Formation 4SH

Kafka - Fundamentals

Version 1.0

Introduction

Objectifs

A la fin de la formation vous serez capable de...

- Comprendre comment fonctionne un cluster **Kafka** et en connaître ses différents composants
- Ecrire des **Producers** et **Consumers**
- Mettre en place une stratégie de gestion des schémas
- Développer des “streaming apps” simples avec **Kafka Streams** et **ksqlDB**
- Prendre des décisions sur l’**Acknowledgement**, les **Partitions** , le **Batching**, la **Retention** et la **Replication**
- Intégrer rapidement Kafka avec des systèmes externes grâce à ksqlDB

Programme

1er jour

Intro (9h - 9h30)

Fundamentals - Slides (9h30 - 11h00)
☕ Pause café (11h – 11h15)
Fundamentals - TP (11h15 - 11h45)
Produire des Messages - Slides (11h45 - 12h15)

🥖 Repas (12h15 - 13h45)

Produire des Messages - Slides (13h45 - 14h45)
Produire des Messages - TP (14h45 - 15h15)
Consommer des Messages - Slides (15h15 - 16h00)
☕ Pause café (16h - 16h15)
Consommer des Messages - Slides (16h15 - 17h00)
Consommer des Messages - TP (17h00 - 17h30)

2ème jour

Schema Management - Slides (9h00 - 10h30)
Schema Management - TP (10h30 - 11h00)
☕ Pause café (11h – 11h15)
Kafka Connect - Slides (11h15 - 12h15)

🥖 Repas (12h15 - 13h45)

REST Proxy (13h45 - 14h15)
Kafka Streams - Slides (14h15 - 15h15)
Kafka Streams - TP (15h15 - 16h00)
☕ Pause café (16h - 16h15)
ksqlDB - Slides (16h15 - 17h15)
ksqlDB - TP (17h15 - 17h45)

Présentation

- Votre expérience avec Kafka
- Des expériences avec d’autres systèmes de messagerie
- Vos langages de programmation
- Vos attentes

Fondamentaux

Fondamentaux - Objectifs

- Etre capable de décrire l’architecture haut niveau de Kafka
- Comprendre les concepts de Broker, Topic, Partitions, Records, etc
- Comprendre par quels moyens Kafka est résilient et tolérant à la panne

Event-Driven System

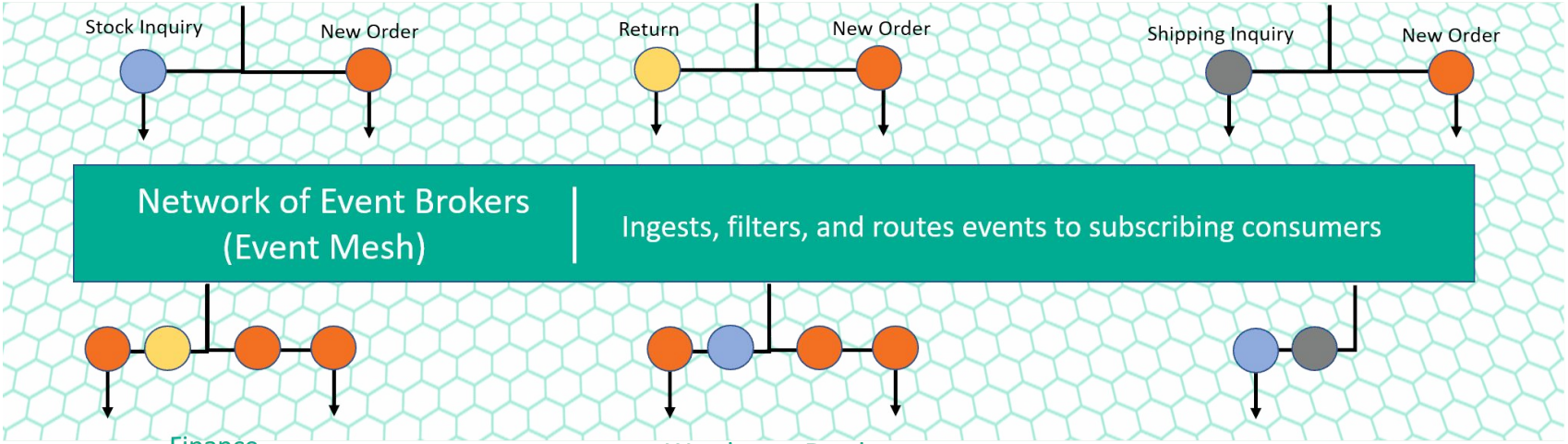
Event **producers** generate events through purchases, inquiries, and other actions

Event producers are decoupled from consumers and events are broadcast to subscribers

Mobile Application

Point-of-Sale System

eCommerce Website



Finance

Event **consumers** subscribe to relevant events that affect their systems/actions

Warehouse Database

Events trigger real-time responses and can generate subsequent events

Customer Service

Qui fait de l'Event-Driven

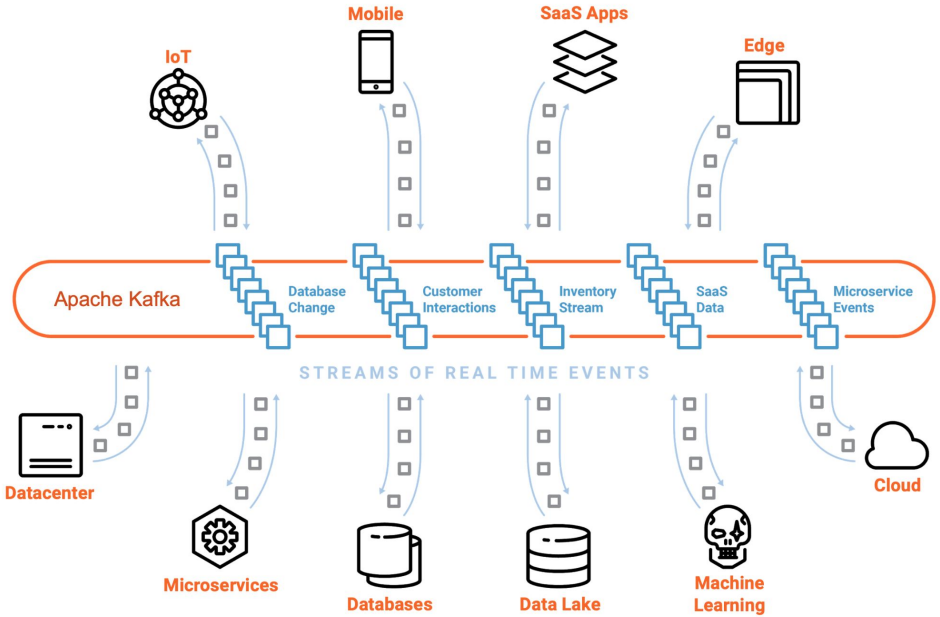
- 🕸️ Les réseaux sociaux
- 🏦 Les banques
- 🚚 Logistique/transport
- 📺 Streaming Provider
- 🌐 VPC
- etc

Qui fait de l'Event-Driven

Historique Kafka

- Créé en 2011 chez LinkedIn
- A rejoint l'incubateur Apache en 2012
- Écrit en Java et Scala
- Confluent a été fondé en 2014

Motivation

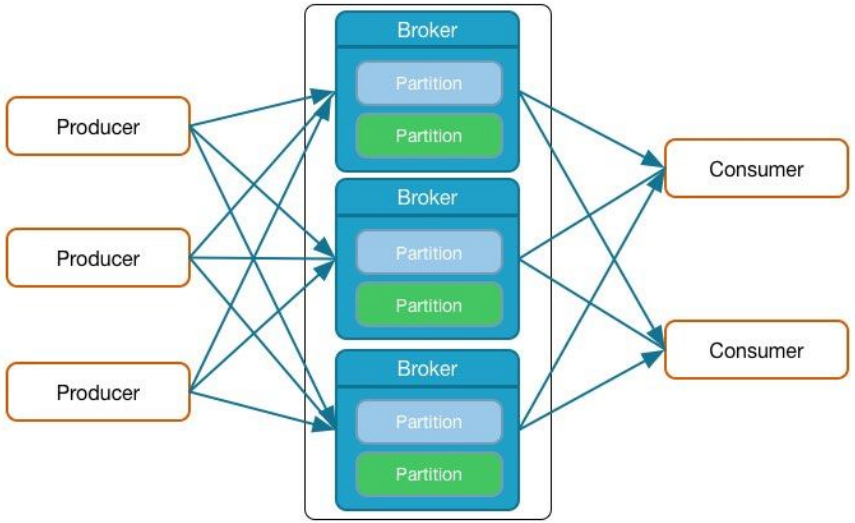


Quand à-t-on besoin d'une Real Time Streaming Platform:

- Glocal-scale
- Real-time processing
- Persistent storage
- Stream processing
- High performances

Brokers

- Les **Producers** envoient des **Messages** aux **Brokers**
- Les **Brokers** reçoivent et stockent les **Messages**
- Un **Cluster** Kafka peut être constitué de 1..N **Brokers**
- Un **Broker** gère de multiples **Partitions**



Découplage des Consumers et Producers

- Les **Producers** et les **Consumers** sont découplés
- Les **Consumers** qui sont lents n'affectent pas les autres **Consumers**/**Producers**
- Les nouveaux **Consumers** peuvent lire les données depuis le début des topics
- Un **Consumer** down n'affecte pas le système
- Les **Brokers** n'ont pas connaissance des **Producers**/**Consumers**

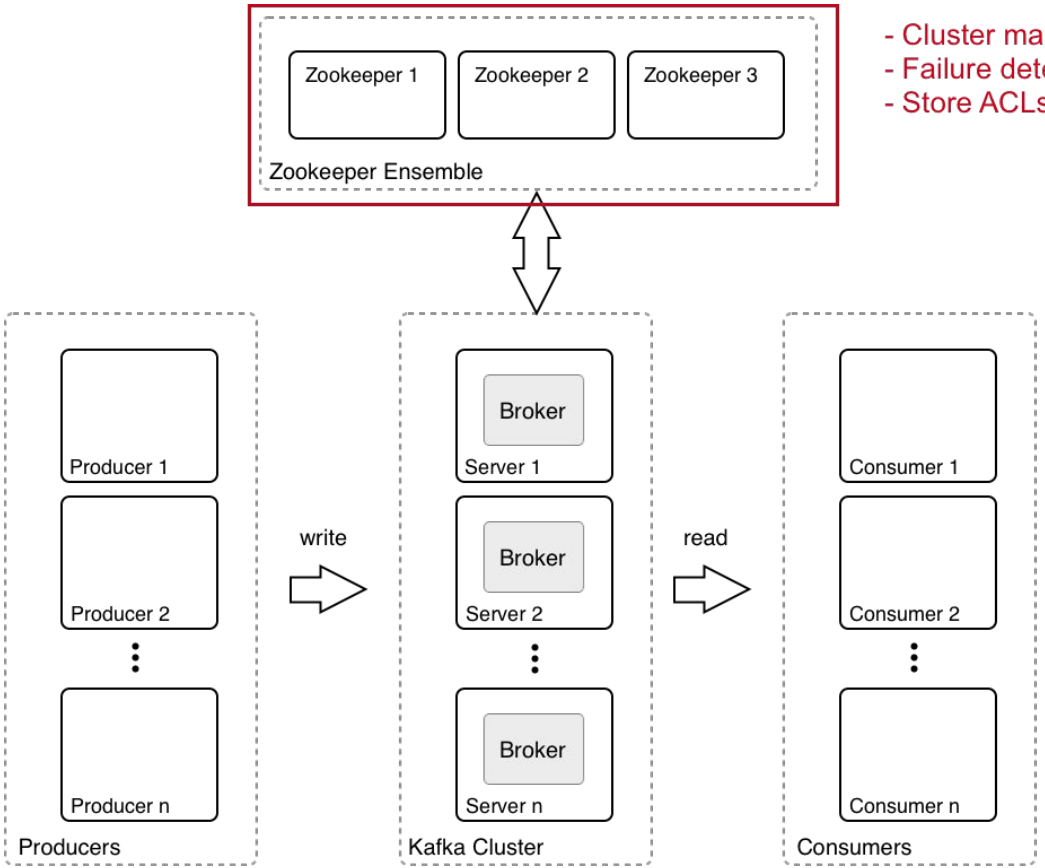
C’est quoi Zookeper ?

- Un Key Value Store distribué (3 à 5 instances recommandé)
- Open Source Apache Project
- Utilisé par Kafka, Neo4j, Mesos, etc

A quoi sert Zookeeper ?

- Cluster Management (synchronization entre les **Brokers**)
- Stocker les ACLs (Access Control Lists)
- Failure détection and recovery (ex: quand un **Broker** est down)

A quoi sert Zookeeper ?



- Cluster management
- Failure detection & recovery
- Store ACLs & secrets

Kafka sans Zookeeper ?

- Kafka 2.8 => Kafka peut fonctionner sans Zookeeper (preview)
- Kafka 3.x => Kafka peut fonctionner avec ou sans Zookeper
- Kafka 4.x => Support Zookeeper deprecated

C’est quoi un topic ?

- Représentation logique d’un flux de données
- Définis par les développeurs
- Pas de limite au nombre de **Topics** (⚠️ un trop grand nombre de partitions peut avoir un impact sur les performances des **Brokers**)
- **Producer** <-> **Topic** : N to N relation

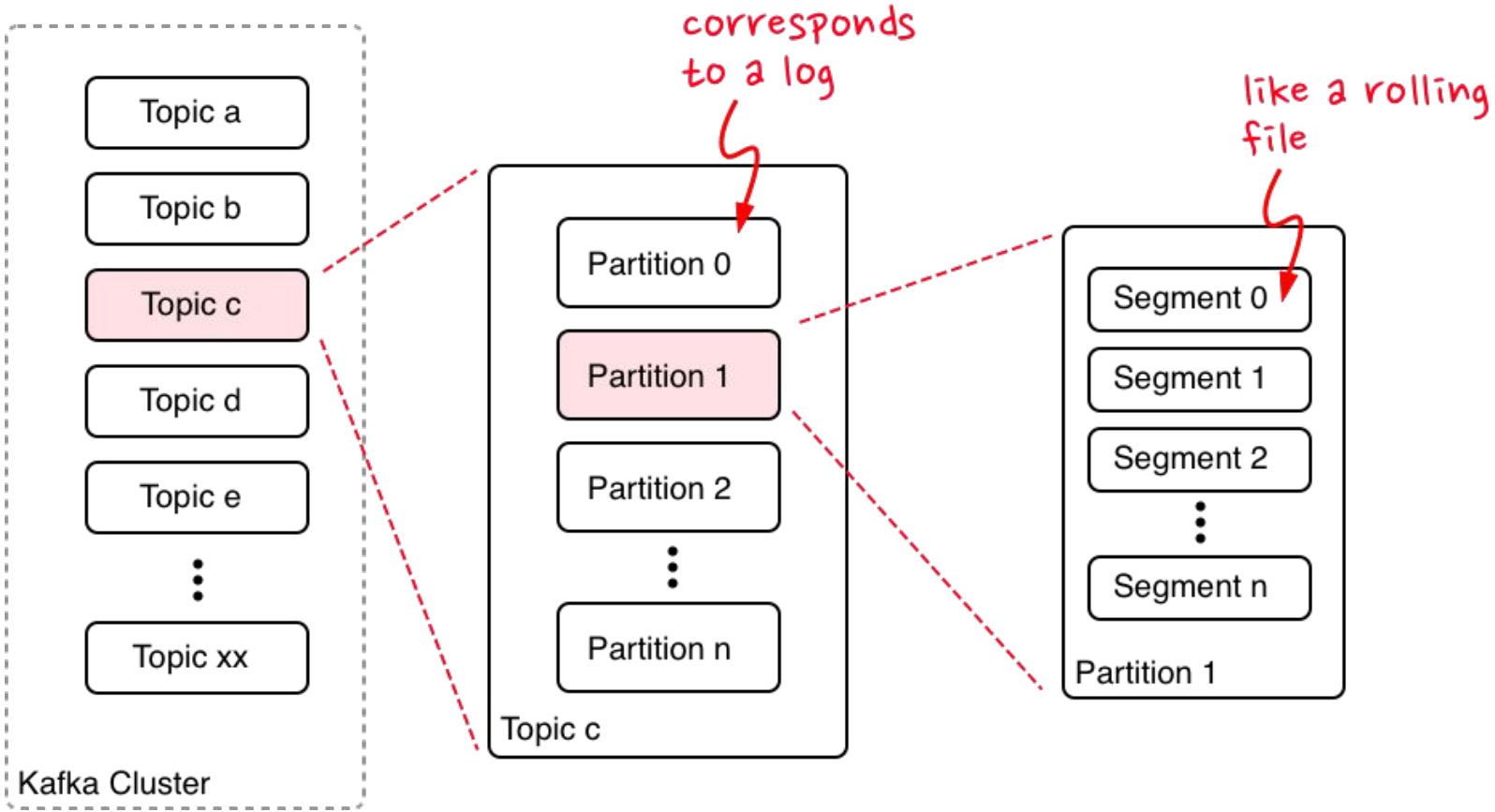
C’est quoi une partition ?

- Un **Topic** est divisé en 1..N **Partitions**
- Le nombres de **Partitions** d’un **Topic** peut être augmenté, mais pas diminué
- Les **Partitions** permettent de paralléliser la lecture et l’écriture des données dans un **Topic**
- Répartition des **Messages** dans les **Partitions** :
  - Par défaut, un **hash** de la clé du message détermine la partition
  - Si pas de clé => **Round Robin**
  - Partitionner custom côté **Producer**
- ⚠️ L’ordre des messages n’est garanti qu’au niveau **Partition**, pas **Topic** ⚠️

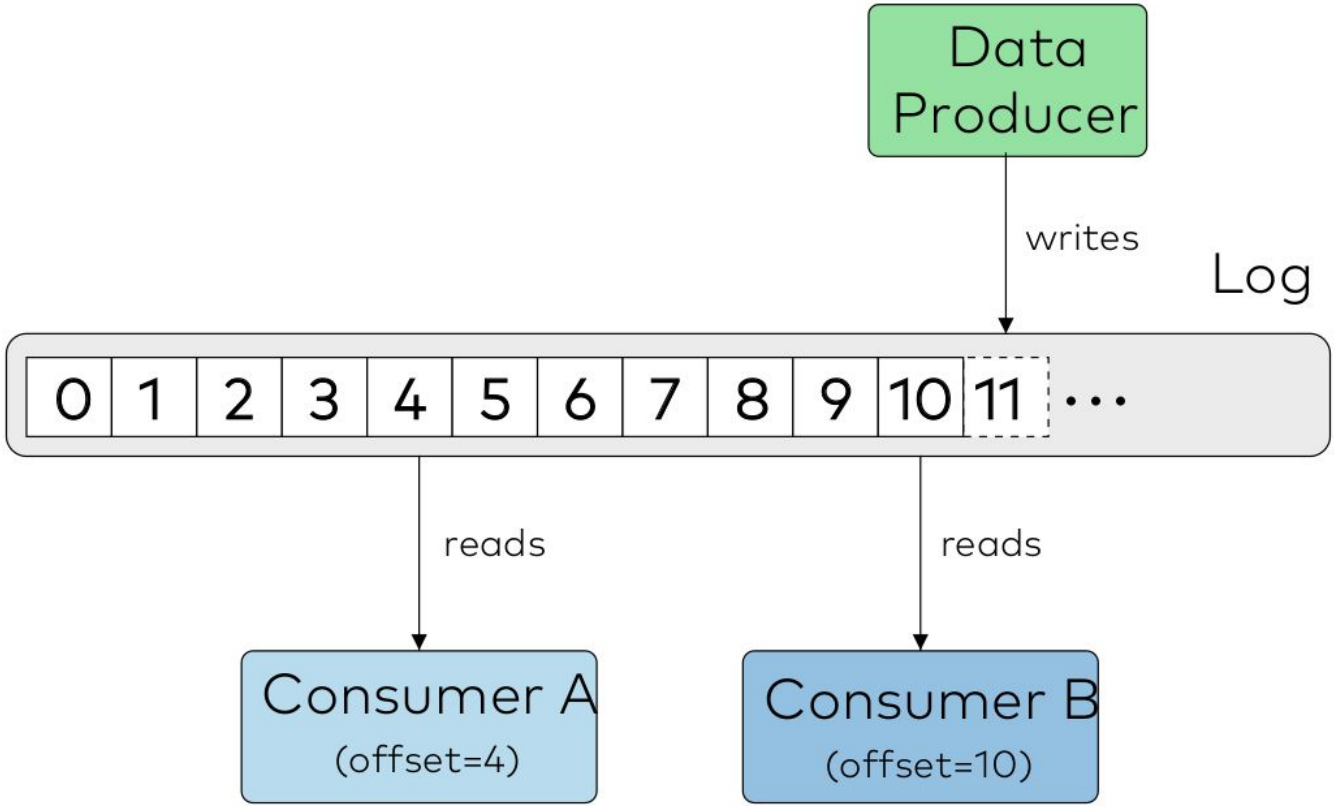
C’est quoi un segment ?

