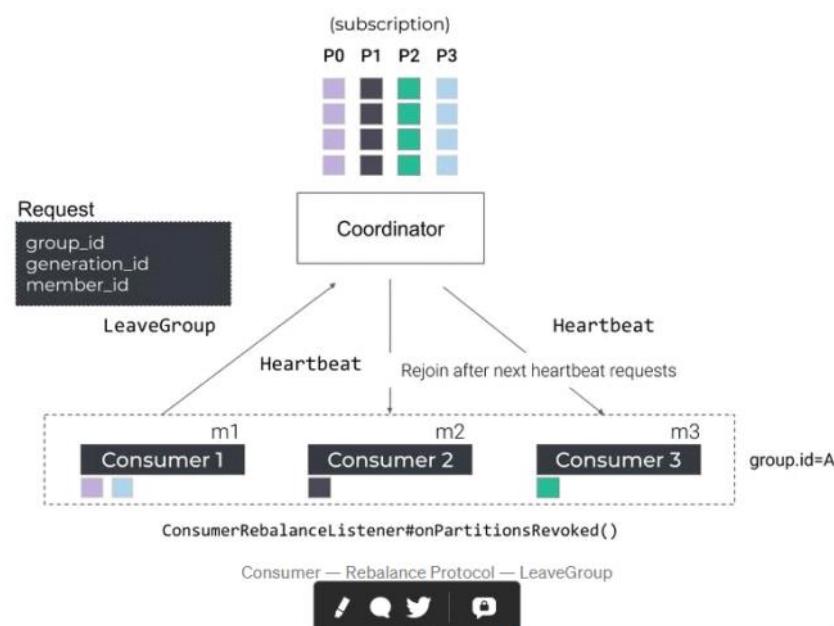
Когда потребитель запускается, он отправляет первый `FindCoordinator` запрос на получение координатора брокера Kafka, который отвечает за его группу. Затем он инициирует протокол перебалансировки, отправив `JoinGroup` запрос.



Потребитель - протокол ребалансировки - запрос SyncGroup

Как мы видим, он `JoinGroup` / туюю конфигурацию клиента-потребителя, такую как свойства `session.timeout.ms` и `max.poll.interval.ms`. Эти свойства используются координатором для исключения участников из группы, если они не отвечают.

For example, let's properly stop one of our instances. In this first rebalance scenario, the consumer will send a `LeaveGroup` request to the coordinator, before stopping.



Remaining consumers will be notified that a **rebalance must be performed on the next heartbeat** and will initiate a new `JoinGroup/SyncGroup` round-trip in order to reassigned partitions.

```
# ...  
heartbeat.interval.ms=3000 ①  
session.timeout.ms=10000 ②  
auto.offset.reset=earliest ③  
# ...
```

- ① Adjust the heartbeat interval lower according to anticipated **rebalances**.
- ② If no heartbeats are received by the Kafka broker before the timeout duration expires, the consumer is removed from the consumer group and a **rebalance** is initiated. If the broker configuration has a `group.min.session.timeout.ms` and `group.max.session.timeout.ms`, the session timeout value must be within that range.

```
# ...
heartbeat.interval.ms=3000 1
session.timeout.ms=10000 2
auto.offset.reset=earliest 3
# ...
```

- 1** Adjust the heartbeat interval lower according to anticipated rebalances.
- 2** If no heartbeats are received by the Kafka broker before the timeout duration expires, the consumer is removed from the consumer group and a rebalance is initiated. If the broker configuration has a `group.min.session.timeout.ms` and `group.max.session.timeout.ms`, the session timeout value must be within that range.
- 3** Set to earliest to return to the start of a partition and avoid data loss if offsets were not committed.

`heartbeat.interval.ms`

Type: int
Default: 3000 (3 seconds)
Importance: high

The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.

`session.timeout.ms`

Type: int
Default: 10000 (10 seconds)
Importance: high

The timeout used to detect client failures when using Kafka's group management facility. The client sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this client from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.