- Les **Brokers** stockent les messages tels qu’ils arrivent in memory (page cache)
- Les **Brokers** utilisent une stratégie de “rolling-file” => ces fichiers sont les **Segments**
- Lorsqu’un **Segment** est plein (ou que sa durée max expire) un nouveau **Segment** est créé
- Le découpage des **Partitions** en **Segments** permet de gérer facilement la rétention des données

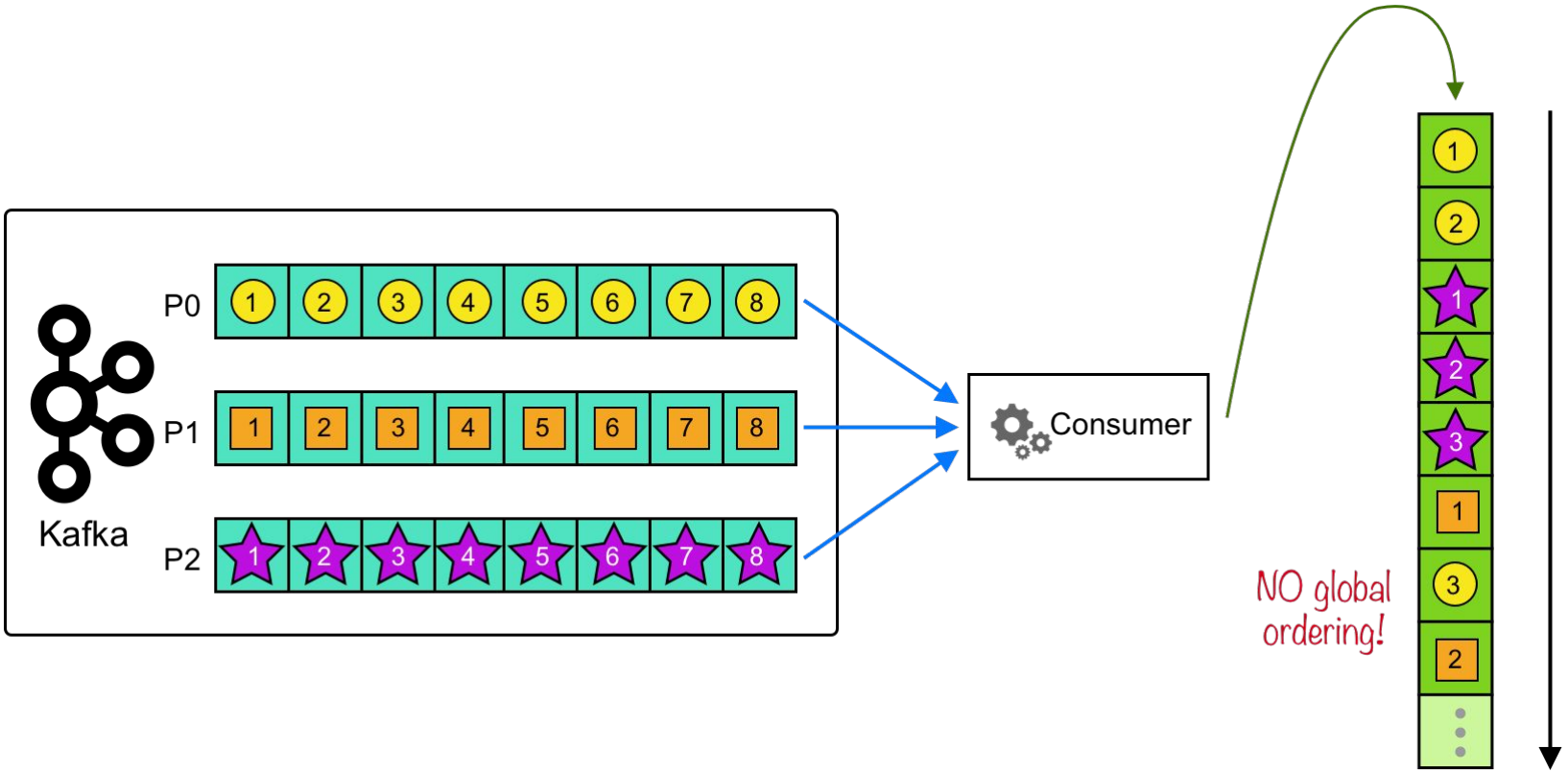
Topics / Partitions / Segments



Commit Log


Data Producer
writes
Log
0
1
2
3
4
5
6
7
8
9
10
11
reads
reads
Consumer A
(offset=4)
Consumer B
(offset=10)

Ordering



Ordering

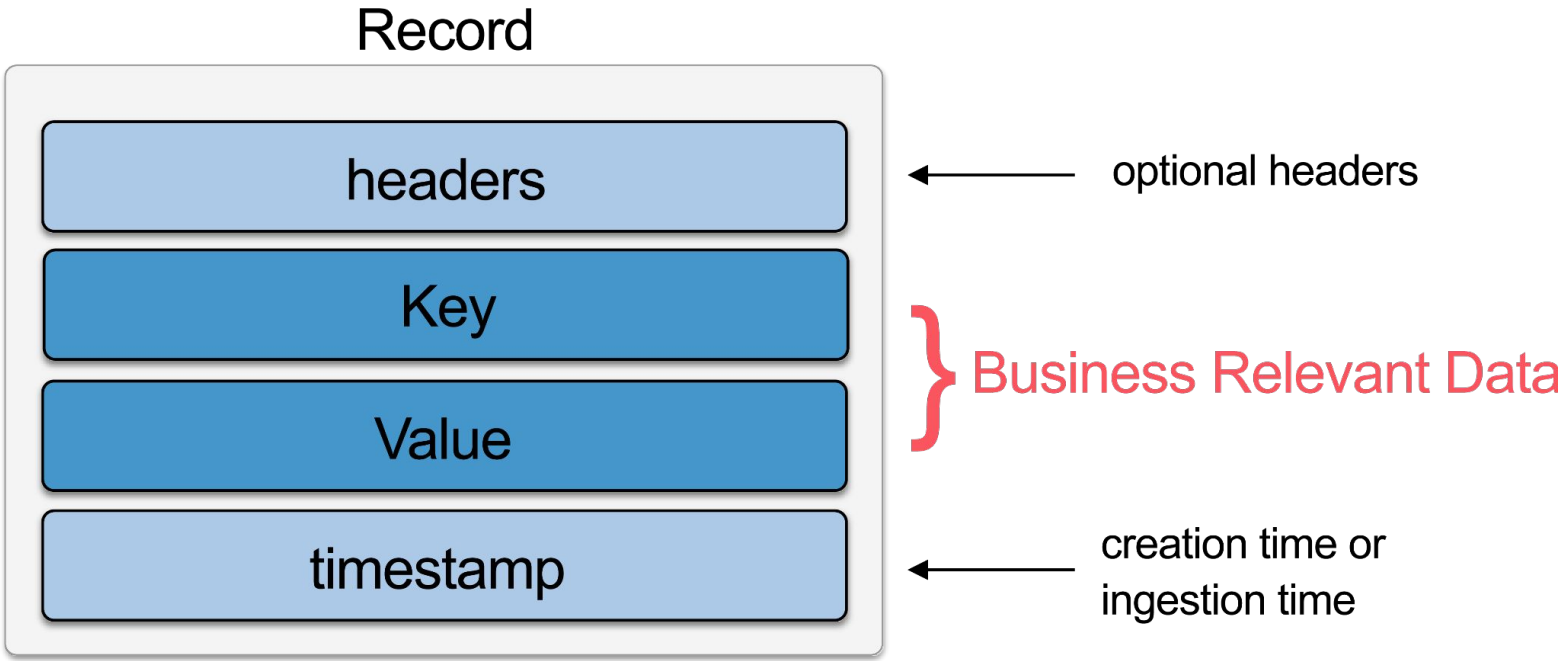
Si l'ordre des **Messages** au sein d'un topic est primordial, alors :

- Choisir avec attention la **Key** des **Messages**
- Utiliser un seul **Producer**
- S'assurer que l'ordre est préservé par l'application appelant le **Producer**

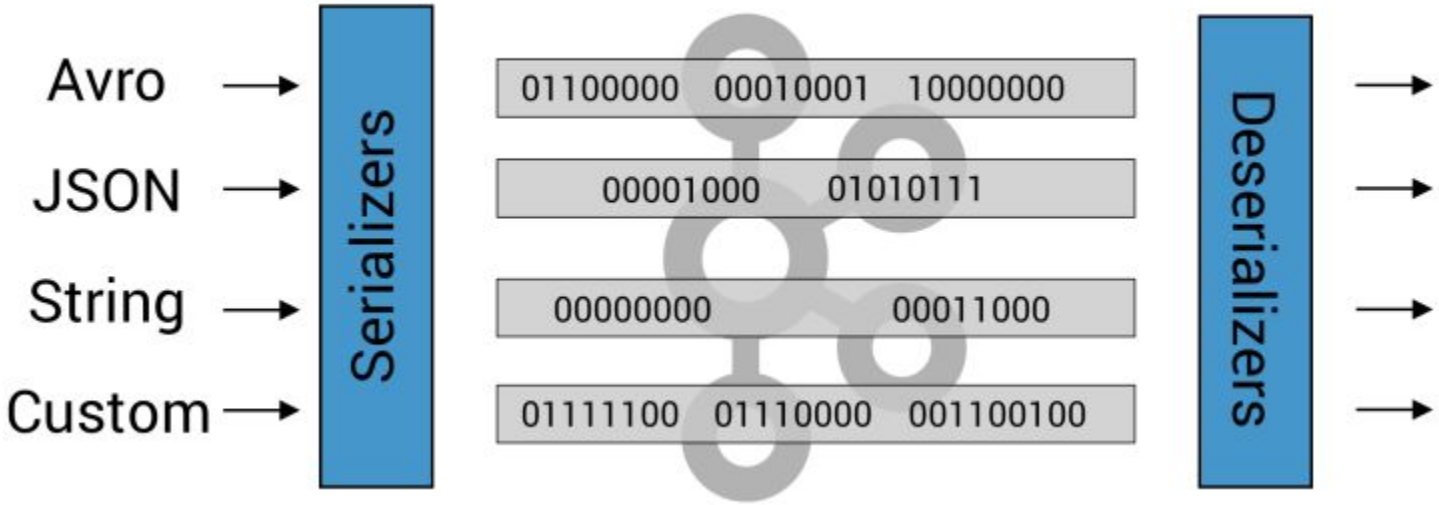
Record

- Un **Record** est souvent également appelé un **Message** ou un **Event**
- Il est constitué de **Metadata** et d'un **Body**
- Les Metadata contiennent l'**Offset**, la compression, le timestamp, le magic byte et les **Headers**
- Le **Body** est constitué de la **Key** et de la **Value** du **Record**

Record


Record
headers
Key
Value
timestamp
optional headers
Business Relevant Data
creation time or
ingestion time

Serialization/Deserialization



Serialization/Deserialization

- Les données sont stockées dans des tableaux de bytes
- Les données doivent être sérialisées par les **Producers**
- Les données récupérées par les **Consumers** doivent être désérialisées

Formats

- Avro
- Protobuf
- JSON
- String
- Custom

Cleanup policy

Il existe 2 cleanup policies (cleanup.policy) :

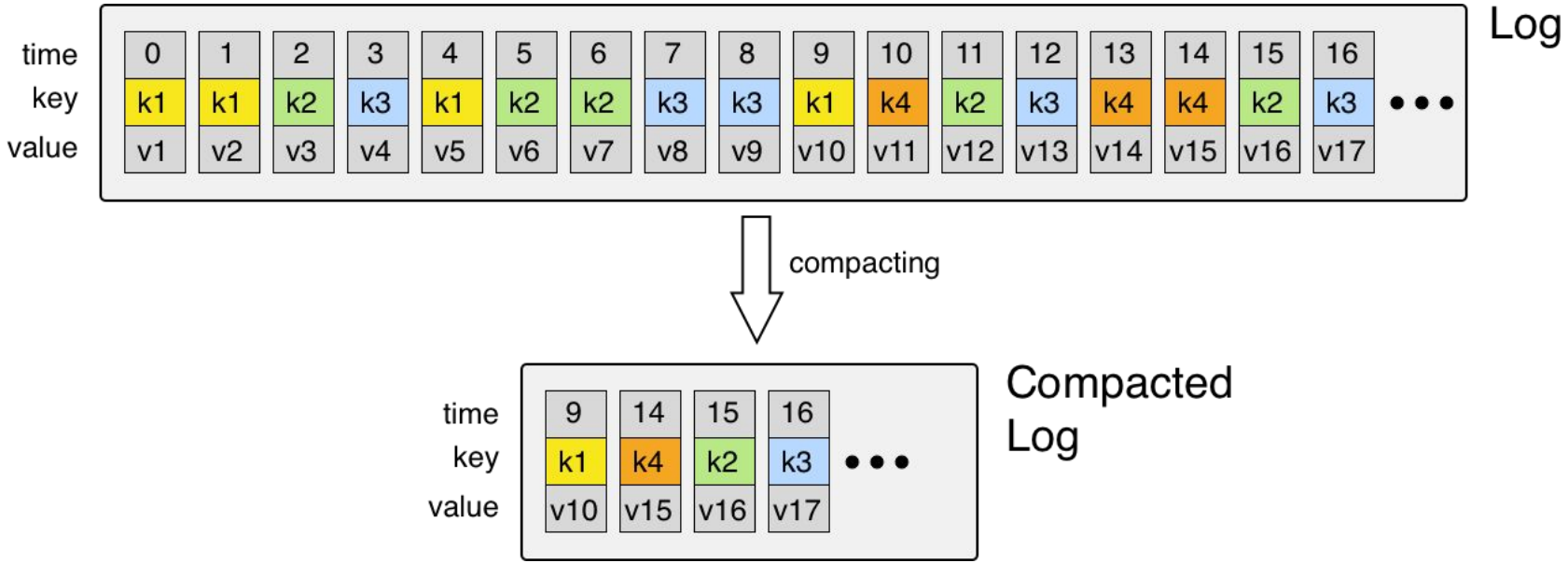
- delete : lorsque la condition est remplie, le **Segment** est supprimé
- compact : seulement le dernier **Message** d'une clé donnée est gardé
  - Use cases : Event sourcing, CDC, Real-time lookups during Stream processing, etc

La cleanup policy est configurable au niveau **Broker** et **Topic**

ℹ️ Le cleanup ne supprime jamais les messages du **Segment** actif

ℹ️ Le cleanup s'applique toujours à un **Segment** entier, pas à des **Messages** individuels

Log compaction



Data retention

Retention

Supprimer les **Segments** qui sont **trop vieux** :

Le **Segment** est supprimé si le message le plus récent est plus vieux que retention.ms

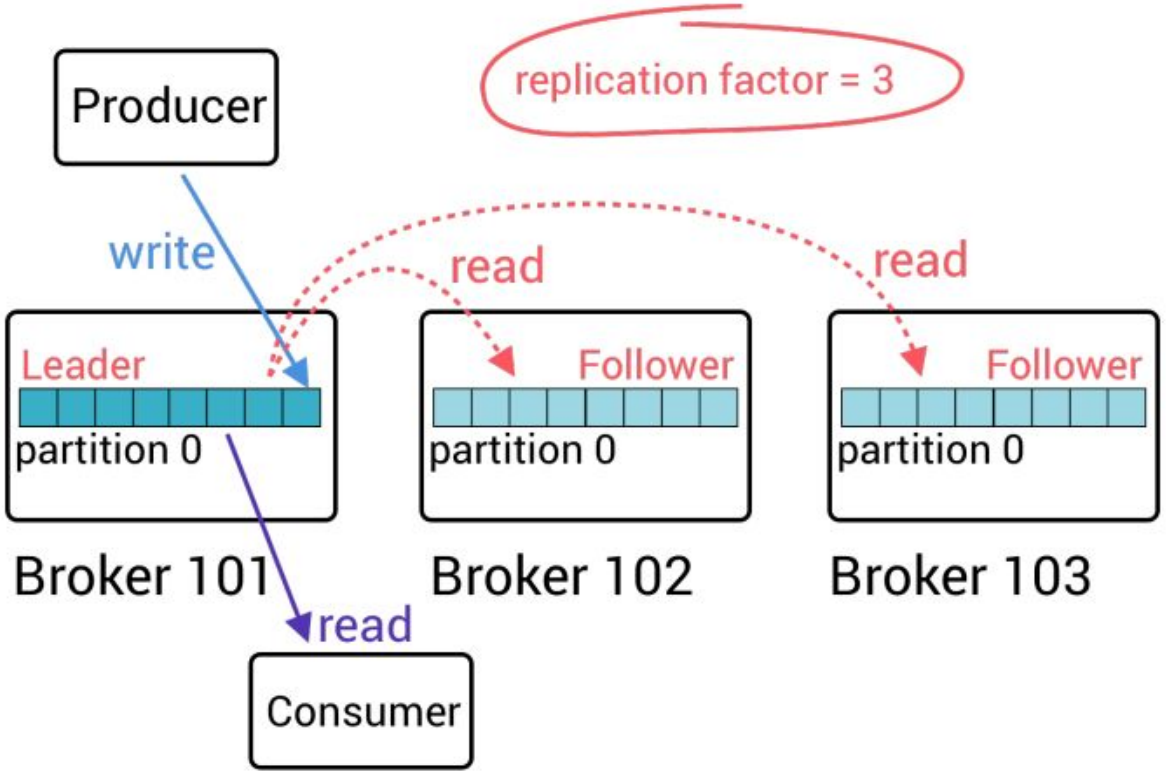
- Valeur par défaut : 7 jours

Supprimer les Segments qui sont trop volumineux :