`max.poll.interval.ms`

Type: int
Default: 300000 (5 minutes)
Valid Values: [1,...]
Importance: medium

The maximum delay between invocations of `poll()` when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If `poll()` is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member. For consumers using a non-null `group.instance.id` which reach this timeout, partitions will not be immediately reassigned. Instead, the consumer will stop sending heartbeats and partitions will be reassigned after expiration of `session.timeout.ms`. This mirrors the behavior of a static consumer which has shutdown.

? прочие настройки

https://access.redhat.com/documentation/en-us/red_hat_amq/7.5/html/using_amq_streams_on_rhel/kafka-connect-configuration-parameters-str

`heartbeat.interval.ms`

Type: int
Default: 3000
Importance: high

The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.

`rebalance.timeout.ms`

Type: int

`rebalance.timeout.ms`

Type: int

Default: 60000

Importance: high

The maximum allowed time for each worker to join the group once a rebalance has begun. This is basically a limit on the amount of time needed for all tasks to flush any pending data and commit offsets. If the timeout is exceeded, then the worker will be removed from the group, which will cause offset commit failures.

`session.timeout.ms`

Type: int

Default: 10000

Importance: high

The timeout used to detect worker failures. The worker sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove the worker from the group and initiate a `rebalance`. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.

`scheduled.rebalance.max.delay.ms`

Type: int

Default: 300000

Valid Values: [0,...,2147483647]

Importance: low

Compatibility mode for Kafka Connect Protocol.

`auto.leader.rebalance.enable`

Type: boolean

Default: true

Importance: high

Dynamic update: read-only

Enables auto leader balancing. A background thread checks the distribution of partition leaders at regular intervals, configurable by `leader.imbalance.check.interval.seconds`. If the leader imbalance exceeds `leader.imbalance.per.broker.percentage`, leader `rebalance` to the preferred leader for partitions is triggered.

`leader.imbalance.check.interval.seconds`

Type: long

Default: 300

Importance: high

Dynamic update: read-only

The frequency with which the partition `rebalance` check is triggered by the controller.

`group.initial.rebalance.delay.ms`

Type: int

Default: 3000 (3 seconds)

Importance: medium

Dynamic update: read-only

The amount of time the group coordinator will wait for more consumers to join a new group before performing the first `rebalance`. A longer delay means potentially fewer `rebances`, but increases the time until processing begins.

brocker config

<code>leader.imbalance.check.interval.seconds</code>	The frequency with which the partition <code>rebalance</code> check is triggered by the controller	long	300
--	--	------	-----

consumer config

heartbeat.interval.ms	The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than <code>session.timeout.ms</code> , but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	int	3000
max.partition.fetch.bytes	The maximum amount of data per-partition the server will return. Records are fetched in batches by the consumer. If the first record batch in the first non-empty partition of the fetch is larger than this limit, the batch will still be returned to ensure that the consumer can make progress. The maximum record batch size accepted by the broker is defined via <code>message.max.bytes</code> (broker config) or <code>max.message.bytes</code> (topic config). See <code>fetch.max.bytes</code> for limiting the consumer request size.	int	1048576
session.timeout.ms	The timeout used to detect consumer failures when using Kafka's group management facility. The consumer sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this consumer from the group and initiate a <code>rebalance</code> . Note that the value must be in the allowable range as configured in the broker configuration by <code>group.min.session.timeout.ms</code> and <code>group.max.session.timeout.ms</code> .	int	10000
max.poll.interval.ms	The maximum delay between invocations of <code>poll()</code> when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If <code>poll()</code> is not called before expiration of this timeout, then the consumer is considered failed and the group will <code>rebalance</code> in order to reassign the partitions to another member.	int	300000
kafka connect config			
rebalance.timeout.ms	The maximum allowed time for each worker to join the group once a <code>rebalance</code> has begun. This is basically a limit on the amount of time needed for all tasks to flush any pending data and commit offsets. If the timeout is exceeded, then the worker will be removed from the group, which will cause offset commit failures.	int	60000
session.timeout.ms	The timeout used to detect worker failures. The worker sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove the worker from the group and initiate a <code>rebalance</code> . Note that the value must be in the allowable range as configured in the broker configuration by <code>group.min.session.timeout.ms</code> and <code>group.max.session.timeout.ms</code> .	int	10000
scheduled.rebalance.max.delay.ms	Compatibility mode for Kafka Connect Protocol	int	300000

custom partitioner

12 декабря 2020 г. 18:26

<https://dzone.com/articles/custom-partitioner-in-kafka-lets-take-quick-tour>

Kafka's Default Partitioning Scheme

- `partition_index = hash(key) % number_of_partitions`
- All messages with **same key** go to **same partition**
- If `key=NULL`: use **Round Robin**
- Override with **custom partitioner** if needed

- Recall that by default, if a message has a key, Kafka will hash the key and use the result to map the message to a specific Partition
- This means that all messages with the same key will go to the same Partition
- If the key is null and the default Partitioner is used, the record will be sent to a random Partition (using a round-robin algorithm)
- You may wish to override this behavior and provide your own Partitioning scheme
 - For example, if you will have many messages with a particular key, you may want one Partition to hold just those messages



Kafka uses its own hash algorithm, so the hash will not change if the version of Java on the machine is upgraded and a new hashing algorithm is introduced.

~~т-ф: сделать partitioner который 1 в 1 сопоставляет партиции и консьюмеры~~ заменен на static membership

- в таком случае консьюмеров всегда должно быть ровно столько сколько партиций
- недостаток будет в том что сообщения в партиции (у которой нет консьюмера) будут накапливаться

- **партишенер** ответчает за то в какую партицию, продьюсер **отправит сообщение** (сам партишенер работает на клиенте-продьюсере)
- **a consumer rebalancing** протокол отвечает за то какой консьюмер получит сообщение (сам протокол работает на клиенте-консьюмере)

мой кастомный партишенер на основе данного ключа и значения решает в какую партицию топика отправить данное сообщение

- в идеале я могу прям заранее четко знать партицию куда в итоге отправится сообщение

Creating a Custom Partitioner

- Implement interface **Partitioner**

```
public interface Partitioner extends Configurable, ... {  
    void configure(java.util.Map<java.lang.String,?> configs)  
  
    int partition(java.lang.String topic,  
                 java.lang.Object key,  
                 byte[] keyBytes,  
                 java.lang.Object value,  
                 byte[] valueBytes,  
                 Cluster cluster)  
  
    void close();  
}
```



configure is inherited from **Configurable**

- Mostly implement method **partition**
- Return number of message **target partition**

- To create a custom Partitioner, you should implement the **Partitioner** interface
 - This interface includes **configure**, **close**, and **partition** methods, although often you will only implement **partition**
 - **partition** is given the Topic, key, serialized key, value, serialized value, and cluster metadata
- It should return the number of the Partition this particular message should be sent to (0-based)



The part we don't show on the slide is that you need to register the custom partitioner with the **partitioner.class** producer config property.

Also, **configure** and **close** are optional lifecycle method called once by the producer client library when the producer starts and is closed, and **partition** is called for every message. So whatever the **partition** method does, it should ideally be fast, otherwise it can slow down the entire producer

например я могу все сообщения с определенным ключем (не зависимо от какого продьюсера оно пришло) отправлять только на партицию 0

- но это противоречит концепции load balancing, так как партиция может сломаться и тп

Custom Partitioner: Example

- Assume we want to store all messages with a particular key in one Partition, and distribute all other messages across the remaining Partitions

```
1 public class MyPartitioner implements Partitioner {
```

Custom Partitioner: Example

- Assume we want to store all messages with a particular key in one Partition, and distribute all other messages across the remaining Partitions

```
1 public class MyPartitioner implements Partitioner {  
2     public void configure(Map<String, ?> configs) {}  
3     public void close() {}  
4  
5     public int partition(String topic, Object key, byte[] keyBytes,  
6                           Object value, byte[] valueBytes, Cluster cluster) {  
7         List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  
8         int numPartitions = partitions.size();  
9  
10        if ((keyBytes == null) || (!(key instanceof String)))  
11            throw new InvalidRecordException("Record did not have a string Key");  
12  
13        if (((String) key).equals("OurBigKey")) ①  наше исключение (то ради чего мы делали свой  
14            return 0; // This key will always go to Partition 0  
15  
16        // Other records will go to the rest of the Partitions using a hashing function  
17        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;  
18    }  
19 }
```

- ① This is the key we want to store in its own Partition

This sample partitioner verifies that messages have a non-null String-type key. Then it returns 0 for messages with the specified key and a non-0 partition number for any other key, based on the standard hashing function.