Le Segment est supprimé si sa taille dépasse : retention.bytes

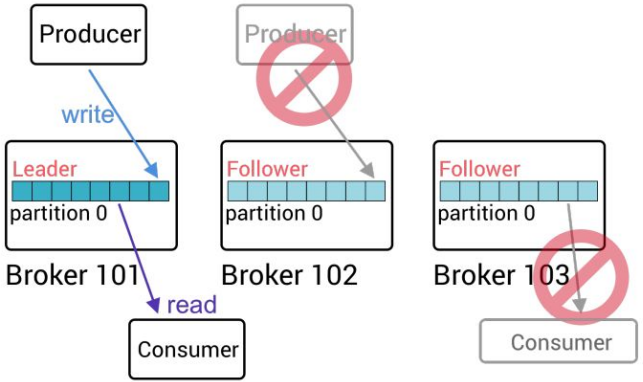
- Valeur par défaut : -1 (illimité)

Replication factor



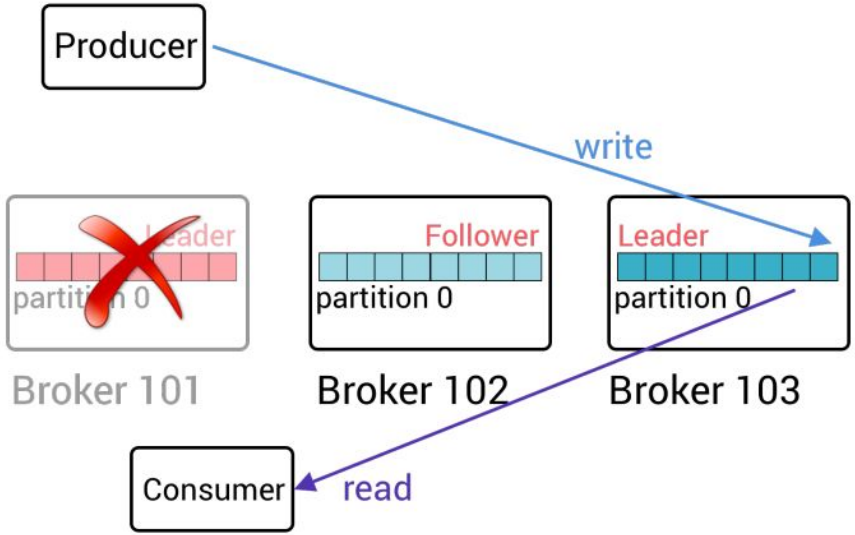
Replication factor

- Le nombre de replicas est configurable au niveau **Topic**
- Les replicas permettent à **Kafka** d'être "fault tolerant"
- L'ordre des **Messages** est identiques sur toutes les **Partitions**
- Pour chaque **Partition**, un **Broker** est élu **Leader** de la **Partition**
  - Toutes les écritures et lectures se font sur le Leader
  - Les autres **Brokers** sont considérés comme **Follower** sur cette partition
  - Il est possible d'activer le "follower fetching" qui permet aux **Consumers** de lire depuis les **Followers**
- Les **Followers** copient les données du commit log du **Leader** par le biais de "fetch request"



Leader failover

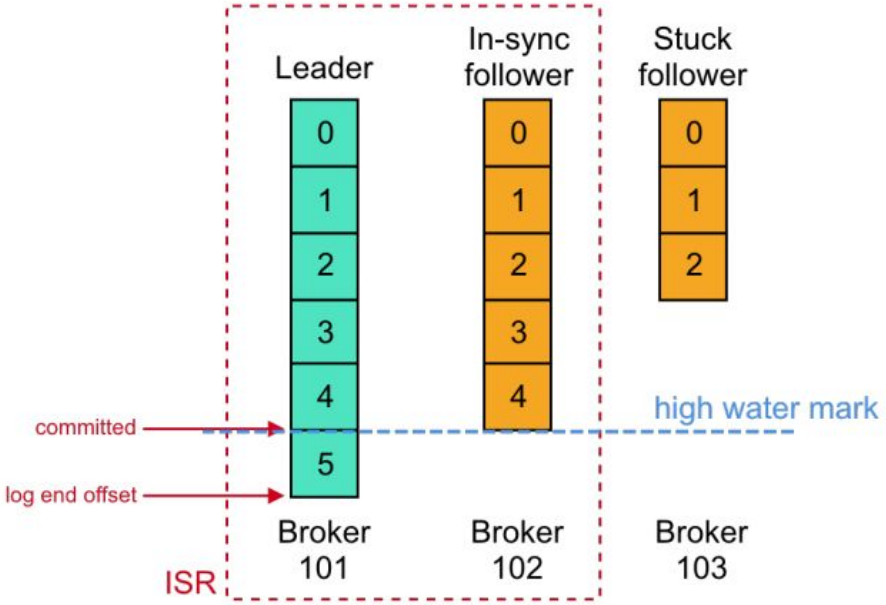
- Lorsqu'un **Broker** tombe, le cluster va élire un nouveau **Leader** pour chaque **Partition** on il était **Leader**
- Les clients basculent automatiquement sur le nouveau **Leader**
- **Kafka** essaye de répartir au mieux les **Leader** au sein du cluster



In-Sync Replicas

- Les **In-Sync Replicas** (**ISR**) sont une liste de replicas (**Leader** et **Follower**) qui sont identiques jusqu'à un point spécifique appelé "**high water mark**"

- Si le **Leader** tombe, c'est la liste des **ISR** qui est utilisée pour l'élection du nouveau **Leader**



Partition Leader discovery

- Lorsqu'un **Client** (**Producer** ou **Consumer**) se connecte, il émet une "metadata request" auprès d'un **Broker**
- Les informations des **Partitions** sont stockés en cache sur chaque **Broker**
    - N'importe quel **Broker** peut répondre à une "metadata request"
- Une fois que le **Client** a reçu la réponse, il sait qui est le **Leader** de chaque **Partition**
- Lorsqu'un **Broker** tombe, le **Client** reçoit une erreur et va automatiquement ré-émettre une "metadata request"

Security

- Kafka supporte l'**Encryption in Transit** (SSL)
- Kafka supporte l'**Authentication** et l'**Authorization** (SSL/SASL & ACL)
- No **Encryption at Rest** out of the box!
  - Si nécessaire, il est recommandé de faire du chiffrement côté **Client** ou au niveau du disque de stockage des **Brokers**
- Il est possible d'avoir des **Clients** avec et sans **Chiffrement**/**Authentication**

Izivia - Gestion des Topics

Topics physiques vs Topics applicatifs

- Sur Izivia, il existe des **Topics** physiques et des **Topics** applicatifs :
  - Les **Topics** physiques représentent les **Topics** qui seront stockés sur les **Brokers**
  - Les **Topics** applicatifs sont les **Topics** utilisés par les **Producers** et les **Consumers**

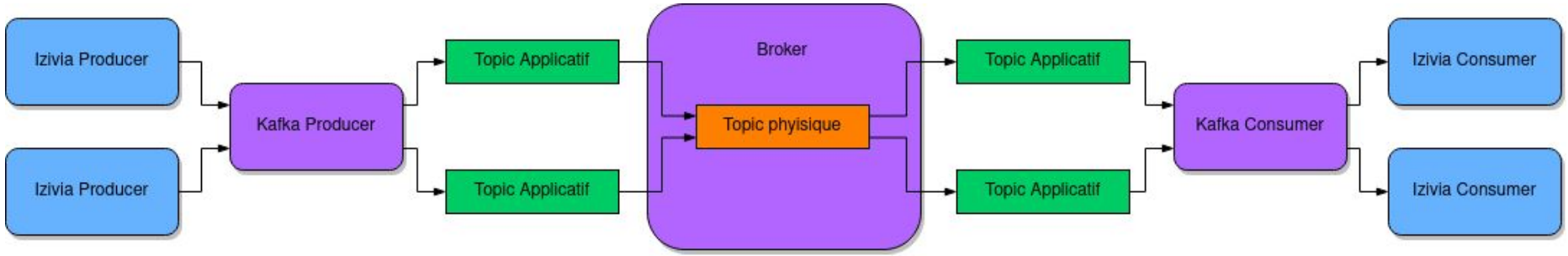
- L'objectif est de réduire le nombre de **Partitions** en groupant les **Messages** de plusieurs **Topics** applicatifs dans un seul **Topic** physique

- Le fichier de configuration kafka-platform-configuration.yaml permet de faire le mapping entre **Topics** physiques et **Topics** applicatifs :

```
appTopics:
   - appTopic: hardware-csms-status-notification-event-v1
     defaultKafkaTopic: infra-v1
```

Izivia - Gestion des Topics

Topics physiques vs Topics applicatifs



Lab

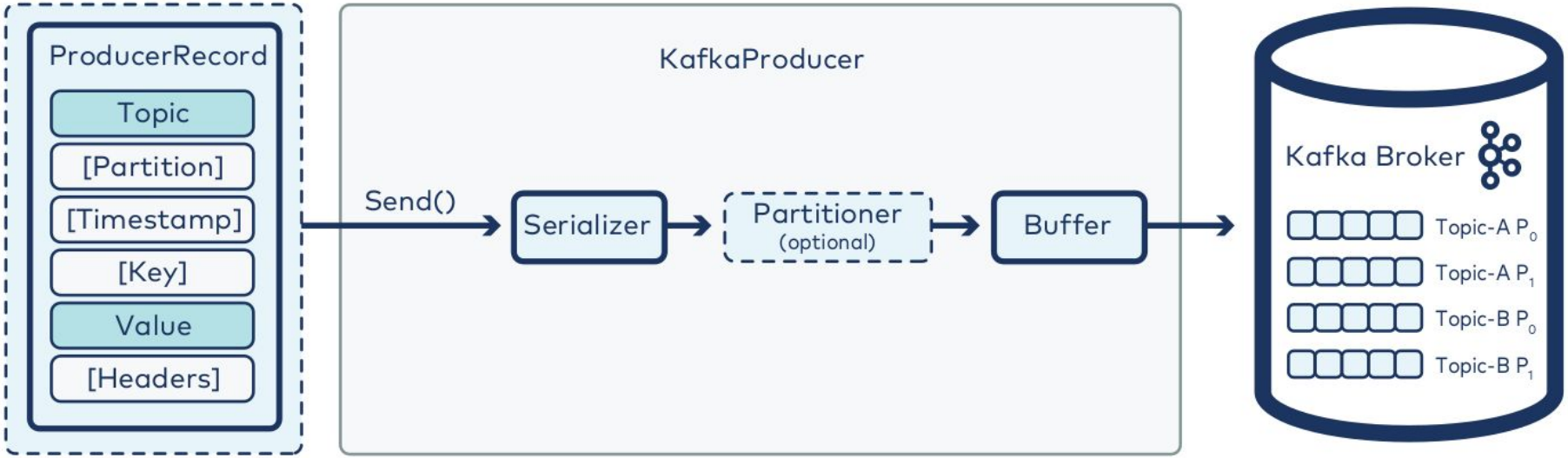
Produire des messages

Kafka Clients libs

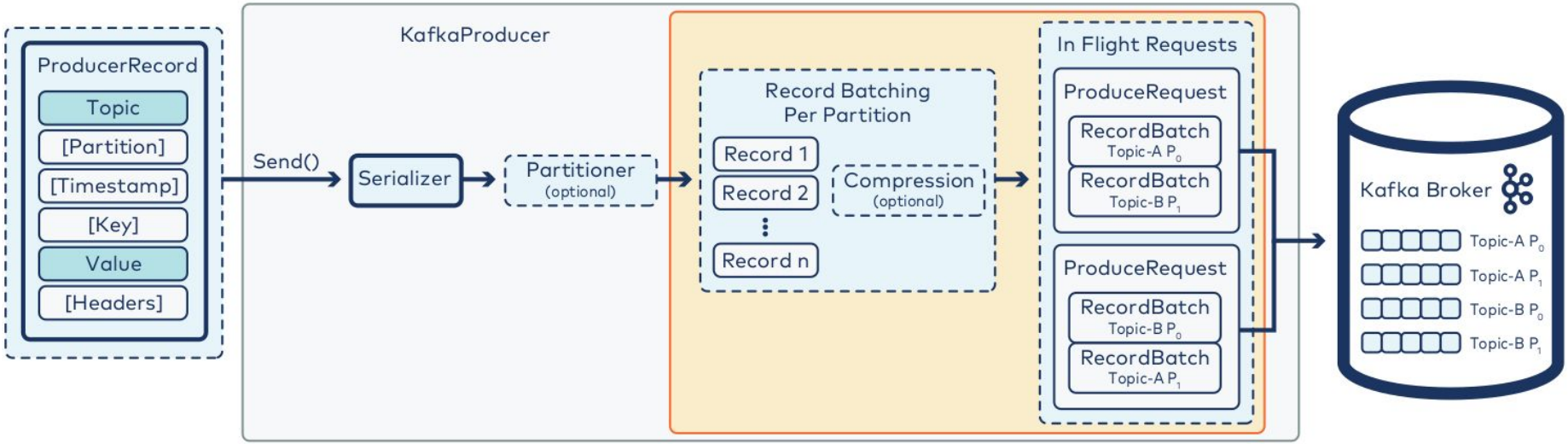
JVM

librdkafka (C library)

Producer Design



Producer Design



Producer Design

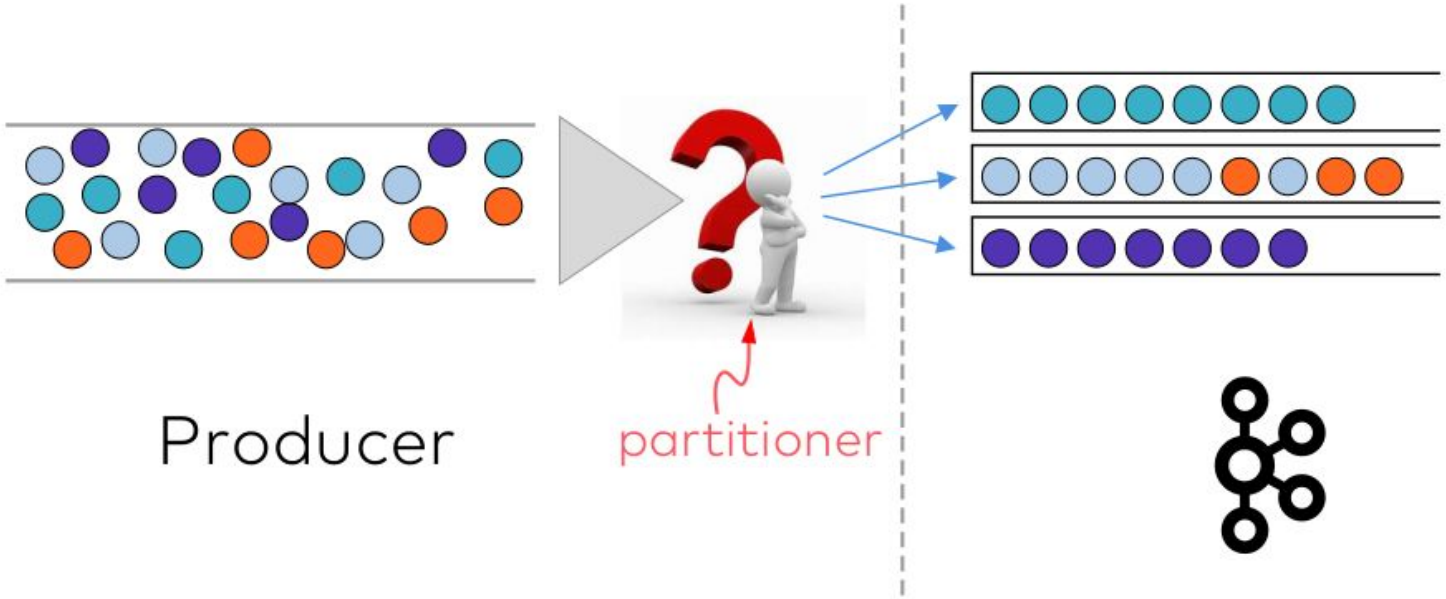
- Le **Message** est sérialisé en un bytes array
- Le **Message** passe par le **Partitioner** qui décide dans quelle partition il doit être envoyé
- Le **Message** n'est pas envoyé directement. Le **Producer** prépare un batch de **Messages**
- Lorsqu'un **Batch** est "plein", le **Producer** le compresse (optional) et envoie (flush) le **Batch** vers le **Broker**
- Si le **Broker** a réussi à stocker le **Batch**, il répond avec un **ACK** et les metadata
- Si il n'a pas réussi, il répond avec un **NACK**
  - Dans ce cas, le **Producer** va immédiatement réessayer de renvoyer le **Batch**
  - Si le **Producer** n'arrive toujours pas à envoyer le **Batch** après le nombre max de tentatives, il lève une exception

Producer properties

Name	Description
bootstrap.servers	La liste des host:port des **Brokers** utilisés pour la connection initiale
key.serializer	Classe utilisée pour sérialiser la **Key**. Doit implémenter Serializer
value.serializer	Classe utilisée pour sérialiser la **Value**. Doit implémenter Serializer
enable.idempotence	Active l'idempotence du **Producer**
compression.type	Type de compression à utiliser sur les **Batchs**. Valeurs possibles : none, snappy,, gzip, lz4, zstd. Default : none
acks	Permet de déterminer si une "write request" est réussie. Valeurs possibles : 0, 1, all. Default: all
delivery.timeout.ms	Permet de configurer une durée max pour laquelle la méthode send() doit répondre avant d'être considérée en timeout. Cela permet de contrôler les retries des **Producers**
batch.size	La taille max d'un **Batch** (bytes). Default: 16384
linger.ms	Le temps qu'un **Batch** va attendre avant d'être envoyé. Default: 0

Default partitioning

Partition = hash(key) % Number of Partitions



Custom partitioner

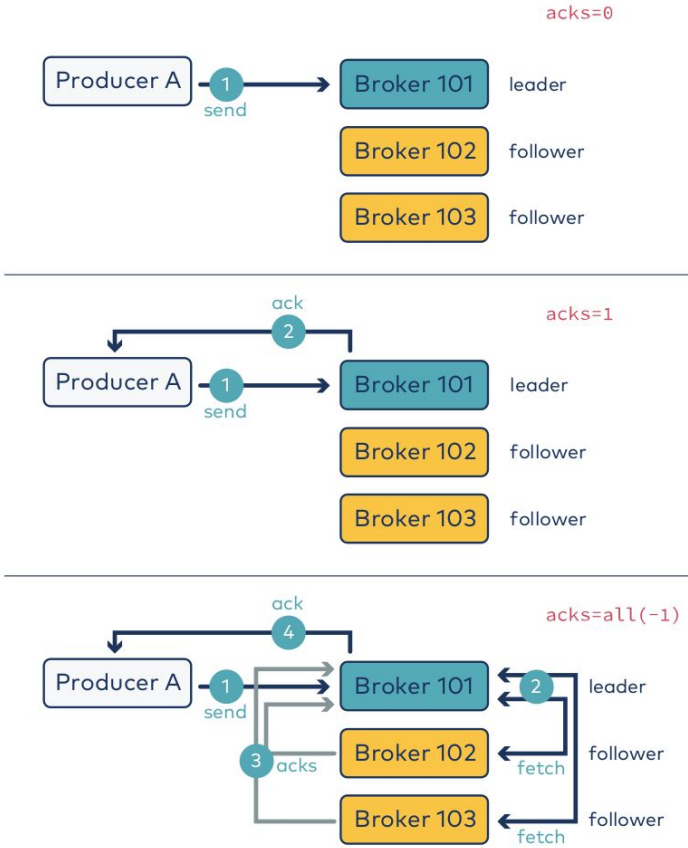
```
public class MyPartitioner implements Partitioner {

   public void configure(Map<String, ?> configs) {}
   public void close() {}
   public void onNewBatch() {}

   public int partition(
       String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster
   ) {
       List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
       int numPartitions = partitions.size();
       if ((keyBytes == null) || (!(key instanceOf String))){
           throw new InvalidRecordException("Record did not have a string Key");
       }
       if (((String) key).equals("OurBigKey")) {
           return 0; // This key will always go to Partition 0
       }
       // Other records will go to the rest of the Partitions using a hashing function
       return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)) + 1;
   }
}
```