This custom partitioner `MyPartitioner` would then be passed into the Producer Java code using the `partitioner.class` config parameter and is used by the `KafkaProducer` as part of the `send()` call. The developer code does not call the partitioner directly.

кастомный партишнер нужен, если я использую составной ключ
например чтобы CRUD операции для Order (1create 1update 1delete) попадали в одну партицию

- Зачем может понадобиться пользовательский способ секционирования? По некоторым причинам, из которых мы рассмотрим тут одну — применение составных ключей
- В таком случае необходимо написать пользовательский метод секционирования, который бы «знал», на основании какой части составного ключа выбирается секция
- При секционировании необходимо убедиться, что все транзакции для конкретного покупателя попадают в одну секцию, но сделать это с помощью ключа целиком не получится. Учет CRUD при секционировании даты приведет к множеству различных значений ключей для одного покупателя, поскольку разные СКГВ операции для одного заказа, в результате чего транзакции будут попадать в секции хаотичным образом. Все транзакции с одним идентификатором покупателя(номером заказа) должны попасть в одну секцию. Единственный способ добиться этого — применять при выборе секции

только идентификатор покупателя.

- Этот пользовательский класс секционирования расширяет класс DefaultPartitioner. Можно реализовать непосредственно интерфейс Partitioner, но в классе DefaultPartitioner уже реализована логика, которая нам пригодится.

Листинг 2.1. Составной ключ PurchaseKey

```
public class PurchaseKey {  
  
    private String customerId;  
    private Date transactionDate;  
  
    public PurchaseKey(String customerId, Date transactionDate) {  
        this.customerId = customerId;  
        this.transactionDate = transactionDate;  
    }  
  
    public String getCustomerId() {  
        return customerId;  
    }  
  
    public Date getTransactionDate() {  
        return transactionDate;  
    }  
}
```

Листинг 2.2. Пользовательский класс секционирования PurchaseKeyPartitioner

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {  
  
    @Override  
    public int partition(String topic, Object key,  
                        byte[] keyBytes, Object value,  
                        byte[] valueBytes, Cluster cluster) {  
        Object newKey = null;  
        if (key != null) {  
            PurchaseKey purchaseKey = (PurchaseKey) key;  
            newKey = purchaseKey.getCustomerId();  
            keyBytes = ((String) newKey).getBytes();  
        }  
        return super.partition(topic, newKey, keyBytes, value,  
                             valueBytes, cluster);  
    }  
}
```

Annotations for Listing 2.2:

- An annotation pointing to the line `PurchaseKey purchaseKey = (PurchaseKey) key;` with the text "Если ключ не пуст, извлекаем идентификатор покупателя".
- An annotation pointing to the line `newKey = purchaseKey.getCustomerId();` with the text "Присваиваем переменной newKey новое значение".
- An annotation pointing to the line `keyBytes = ((String) newKey).getBytes();` with the text "Делегируем вычисление секции для нового ключа методу родительского класса".

После написания пользовательского класса секционирования нужно сообщить Kafka о необходимости его применять вместо секционирования по умолчанию.

- пример конфигурации кафка продьюсера

```
partitioner.class=bbejeck_2.partition.PurchaseKeyPartitioner
```

надо также проверить что ключ равномерно распределен по данным

- Следует с осторожностью выбирать применяемые ключи и части пары «ключ/значение», по которым выполняется секционирование. Проверьте, чтобы выбранный вами ключ был распределен равномерно по всем вашим данным. В противном случае вы в конце концов столкнетесь с проблемой асимметрии данных, поскольку большая их часть будет располагаться лишь в небольшой части секций.

custom partitioner: альтернатива

16 декабря 2020 г. 18:04

вместо того чтобы писать свой партишенер, я могу в продьюсере сам присваивать номер партиции, куда я хочу отправить сообщение

An Alternative to a Custom Partitioner

- Specify Partition when defining `ProducerRecord`

```
ProducerRecord<String, String> record  
= new ProducerRecord<String, String>("my_topic", 0, key, value);
```

Writes message to Partition 0

Question: Which method is preferable?

- It is also possible to specify the Partition to which a message should be written when creating the `ProducerRecord`
- Answer:** Writing the partitioning logic directly into the producer code is simpler than creating a new class. However, the custom partitioner approach is more portable and makes it easier to share the custom code with other producer applications.

пример1 range partitioner

13 декабря 2020 г. 12:15

пример1 : одна и та же партиция постоянно получает один и тот же ключ

сделаю топик с тремя партициями и тремя консьюмерами (а затем один консьюмер упадет и новый поднимется)

```
kafka-topics --bootstrap-server kafka:9092 --create --partitions 3 --replication-factor 1 --topic test
# testing topic
kafka-console-producer --broker-list kafka:9092 --topic testing
kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --
topic testing
zookeeper-shell zookeeper:2181

# keyed messages
kafka-console-producer --broker-list kafka:9092 --topic testing --
property parse.key=true --property key.separator=,
kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --
topic testing --property print.key=true
# consumer group
kafka-console-producer --broker-list kafka:9092 --topic test --
property parse.key=true --property key.separator=,
kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --
topic test --property print.key=true --group first

kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --
topic test --property print.key=true --group second
consumer.config=vovah.properties
```

partition.assignment.strategy=org.apache.kafka.clients.consumer.StickyAssignor



пример2 : если убрать третьего консьюмера, то произойдет перебалансировка и его сообщению будет получать второй консьюмер



пример3а: если заново добавить третьего консьюмера, то он снова продолжит получать сообщения (с тем же ключем что и умерший консьюмер)



пример3б: когда все перемешалось, после удаления и добавления третьего консьюмера

The image displays four separate terminal windows, each showing a different command-line interaction with Apache Kafka.

- Top Left Terminal:** Shows the creation of a topic named "testing" with 3 partitions and a replication factor of 1. It then lists the topics in the cluster.
- Top Right Terminal:** Shows a consumer reading from the "testing" topic. It prints the key and value for each message, grouped by key. The output shows messages with keys "one", "two", "three", "four", and "five".
- Bottom Left Terminal:** Shows the creation of a topic named "test" with 3 partitions and a replication factor of 1. It then lists the topics in the cluster.
- Bottom Right Terminal:** Shows a consumer reading from the "test" topic. It prints the key and value for each message, grouped by key. The output shows messages with keys "one", "two", "three", "four", and "five".

Bottom Terminal (Windows Command Prompt):

```
[1] > [1] => [1] path
is
KeeperErrorCode = ConnectionLoss for /
Path must start with / character
invalid path string "/", caused by relative paths not allowed @1
Terminate batch job (Y/N)? y

vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan> setquota -h1-b val path
setquota: 'is' not recognized as an internal or external command,
operable program or batch file.
vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan> zkServer.sh start
[zk: localhost:2841] welcome to Zookeeper!
[zk: localhost:2841] the support is disabled
[zk: localhost:2841] is
KeeperErrorCode = ConnectionLoss for /
Path must start with / character
[zk: localhost:2841] y