Partitioning

Choix de la partition lors de l'envoi du Message

```
// Record will be sent to Partition 0
ProducerRecord<String, String> record =
     new ProducerRecord<String, String>("my_topic", 0, key, value);
```

Acknowledgement



Acknowledgement

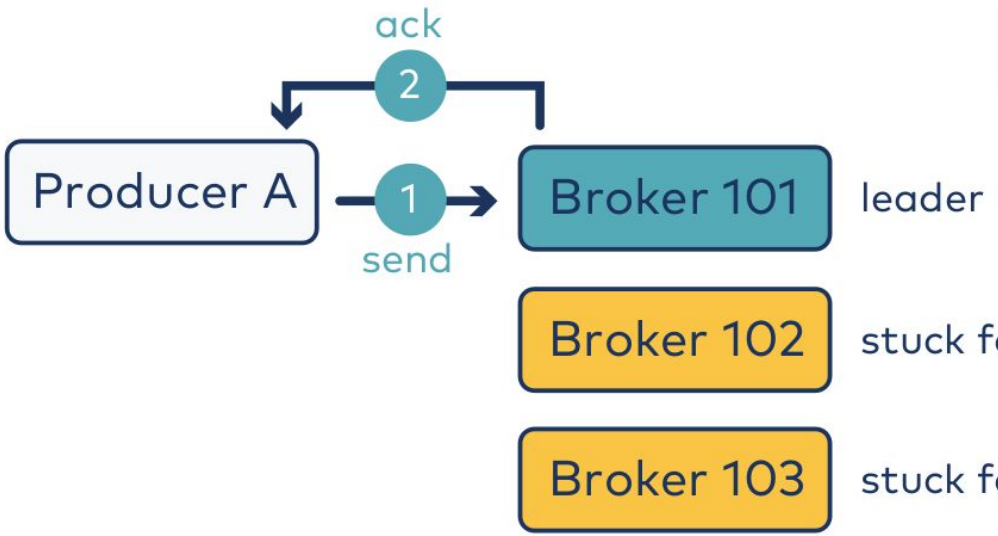
Le **ACK** permet de configurer quand un **Producer** doit considérer une “write request” comme complétée.

- **ACK 0** : le Producer n’attend aucun **ACK** du **Broker**. Il n’y a aucune garantie de delivery. Par ailleurs, comme il n’y a pas de retour du **Broker**, le **Producer** n’a pas connaissance des **Offsets** générés

- **ACK 1 (Leader)** : Le **Producer** attend le **ACK** du **Leader** mais pas celui des replicas. Si le **Leader** tombe juste après avoir reçu une “write request”, il est possible de perdre des **Messages** si ceux-ci n’avaient pas encore été répliqués

- **ACK ALL** : Le Leader attend d’avoir eu un **ACK** de tous les **ISR** avant de renvoyer le **ACK** au **Producer**. Tant qu’il y a au moins 2 **ISR**, on est garanti de ne pas perdre de **Messages**. Il est conseillé de configurer la propriété min.in.sync.replicas à une valeur supérieure à 1. Elle permet de déterminer combien d’**ISR** doivent **ACK** une write request

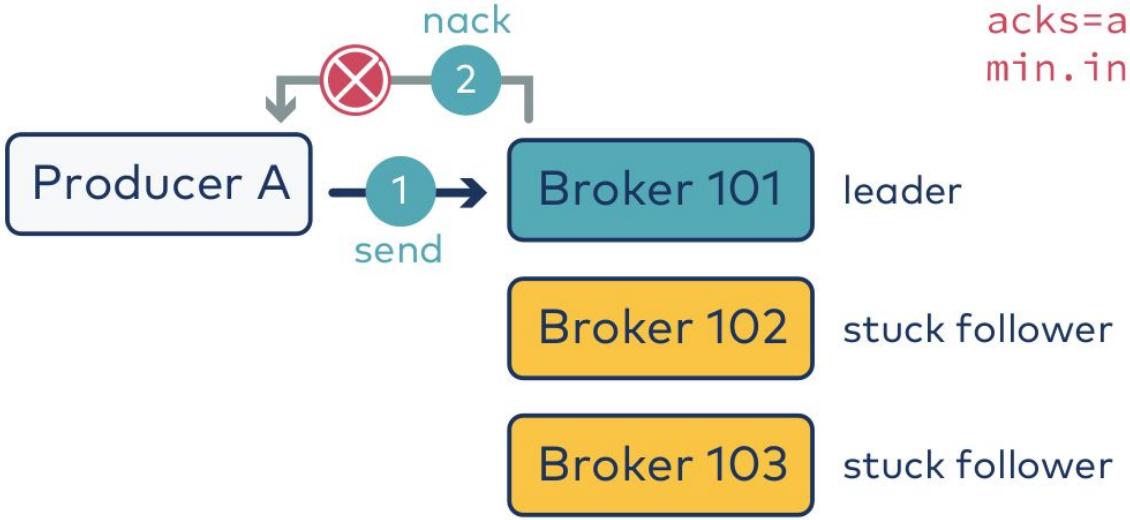
Stuck Followers, min ISR = 1




```
acks=all
min.in.sync.replicas=1
        (default value)
```

```
ISR = [101]
```

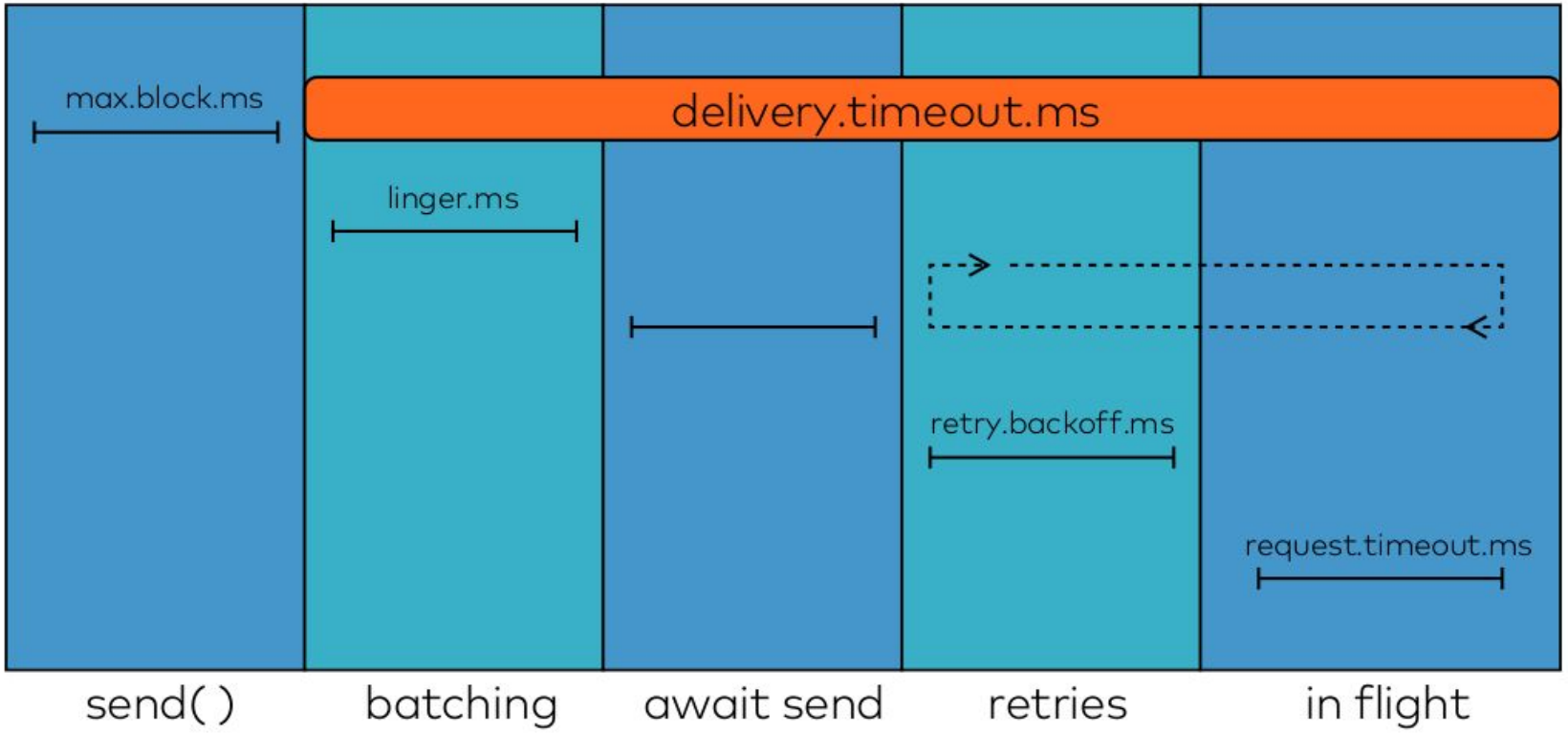
Stuck Followers, min ISR = 2




```
acks=all
min.in.sync.replicas=2
```

```
ISR = [101]
```

Delivery Timeout



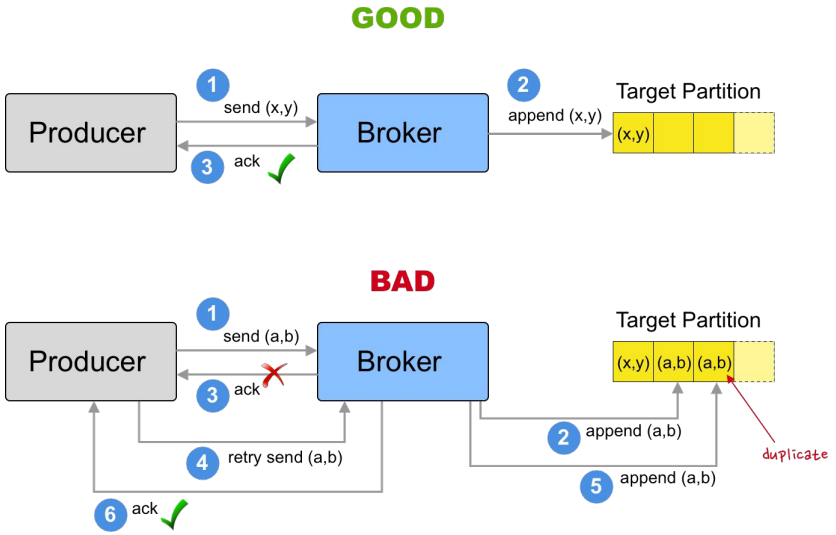
Idempotent Producer

Pour arriver à faire de l'**Exactly Once Semantics** (**EOS**), la première étape est d'activer enable.idempotence = true sur les **Producers**. La propriété max.in.flight.requests.per.connection sera automatiquement à 1.

Depuis CP 7.0 / AK 3.0, l'idempotence est activée par défaut.

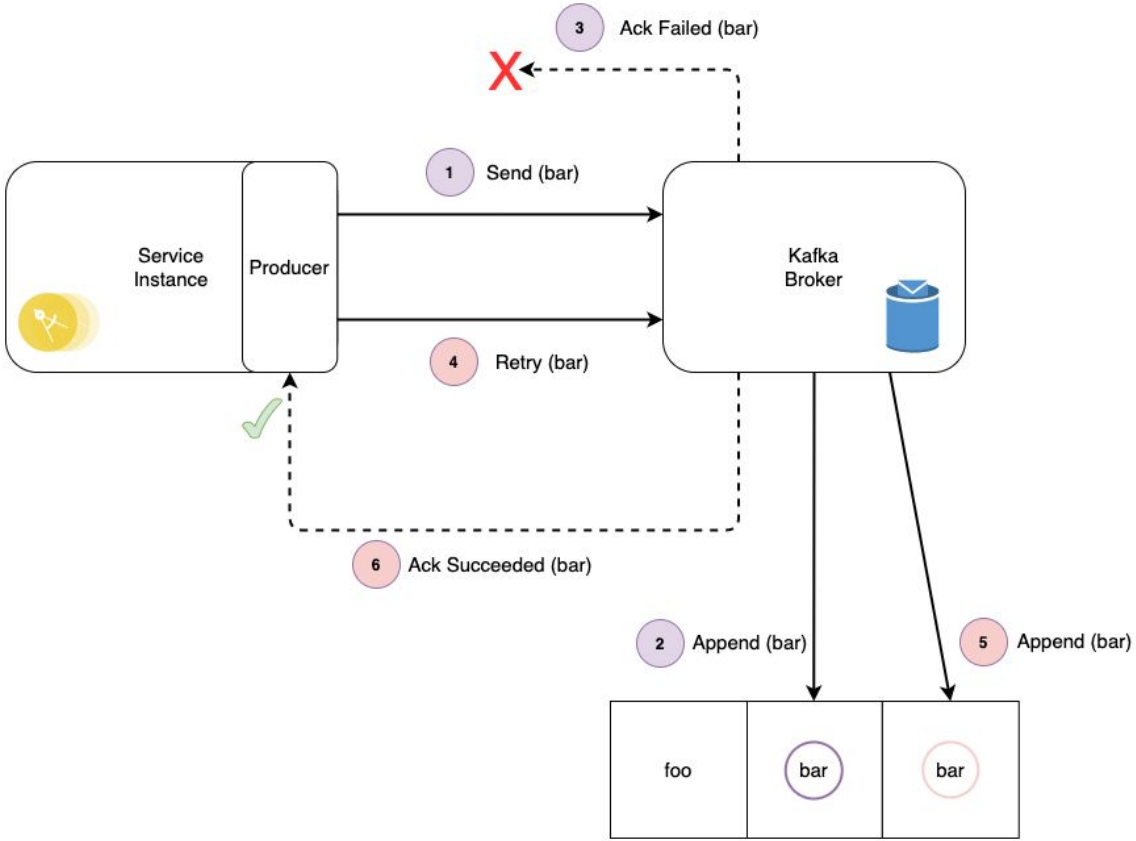
En collaboration avec le **Broker**, un **Producer** idempotent garanti :

- Tous les **Messages** d'une **Partition** sont écrit dans l'ordre dans le Commit Log du **Broker**
- Chaque message est écrit une seule fois (pas de duplica)
- En complément du ack=all on s'assure qu'aucun message n'est perdu



Idempotent Producer

Not Idempotent Producer

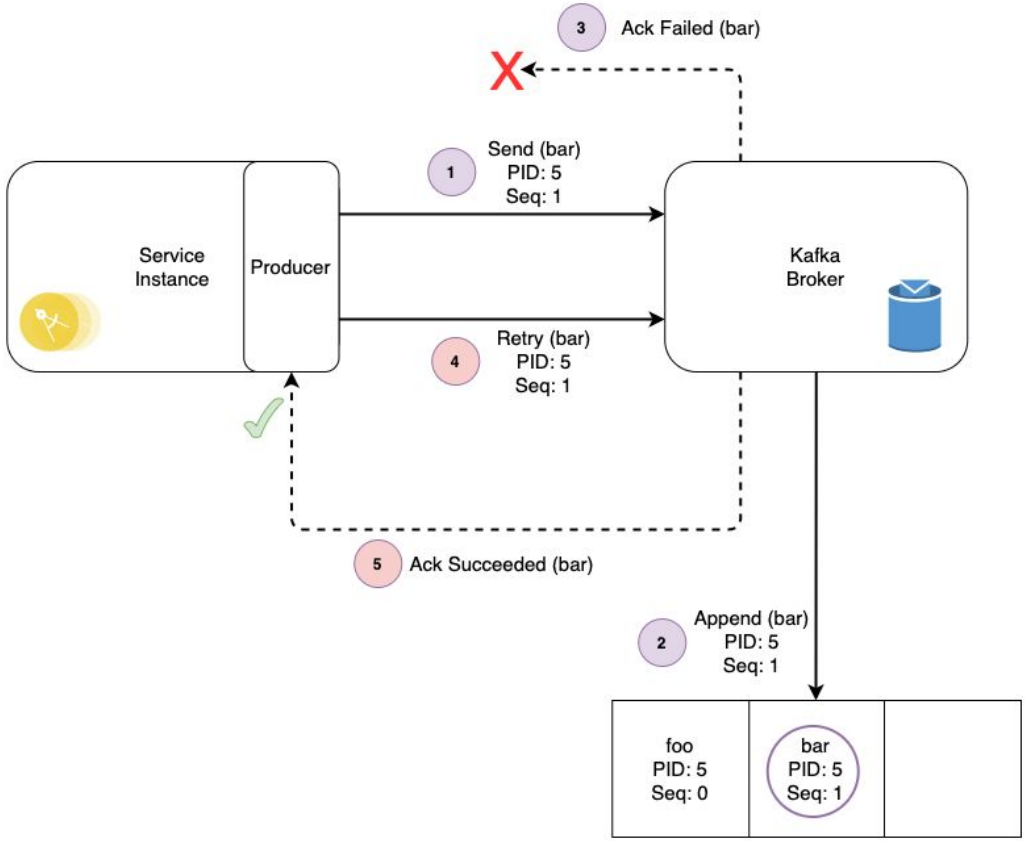


Idempotent Producer

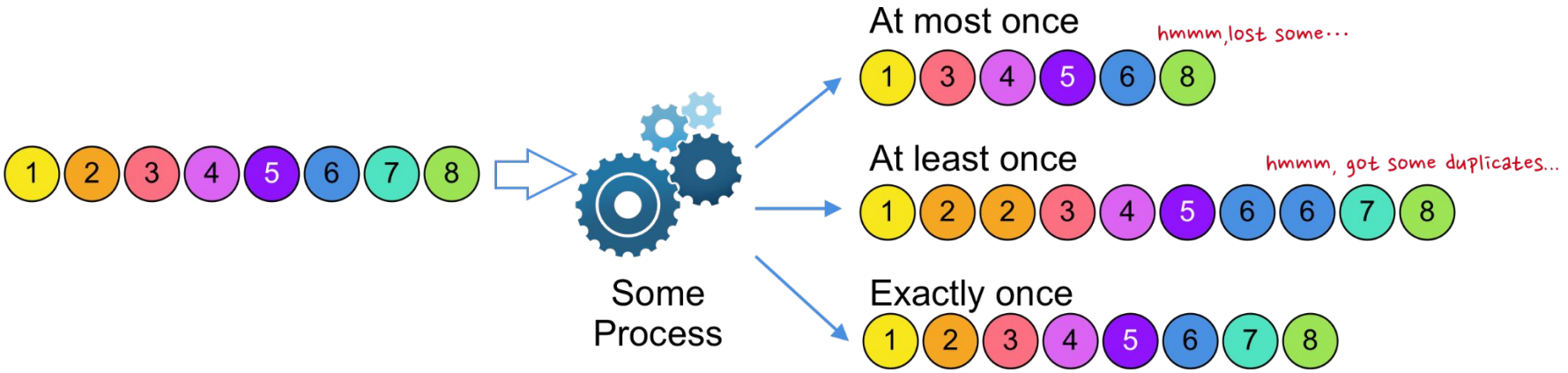
Idempotent Producer

Lorsque le **Producer** est configuré comme idempotent, il se voit assigné un **PID** unique et chaque **Message** se voit assigné un numéro de séquence incrémental.