vovaninOV C:\Users\vovan>
vovaninOV C:\Users\vovan> ./kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test --property print.key=true --group first
[2] > [2] => [2] 
Processed a total of 1 messages
Terminate batch job (Y/N)? y
vovaninOV C:\Users\vovan>
```

пример2

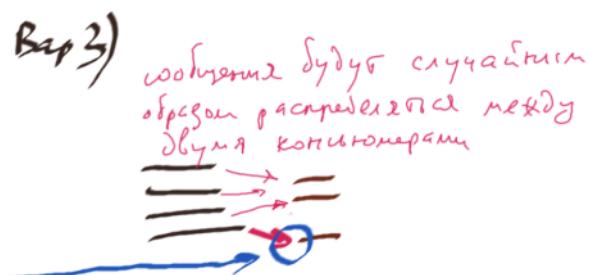
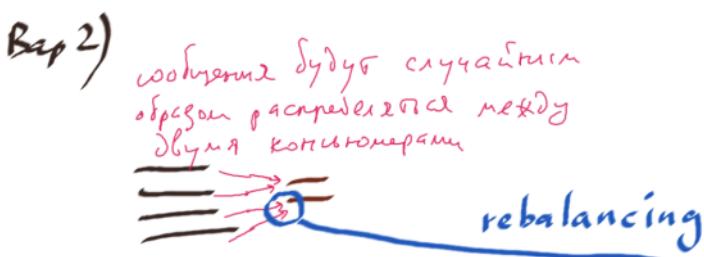
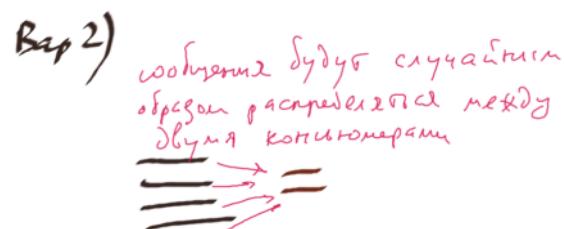
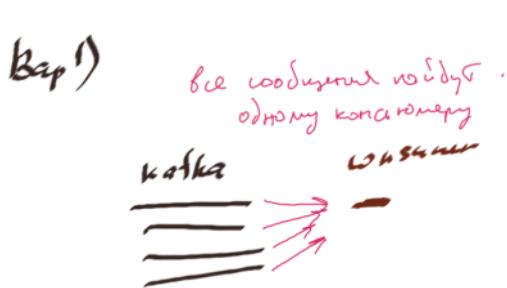
13 декабря 2020 г. 12:24

если просто ввести сообщения(без ключа и **C consumer group**) то все консьюмеры получат случайный набор (RoundRobin partitioning)

если просто ввести сообщения(без ключа и **без consumer group**) то все консьюмеры получат одинаковый набор



(case with no hashing key) **rebalancing** сообщения **автоматически** распределяются между консьюмерами при присоединении или отсоединении консьюмеров



```
x ~/kafka_2.12-2.0.0 ➤ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first_topic --group my-first-application
hello new consumer
message
message 2
message 3
yet another message

```

```
~/kafka_2.12-2.0.0 ➤ kafka-console-producer --broker-list 127.0.0.1:9092 --topic first_topic
>hello new consumer
>message
>message 2
>message 3
>one message
>another message
>yet another message
>
```

```
~/kafka_2.12-2.0.0 kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first_topic --group my-first-application
hello new consumer
message
message 2
message 3
yet another message
>
~/kafka_2.12-2.0.0 kafka-console-producer --broker-list 127.0.0.1:9092 --topic first_topic
>hello new consumer
>message
>message 2
>message 3
>one message
>another message
>yet another message
>
```

```
~/.kafka-console-consumer (java)
Last login: Fri Aug 24 11:18:03 on ttys003
~/kafka_2.12-2.0.0 kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first_topic --group my-first-application
one message
another message
```

(case with no hashing key) даже если в ново-подсоединенному-в-группу-консьюмере (в группе в которой уже были консьюмеры и они прочитали все сообщения) указать читать сообщения **from beginning** то он не получит ничего, так как группа уже прочитала все сообщения