Le **Broker** traque les couples de **PID** + sequence number afin d'identifier les "write requests" qui sont des duplicas.

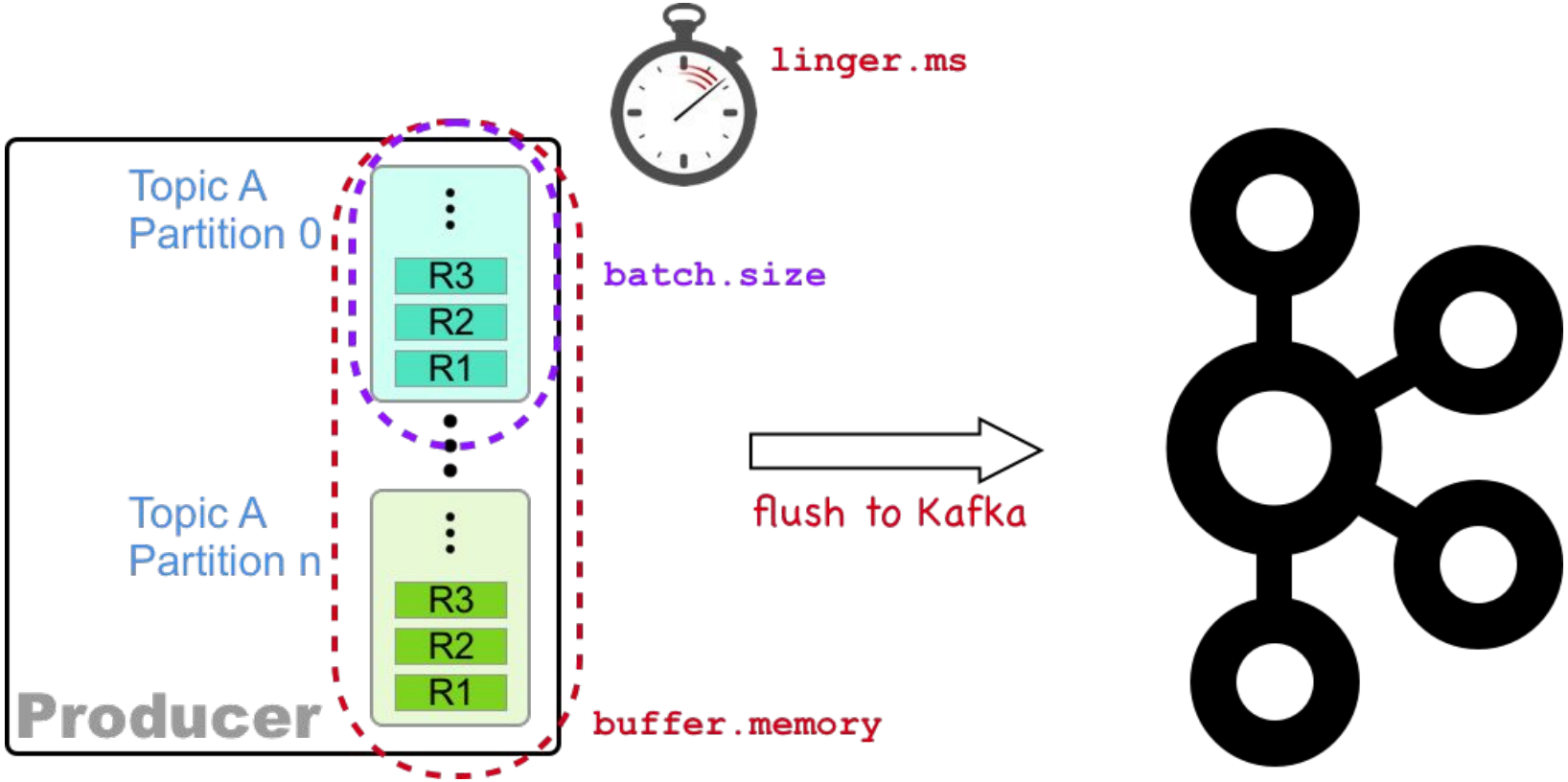


Delivery Guarantees



Batching & Retries

Batching



Batching

- linger.ms indique au **Producer** d'attendre avant d'envoyer le **Batch**. Default: 0
- batch.size indique la taille des **Batchs** (en byte). Default 16KB
- buffer.memory indique la taille max du **Buffer** des **Messages** qui doivent être envoyés. Default : 32MB

Pour un débit élevé, il faut augmenter batch.size et linger.ms

Pour un latence faible, il faut diminuer batch.size et linger.ms

Si des **Topics** ont beaucoup de **Partitions**, que les **Brokers** sont lents, ou que des **Messages** sont volumineux, il faut augmenter buffer.memory

Retries

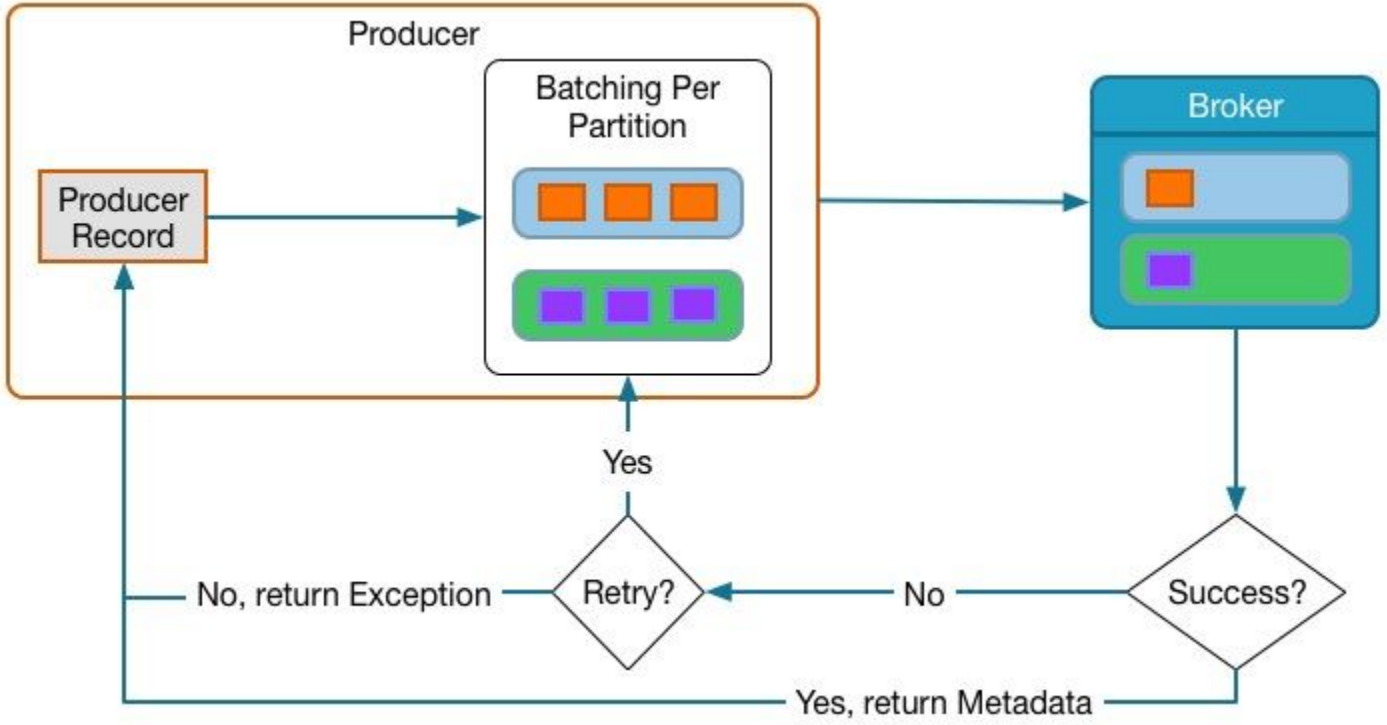
Comme vu précédemment, il est possible de configurer des retries au niveau des **Producers** :

- retries défini combien de voir un **Producer** va essayer de renvoyer une “write request”. Default : MAX_INT . Le Producer s’arrête généralement une fois que delivery.timeout.ms est atteint

- retry.backoff.ms détermine la pause qui est effectuée entre chaque retry

⚠️ Si retries > 0 et max.in.flight.requests.per.connection > 1, alors il se peut que des **Messages** soient stockés dans le désordre

Summary

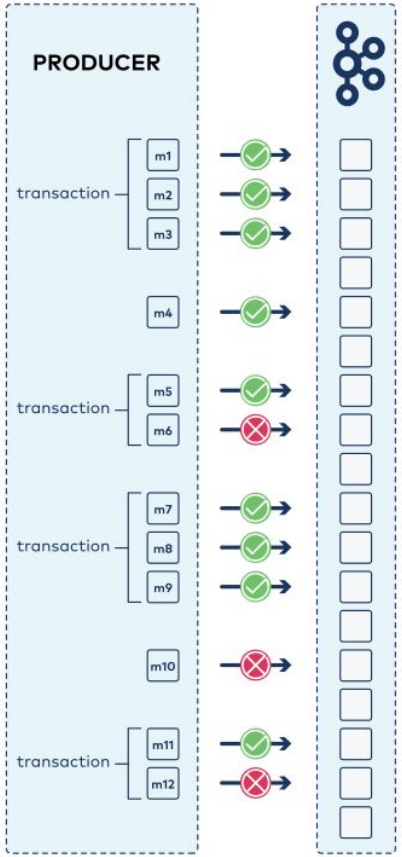

Producer
Batching Per Partition
Producer Record
Broker
Yes
No, return Exception
Retry?
No
Success?
Yes, return Metadata

Callback

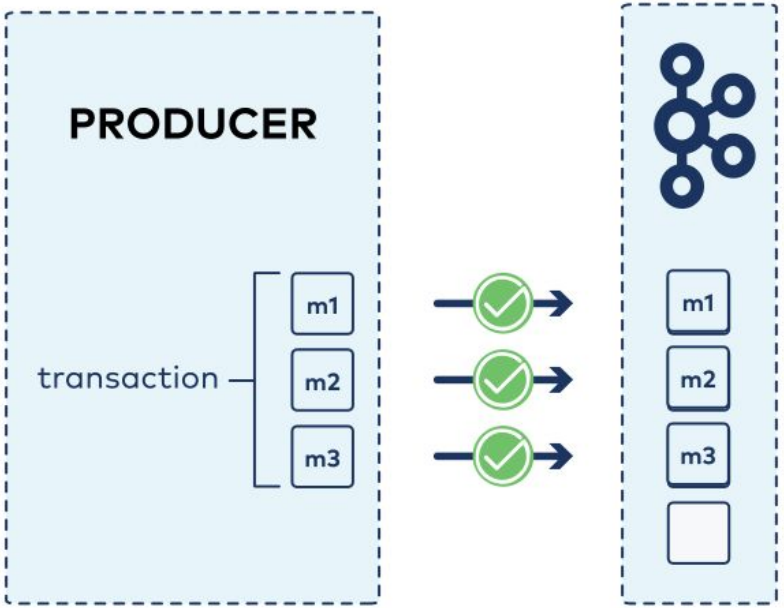
La méthode send() du **Producer** est asynchrone. Toutefois, il est possible d'enregistrer une **Callback** afin de récupérer les **Metadata** ou l'erreur des **Messages** envoyés :

```
producer.send(record, (recordMetadata, e) → {
   if (e ≠ null) {
       e.printStackTrace();
   } else {
       System.out.println("Message String = " + record.value() +
                          ", Offset = " + recordMetadata.offset());
   }
});
```

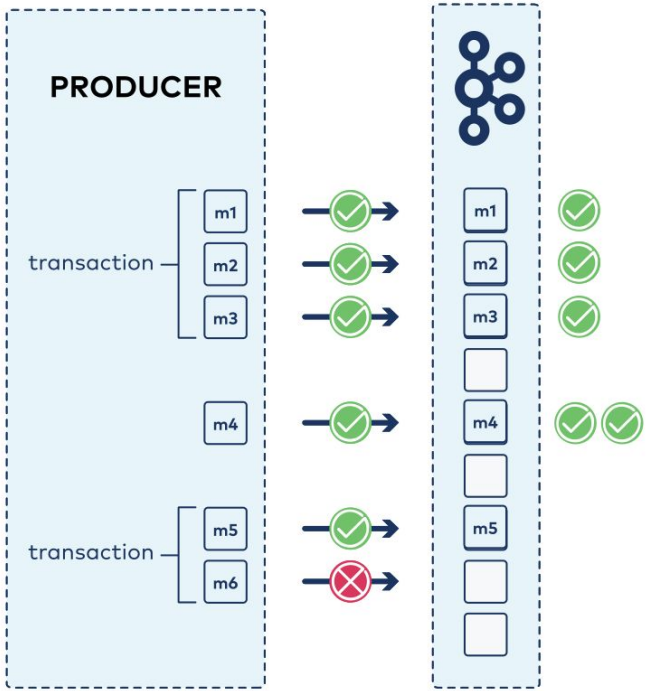
Vue d'ensemble



Transaction success



Transaction failed



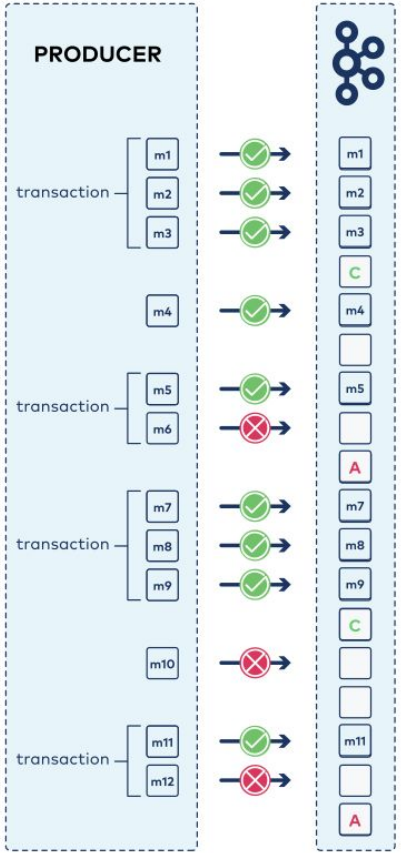
Commit Markers

Pour rappel, les **Messages** sont immutables :

- commitTransaction() génère un **Marker** C dans le **Commit Log**
- abortTransaction() génère un **Marker** A dans le **Commit Log**

Les **Consumers** utilisent les **Markers** pour savoir s'ils doivent lire ou non les **Messages**

ℹ️ Si les **Messages** d'une **Transaction** appartiennent à des **Topics** ou **Partitions** différentes des **Markers** A ou C seront placés dans chacune des **Partitions**



Configurer un Transactional Producer

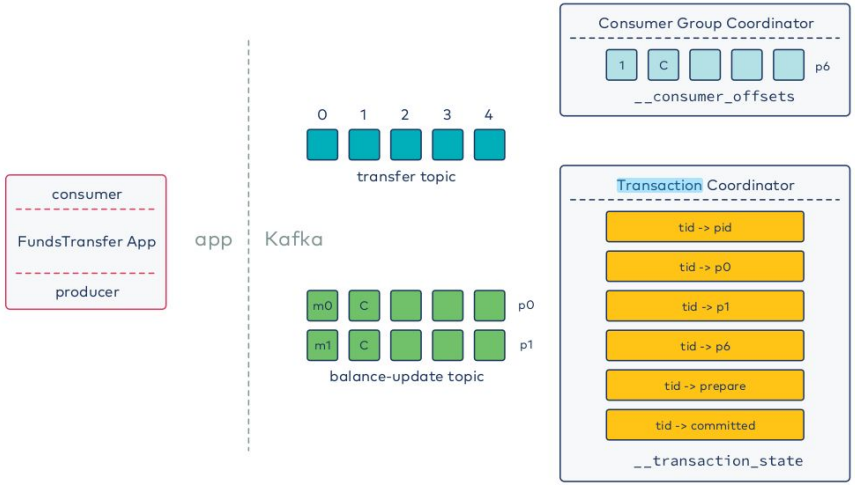
1. L'**EOS** est requise :
   a. enable.idempotence = true
   b. Les **Messages** auront maintenant des **Metadata** supplémentaires :
      i. Producer ID
      ii. Sequence number

2. Démarrer les **Transactions** :
   a. Appeler producer.initTransactions()
   b. Les Messages auront maintenant des **Metadata** supplémentaires :
      i. Transactional ID
      ii. Tous les **Messages** d'une même **Transaction** partagent le même Transactional ID

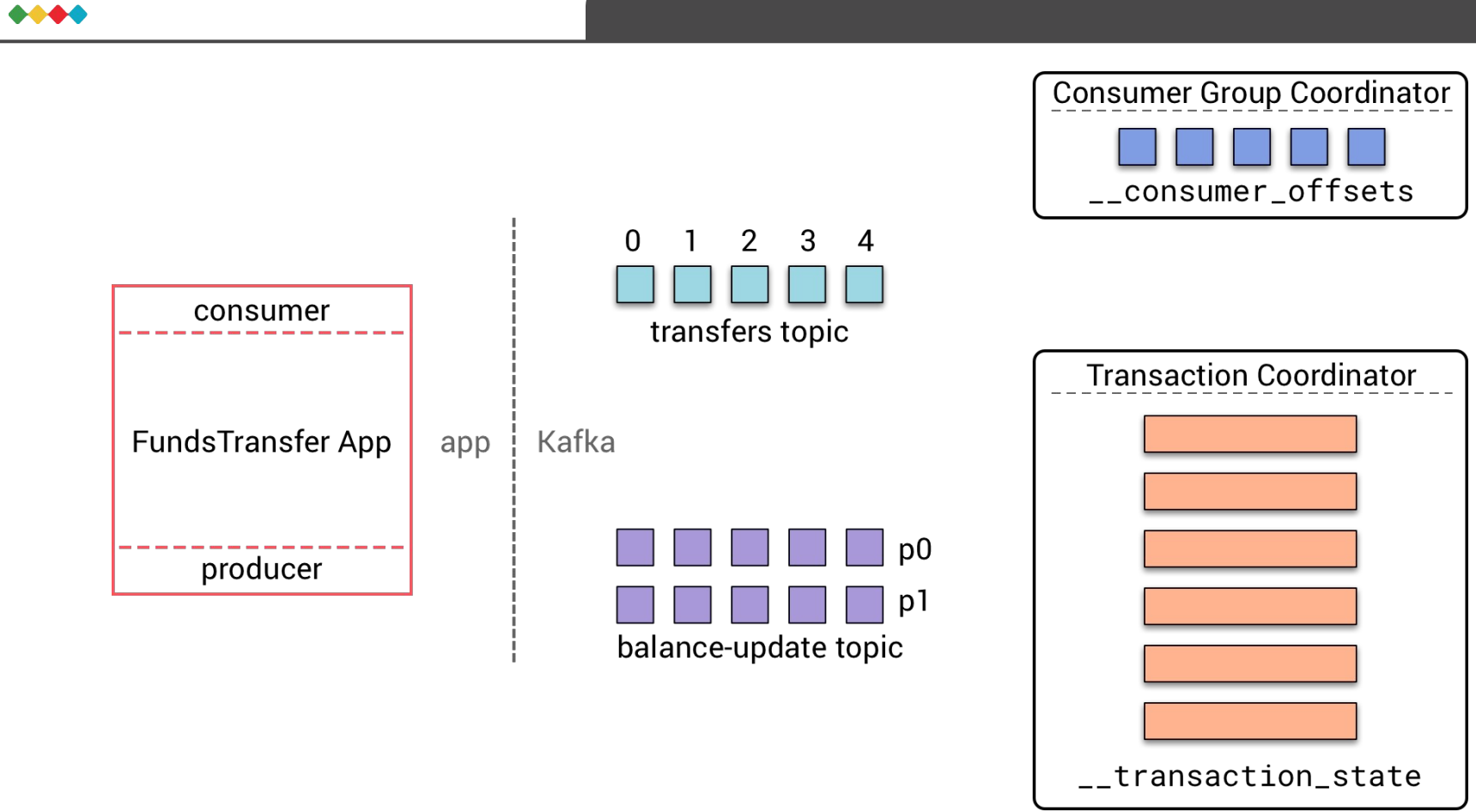
Transaction Coordinator

- Le **Transaction Coordinator** rend les **Transactions** possibles

- Il maintient un **Transaction Log**
  - Il utilise un **Topic** interne : __transaction_state
  - Il suit le statut de toutes les **Transactions**
  - Toutes les **Partitions** affectées par une **Transaction** sont stockés dans ce **Topic**

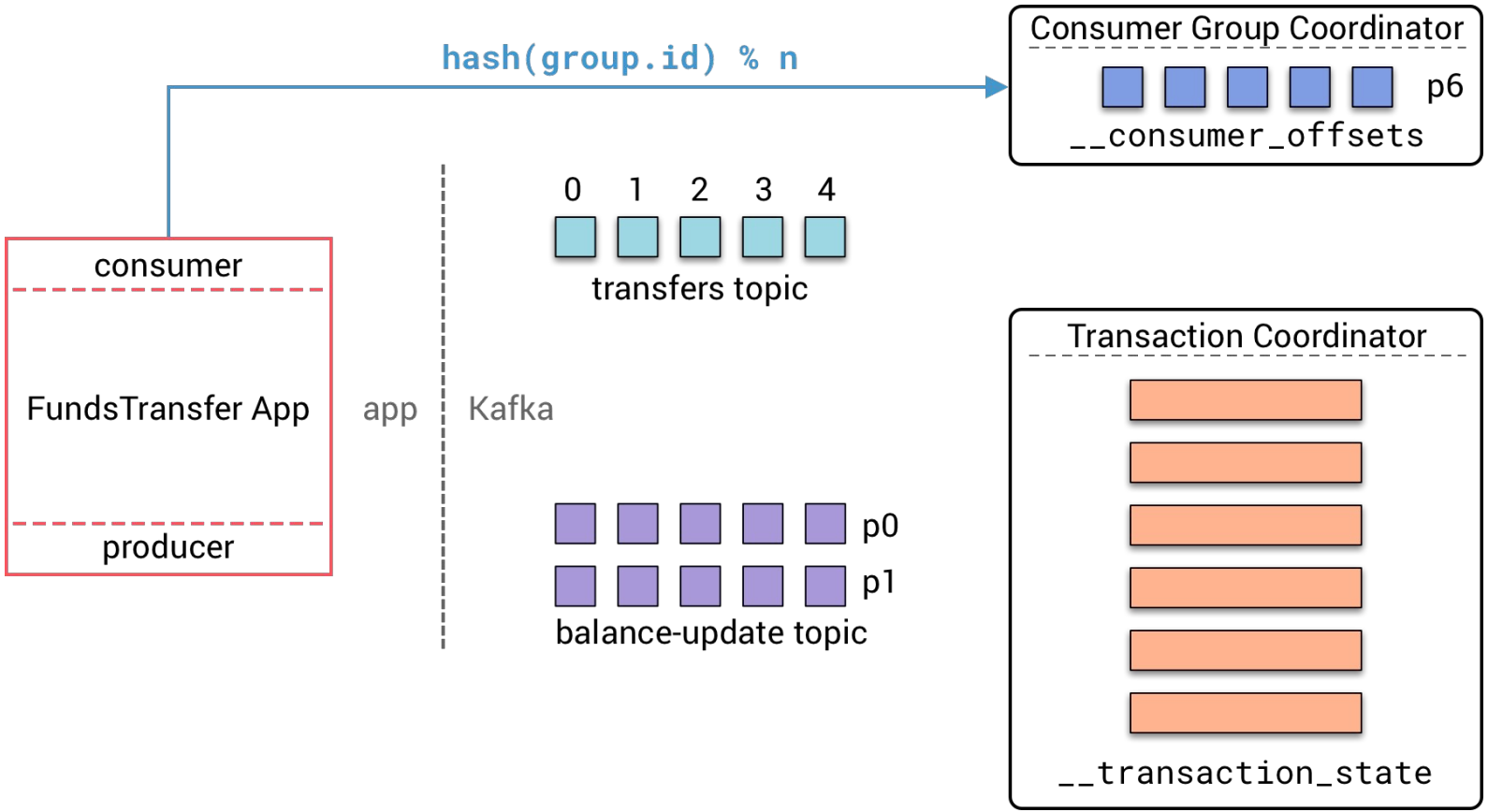


Transactions (1/14)



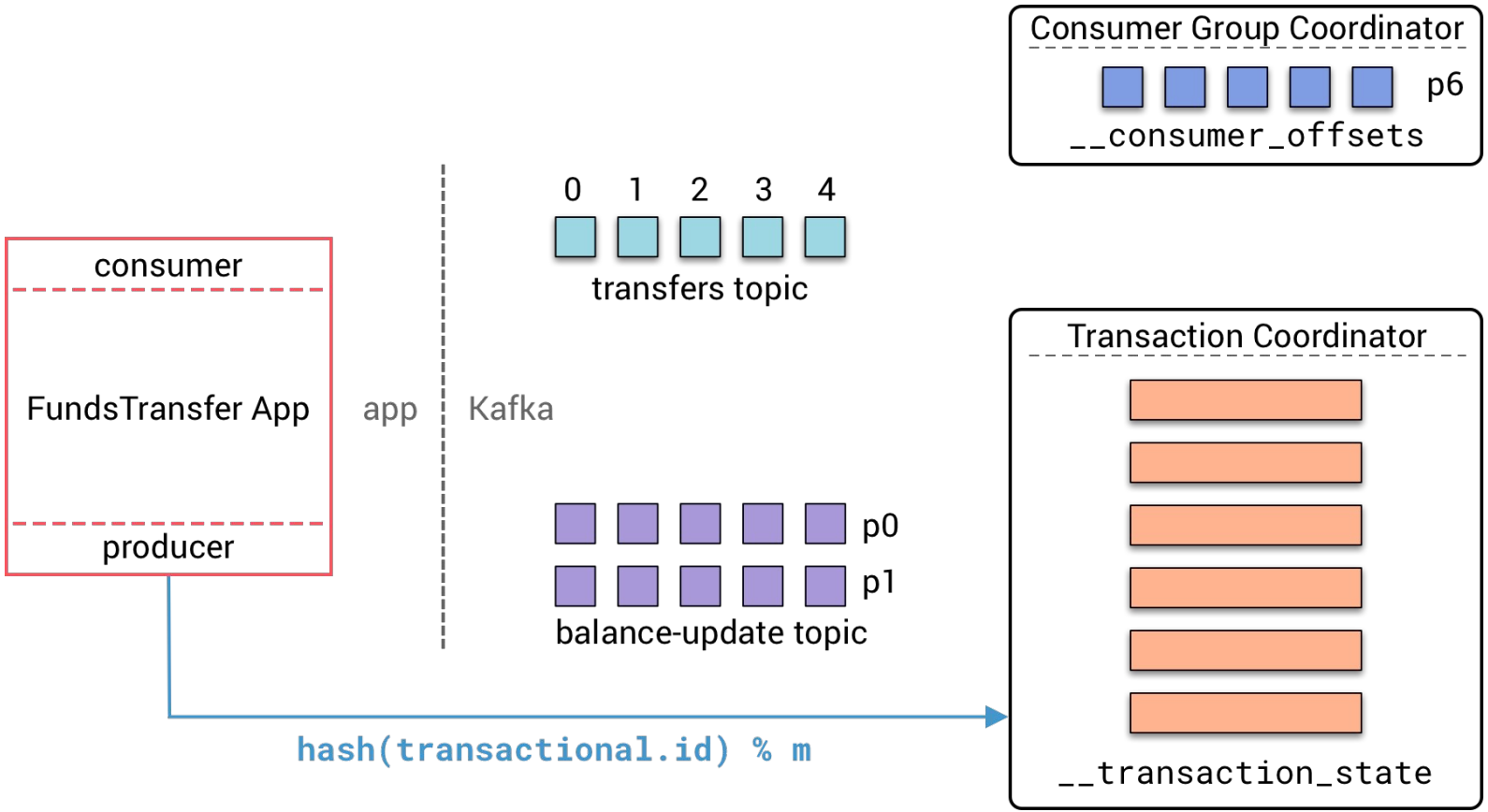
Transactions (2/14)

Initialize Consumer Group



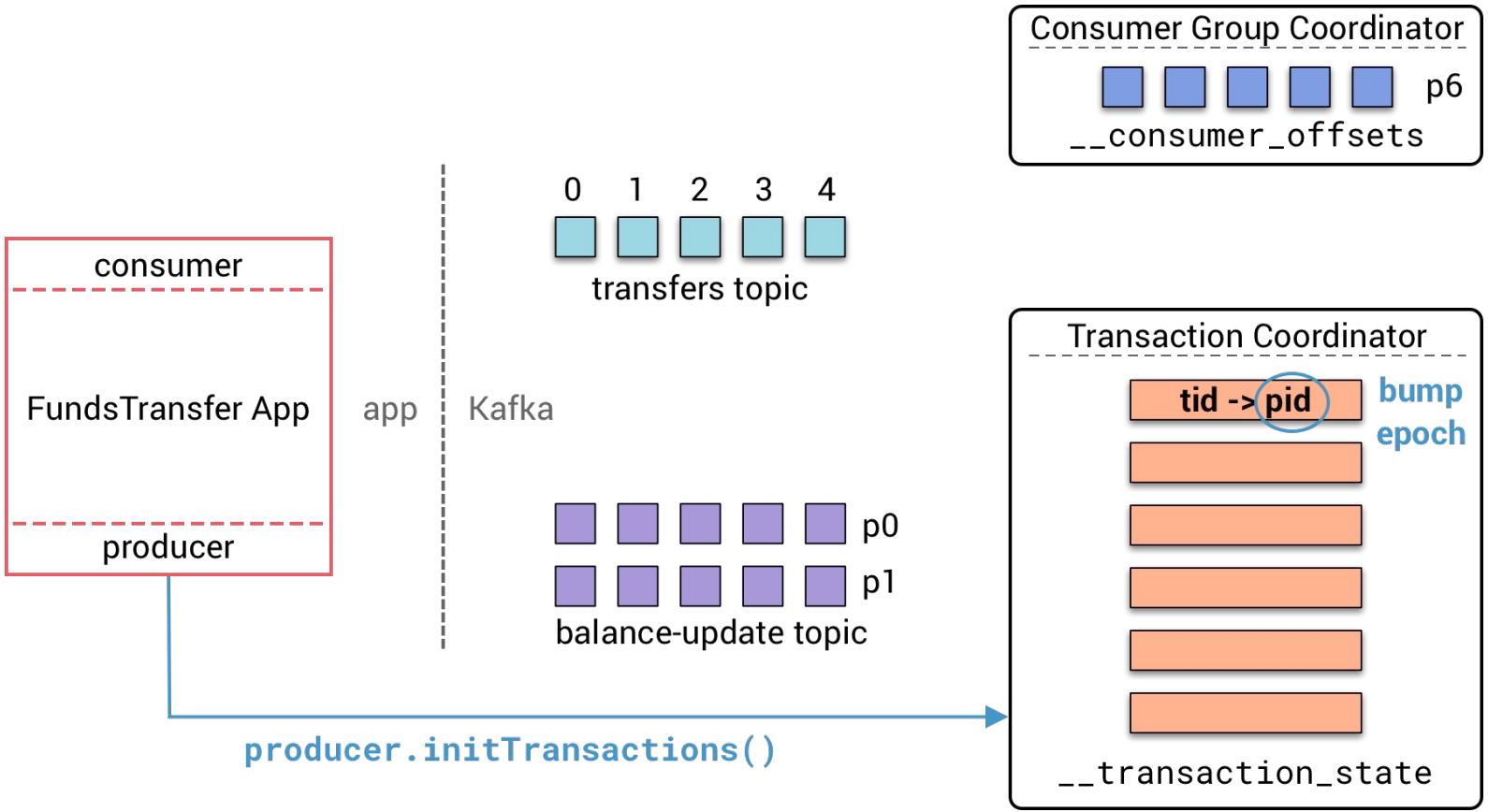
Transactions (3/14)

Transaction Coordinator



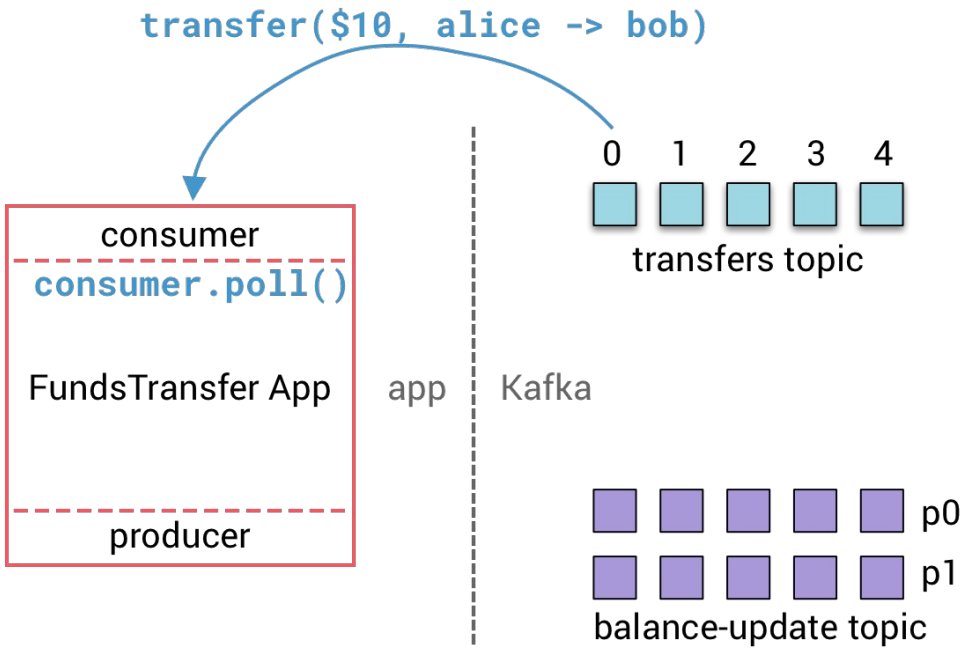
Transactions (4/14)

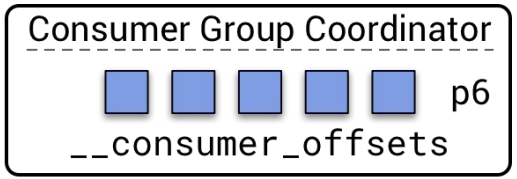
Initialize

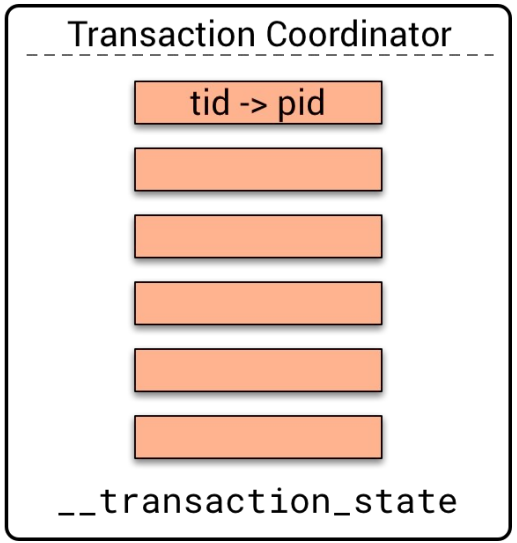


Transactions (5/14)