```
~/kafka_2.12-2.0.0 kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic first_topic --group my-second-application --from-beginning
```

пример3 (CooperativeStickyAssignor)

13 декабря 2020 г. 23:22

The screenshot shows five terminal windows from the SecureCRT application. Each window has a title bar indicating it's connected to 'vovaserver' and a session number (0, 1, 2, 3, or 4). The windows are arranged in a grid-like fashion.

- vovaserver 0:** Shows the output of a 'kafka-console-producer' command. It lists four messages being sent to the 'test1' topic: '1', '2', '3', and '4'. Each message is followed by its offset: '1>1', '2>2', '3>3', and '4>4'. The command used was 'vovani@VOV C:\Users\vovan>kafka-console-producer --broker-list kafka:9092 --topic test1 --property key.separator=.'
- vovaserver 1:** Shows the output of a 'kafka-console-consumer' command. It lists four messages received from the 'test1' topic: '1', '2', '3', and '4'. Each message is followed by its offset: '1>1', '2>2', '3>3', and '4>4'. The command used was 'vovani@VOV C:\Users\vovan>kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test1 --property print.key=true --group second --consumer.config vovani.properties'.
- vovaserver 2:** Shows the output of another 'kafka-console-consumer' command. It lists four messages received from the 'test1' topic: '1', '2', '3', and '4'. Each message is followed by its offset: '1>1', '2>2', '3>3', and '4>4'. The command used was 'vovani@VOV C:\Users\vovan>kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test1 --property print.key=true --group second --consumer.config vovani.properties'.
- vovaserver 3:** Shows the output of a 'kafka-console-consumer' command. It lists four messages received from the 'test1' topic: '1', '2', '3', and '4'. Each message is followed by its offset: '1>1', '2>2', '3>3', and '4>4'. The command used was 'vovani@VOV C:\Users\vovan>kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test1 --property print.key=true --group second --consumer.config vovani.properties'.
- vovaserver 4:** Shows the output of a 'kafka-console-consumer' command. It lists four messages received from the 'test1' topic: '1', '2', '3', and '4'. Each message is followed by its offset: '1>1', '2>2', '3>3', and '4>4'. The command used was 'vovani@VOV C:\Users\vovan>kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test1 --property print.key=true --group second --consumer.config vovani.properties'.

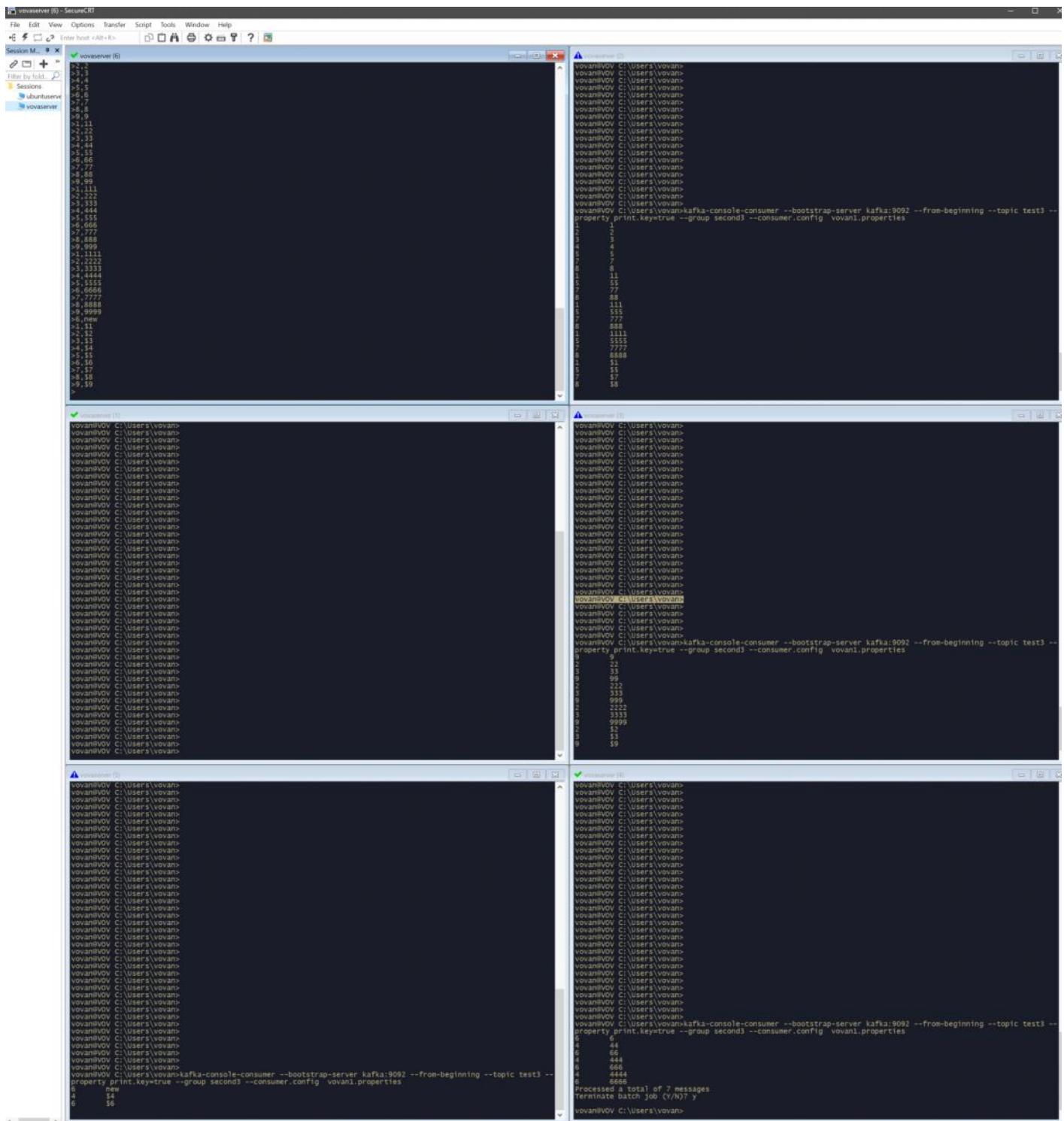
... пример3 (CooperativeStickyAssignor + timeouts)

13 декабря 2020 г. 23:28