Consume and Process

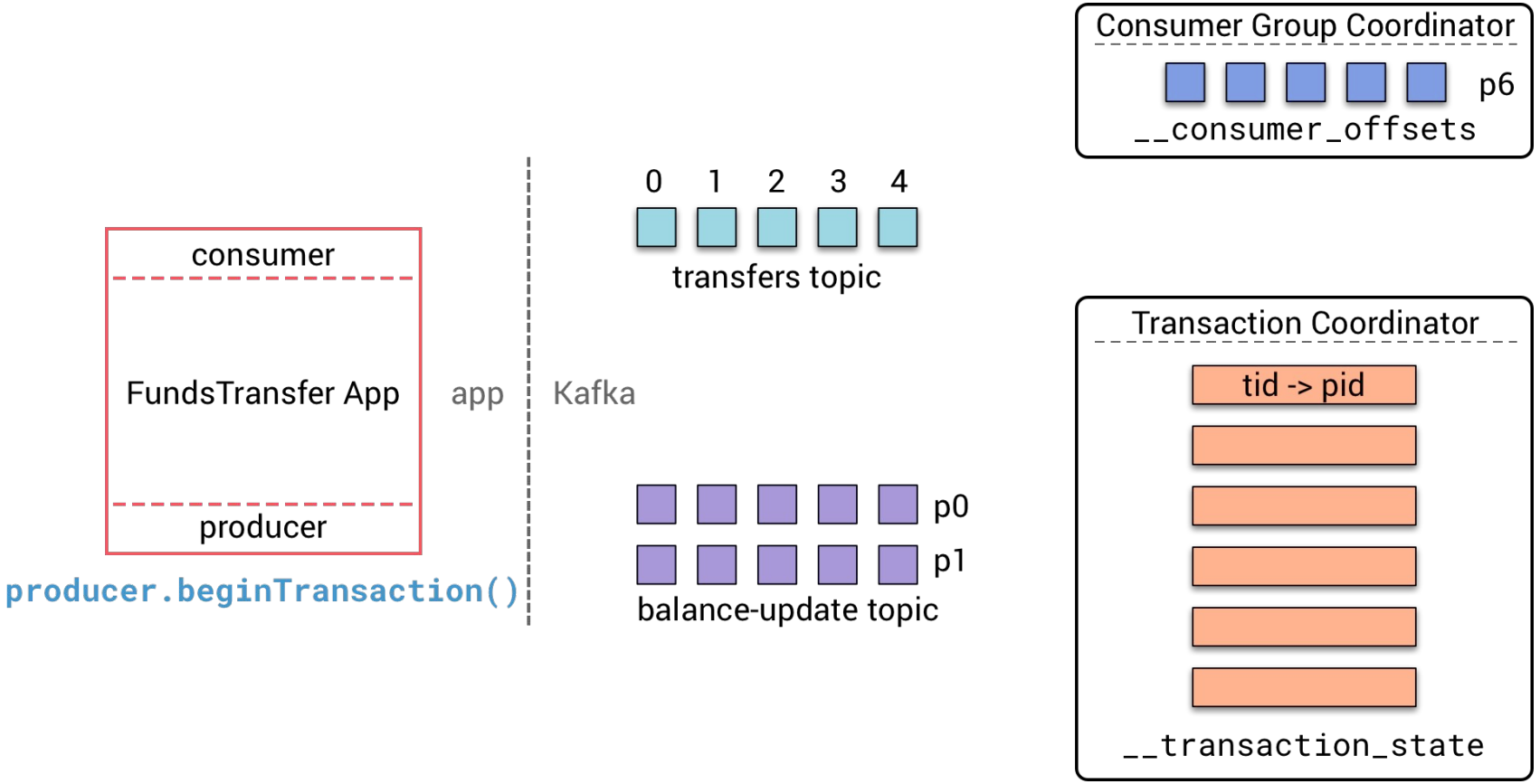






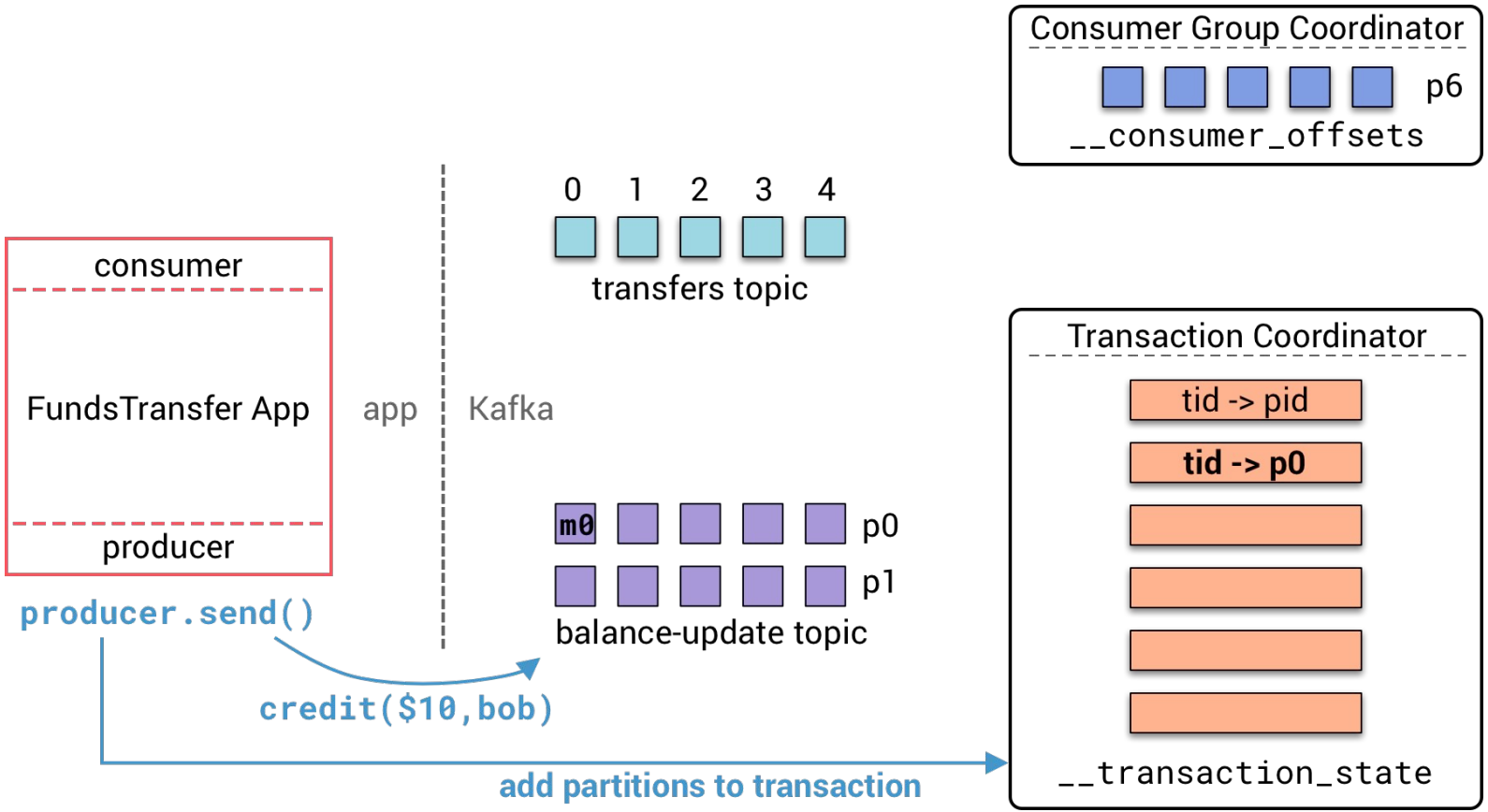
Transactions (6/14)

Begin Transaction



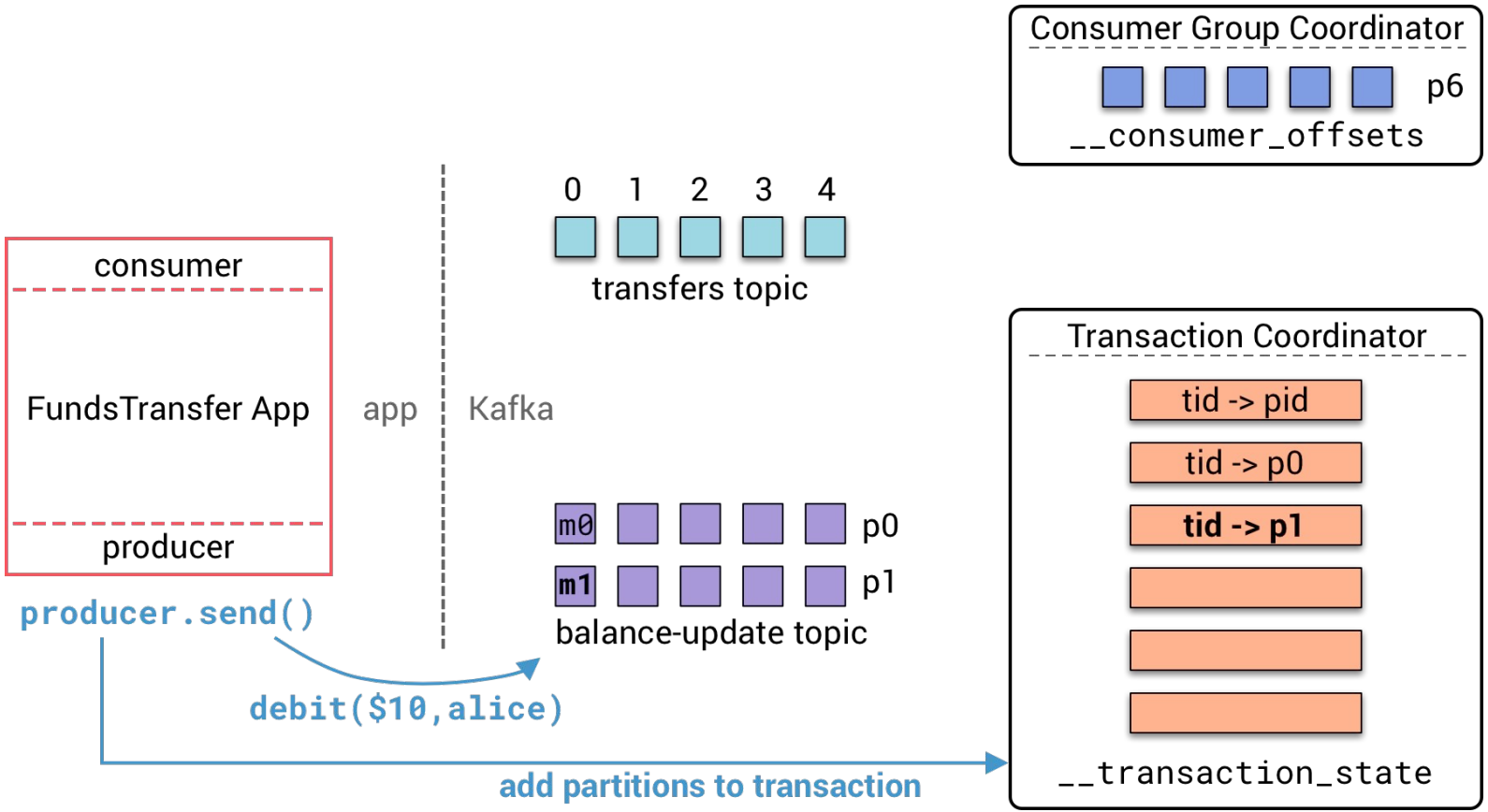
Transactions (7/14)

Send



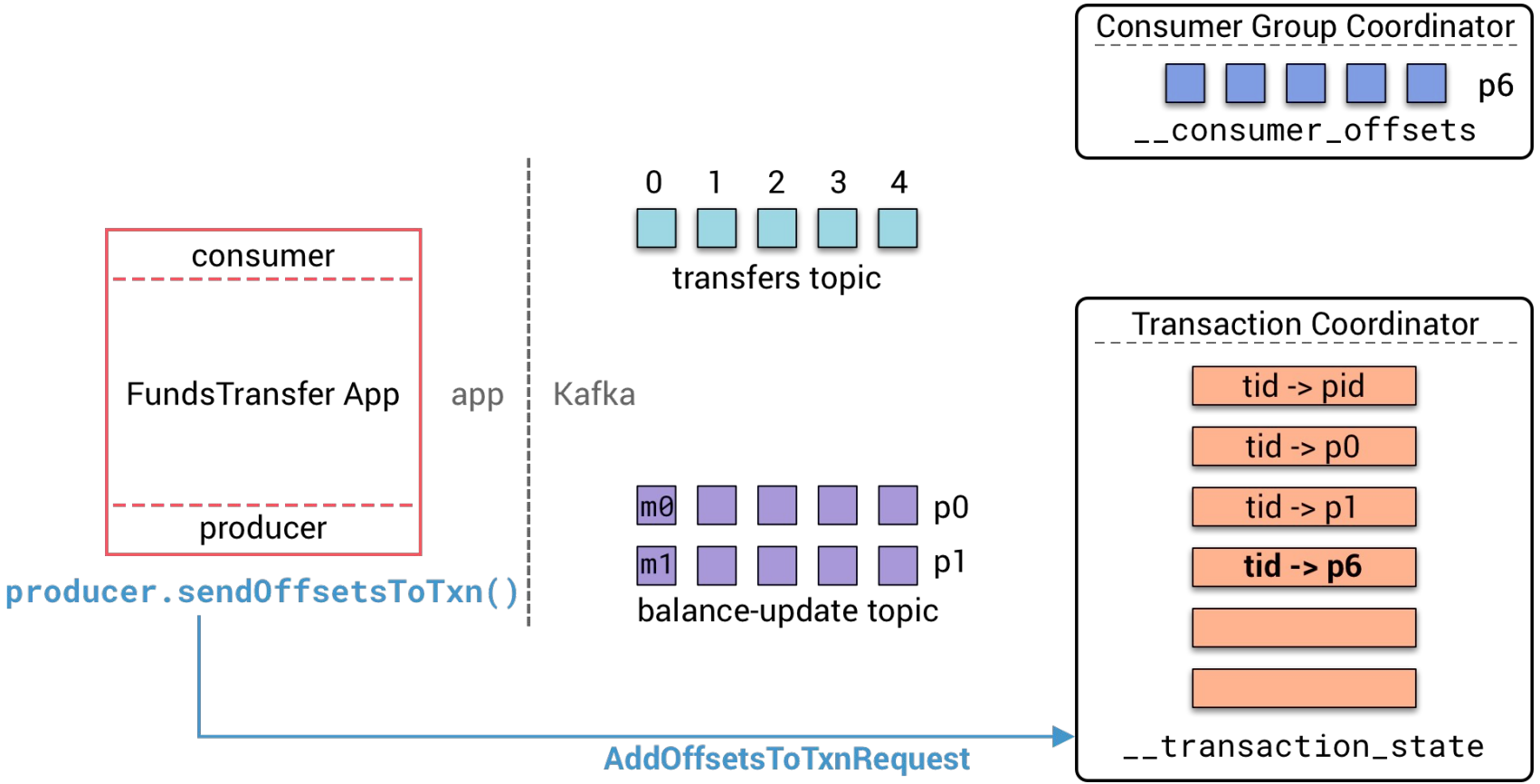
Transactions (8/14)

Send



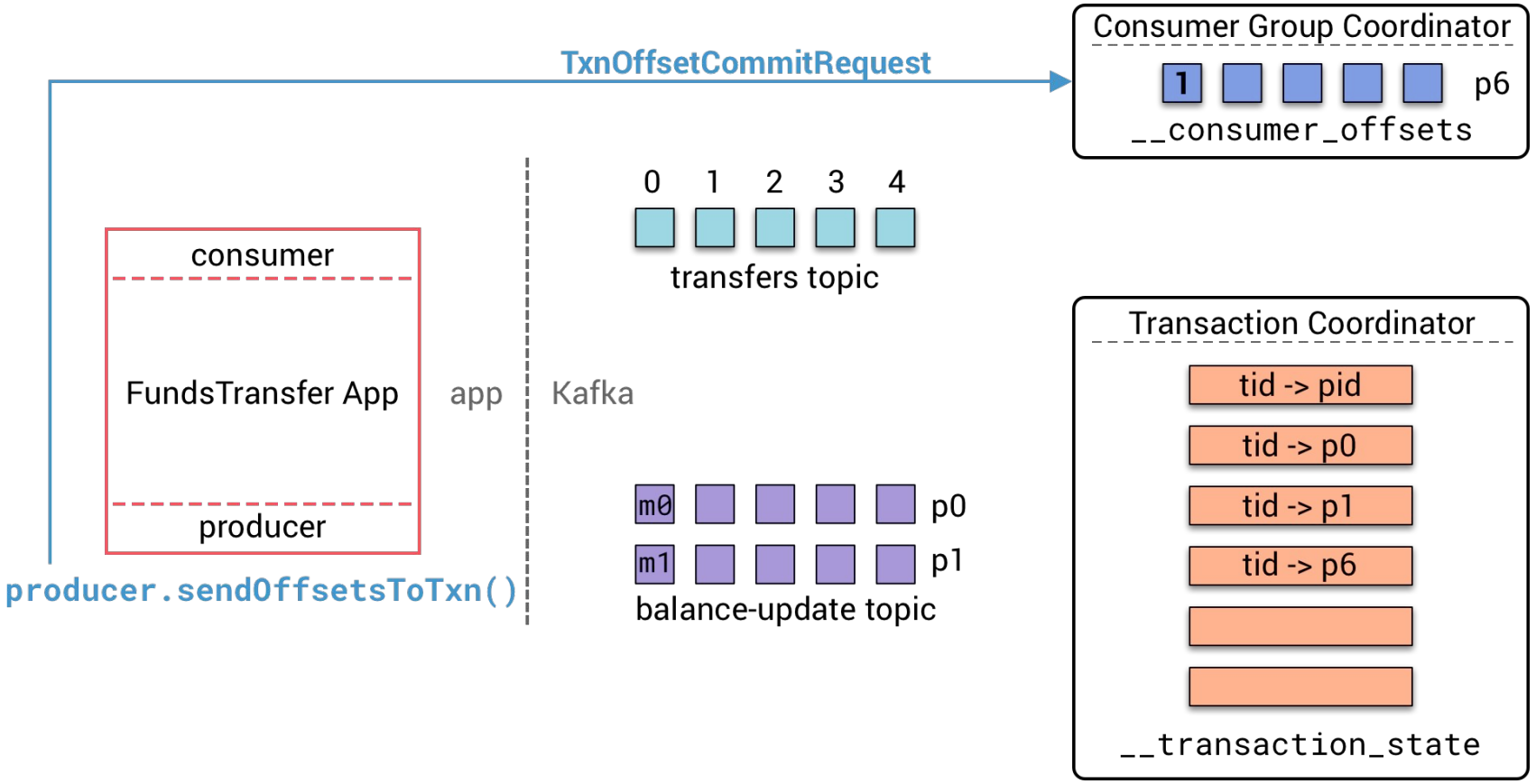
Transactions (9/14)

Track Consumer Offset



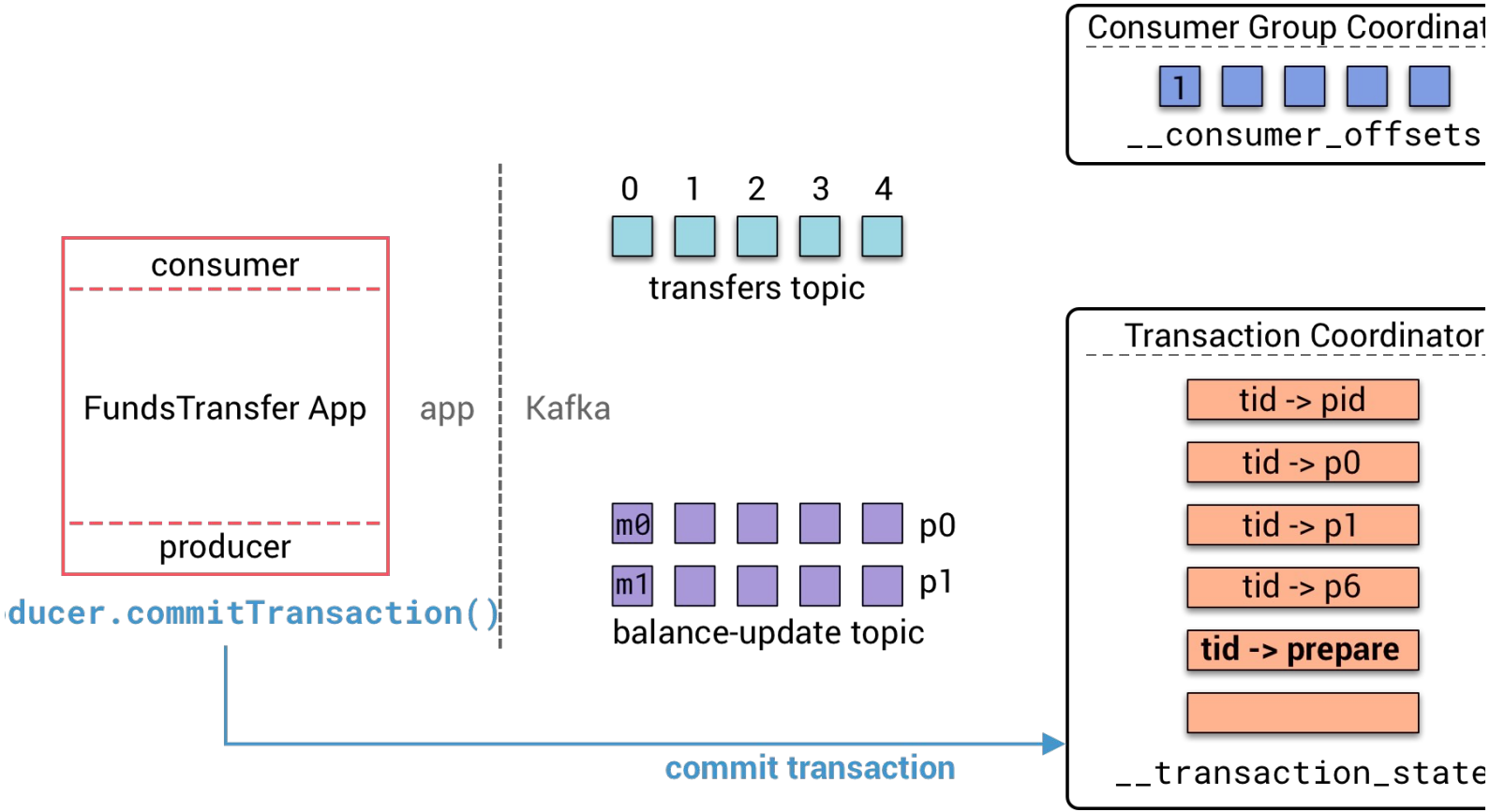
Transactions (10/14)