```
kafka-topics --bootstrap-server kafka:9092 --create --partitions 3 --replication-factor 1 --topic test1
kafka-console-producer --broker-list kafka:9092 --topic test3 --property parse.key=true --property key.separator=,
kafka-console-consumer --bootstrap-server kafka:9092 --from-beginning --topic test3 --property print.key=true --group second3 --consumer.config vovan1.properties
```

```
vovan1.properties
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
session.timeout.ms=100000
heartbeat.interval.ms=30000
```

полностью сработало как ожидалось (те сообщения мгновенно не ушли на другой консьюмер а подождали пока поднимется замена), вначале была небольшая непонятка с сообщениями



пример4 (sticky)

13 декабря 2020 г. 23:24

The screenshot displays five terminal windows, each showing the output of a Kafka console consumer. The consumers are reading from topic 'test2' on Kafka broker 'kafka:9092'. The output shows messages being printed to the screen.

- Terminal 1:** Shows messages from partition 0. It includes configuration parameters like --bootstrap-server, --topic, and --group, followed by a list of messages with keys 1, 2, 3, 4, and 5.
- Terminal 2:** Shows messages from partition 1. It includes configuration parameters like --bootstrap-server, --topic, and --group, followed by a list of messages with keys 1, 2, 3, 4, and 5.
- Terminal 3:** Shows messages from partition 2. It includes configuration parameters like --bootstrap-server, --topic, and --group, followed by a list of messages with keys 1, 2, 3, 4, and 5.
- Terminal 4:** Shows messages from partition 3. It includes configuration parameters like --bootstrap-server, --topic, and --group, followed by a list of messages with keys 1, 2, 3, 4, and 5.
- Terminal 5:** Shows messages from partition 4. It includes configuration parameters like --bootstrap-server, --topic, and --group, followed by a list of messages with keys 1, 2, 3, 4, and 5.

In all terminals, the messages are identical, showing the same key-value pairs (1, 2, 3, 4, 5) repeated across partitions. This demonstrates that the consumer is reading from multiple partitions simultaneously, which is the purpose of the 'sticky' consumer strategy.