Commit Consumer Offset



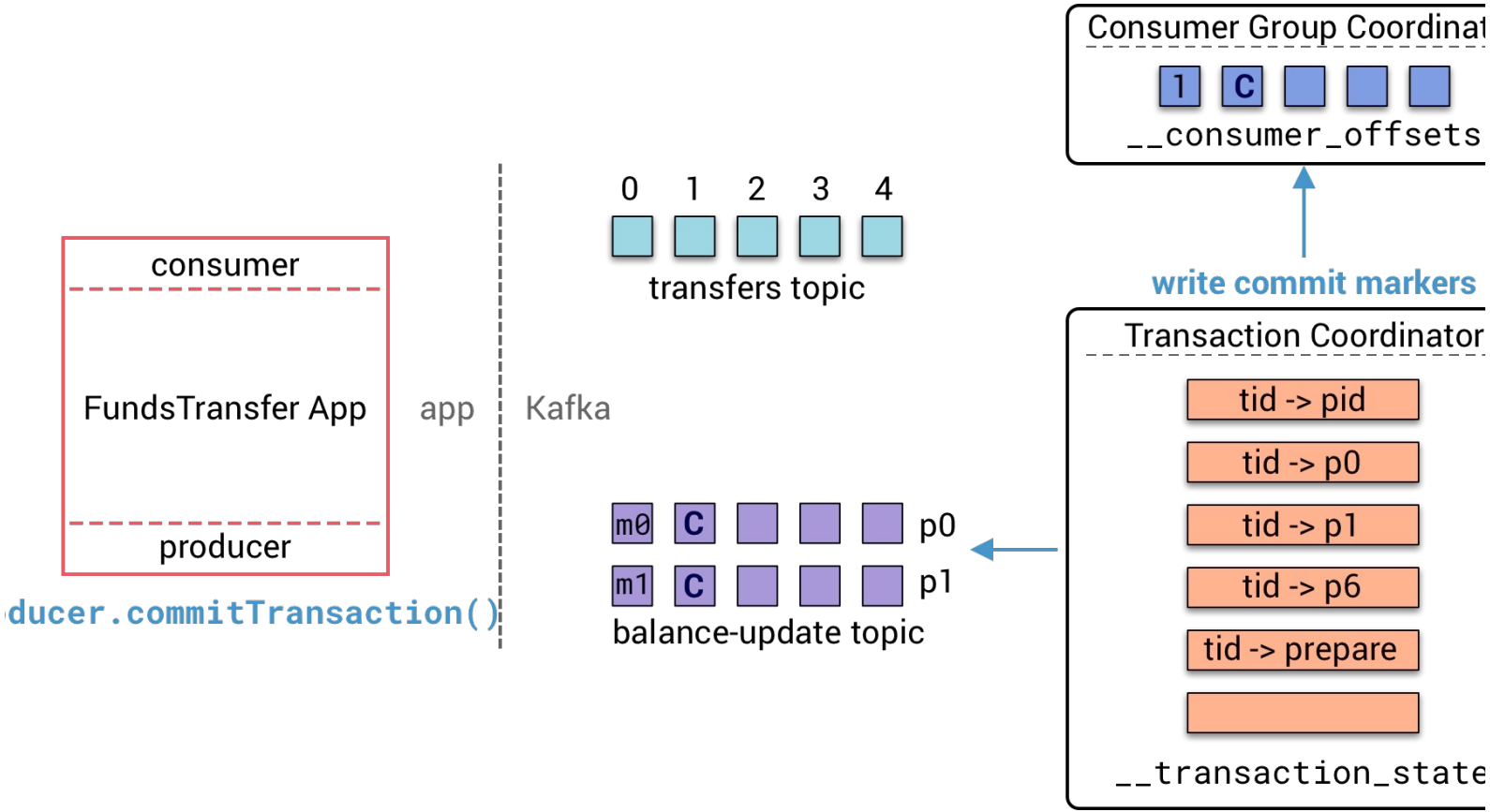
Transactions (11/14)

Prepare Commit



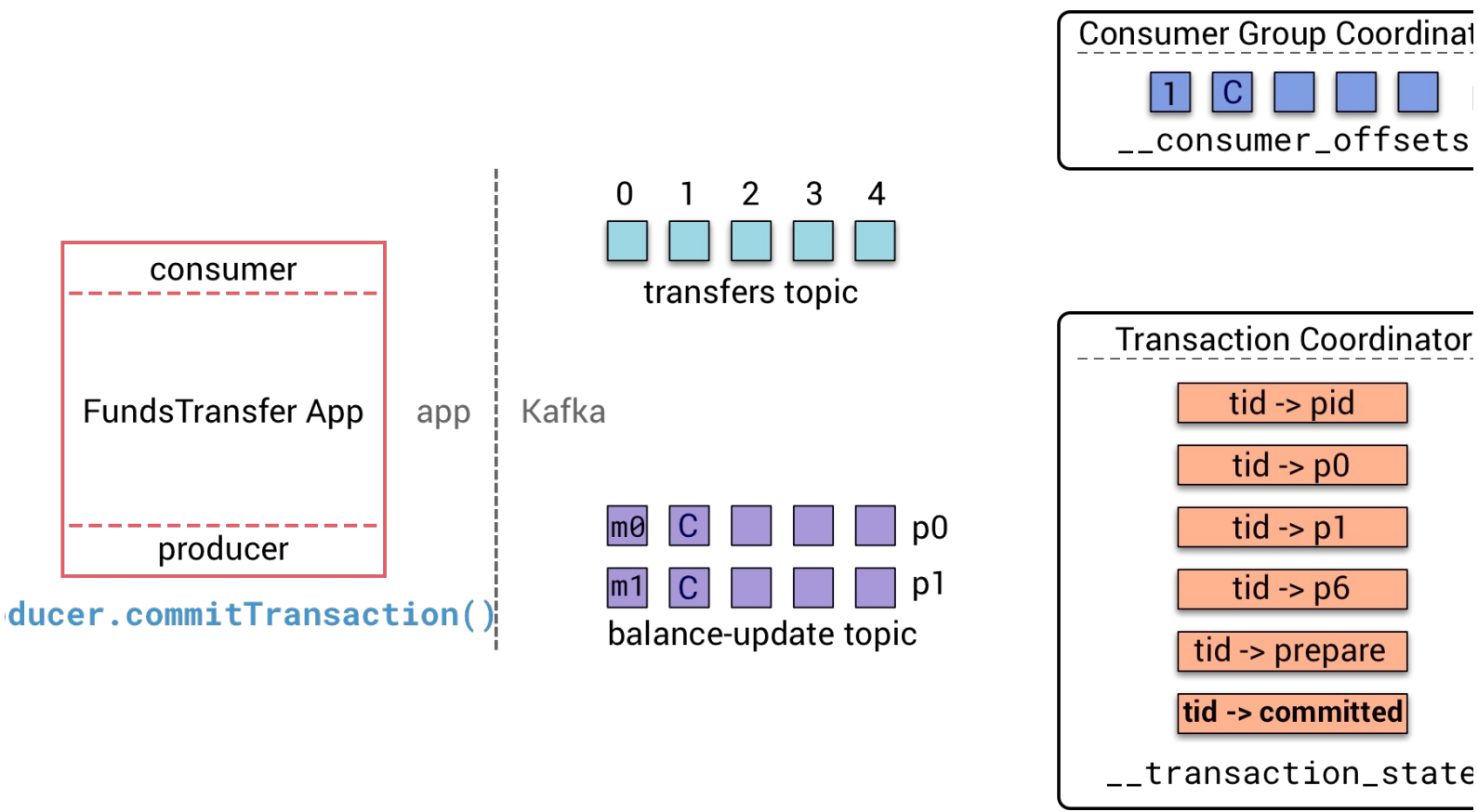
Transactions (12/14)

Write Commit Markers



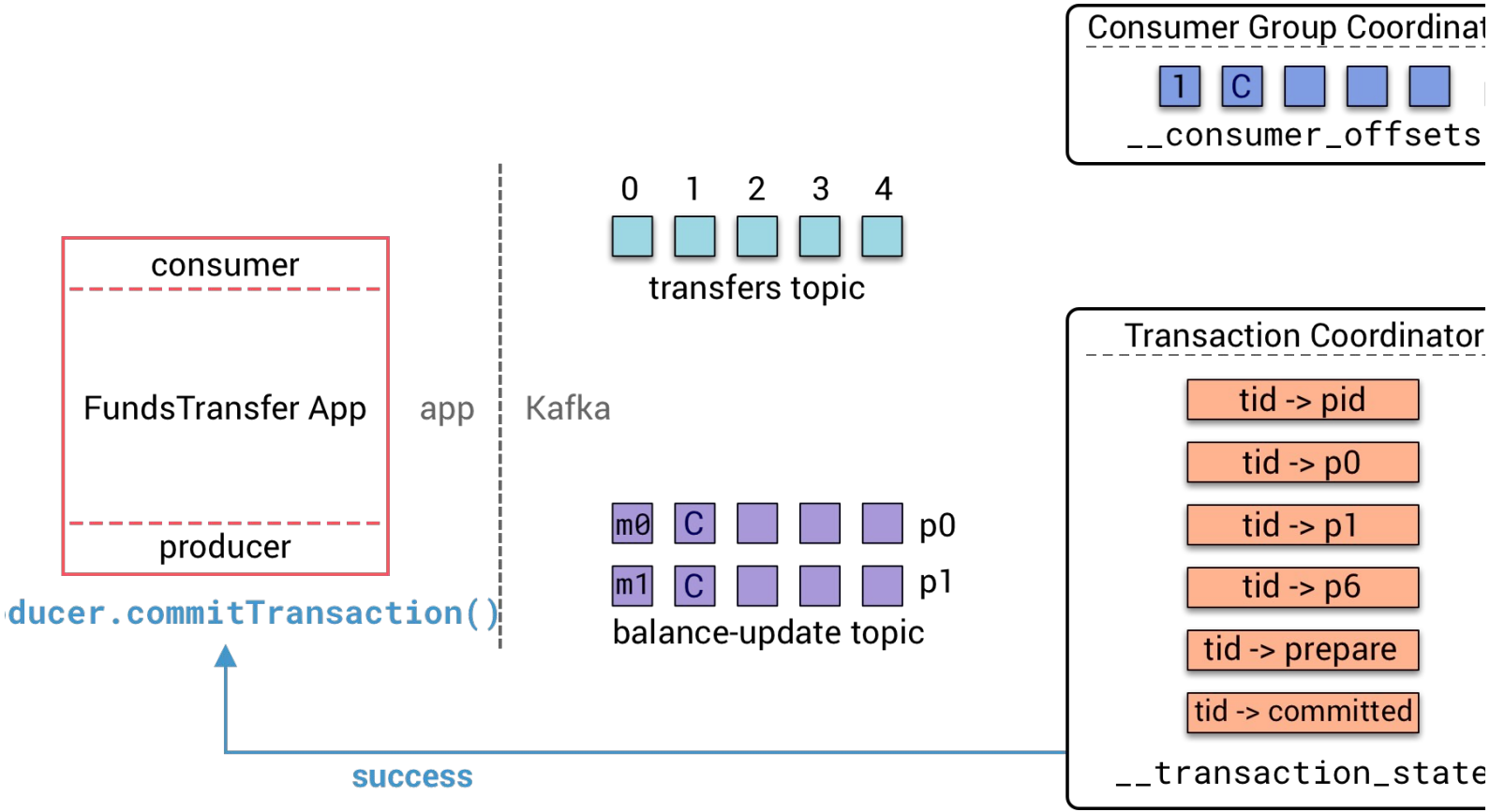
Transactions (13/14)

Commit



Transactions (14/14)

Success

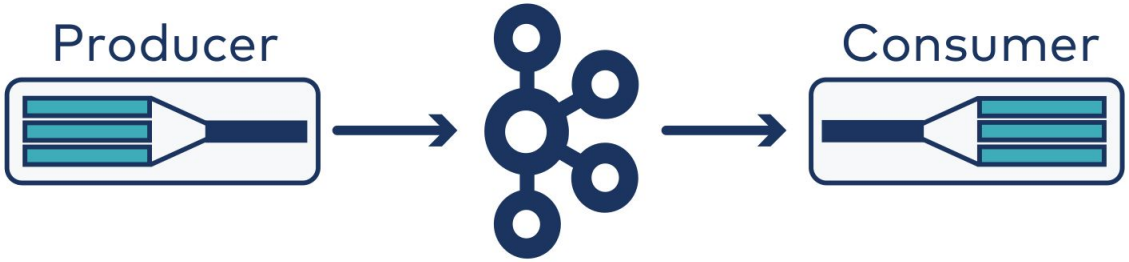


Message Size

- La valeur par défaut de message.max.bytes au niveau des **Brokers** est de 1MB
- La valeur par défaut de max.message.bytes au niveau des **Topics** est de 1MB
- La valeur par défaut de max.request.size au niveau des **Producers** est aussi de 1MB
- Augmenter la taille max des messages peut avoir les conséquences suivantes :
  - Des performances du garbage collector dégradées
  - Moins de mémoire disponible pour les autres tâches importantes du **Broker**
  - Plus de ressources nécessaires pour gérer les requêtes

Message Size

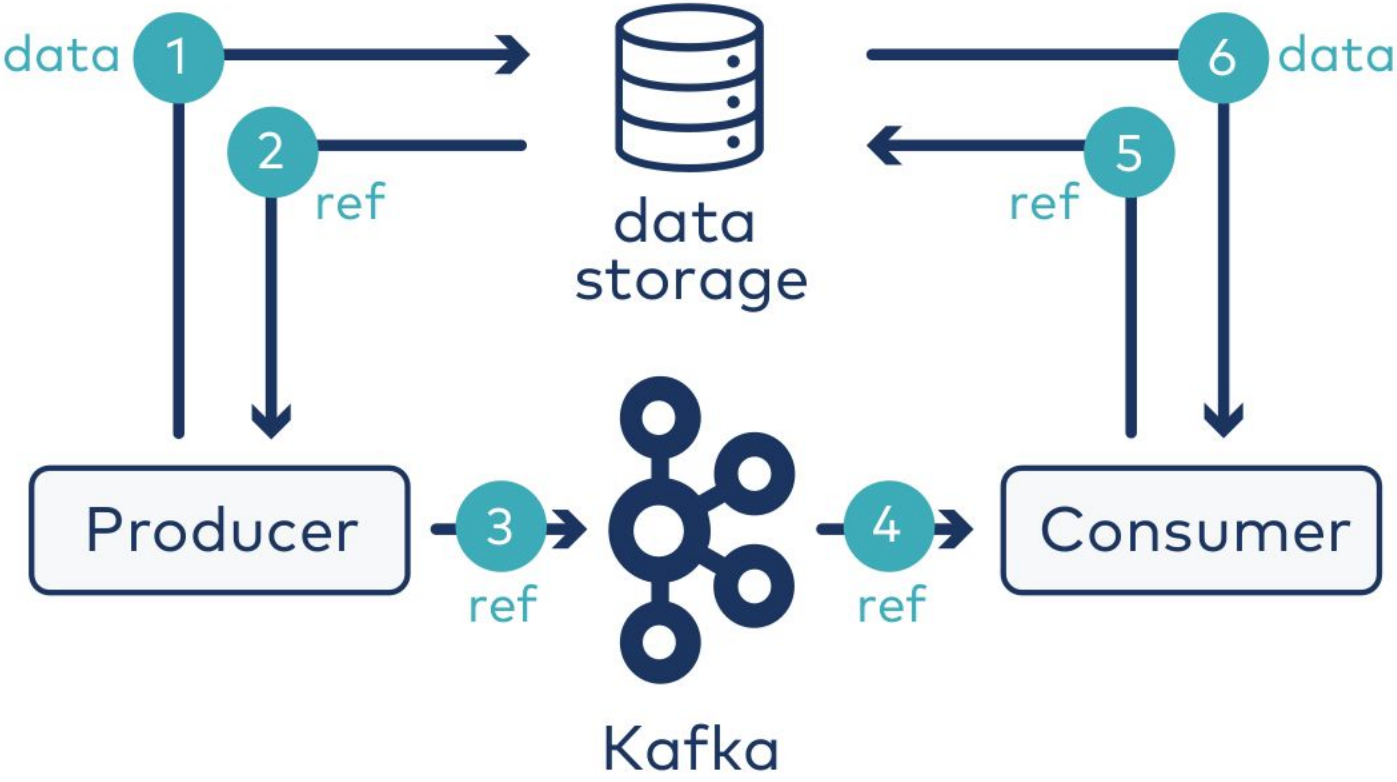
Solution 1 : Compression



- Le **Producer** crée un **Batch** de **Messages** et compresse ensuite le **Batch**
- Le Batch compressé est stocké dans Kafka
- Le Consumer décompresse le **Batch**
- Il est possible d'avoir plusieurs types de compression au sein d'un même **Topic**

Message Size

Solution 2 : External Reference



Exemples de code

Configuration d'un Producer

Sans Helper class

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker-1:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer.class");
props.put("value.serializer", "org.apache.kafka.common.serialization.KafkaAvroSerializer.class");

KafkaProducer<String, MyObject> producer = new KafkaProducer<>(props);
```

Avec Helper class

```
final Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, keySerializer);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, valueSerializer);

KafkaProducer<String, MyObject> producer = new KafkaProducer<>(props);
```

Exemples de code

Envoyer un message

```
String k = "mykey";
String v = "myvalue";
ProducerRecord<String, String> record = new ProducerRecord<String, String>("my_topic", k, v);
producer.send(record);
```

ℹ️ Pour rappel, la méthode send() est asynchrone, mais elle peut prendre une **Callback** en paramètre

ℹ️ Il est possible de flush manuellement les messages du **Producer** en faisant : producer.flush()

Exemples de code

Envoyer des Messages dans une Transaction

```
producer.initTransactions();
try {
   producer.beginTransaction();
   producer.send( ... );
   producer.send( ... );
   producer.send( ... );
   producer.commitTransaction();
} catch (ProducerFencedException pfe) {
   producer.close();
} catch (KafkaException ke) {
   producer.abortTransaction();
}
```

Exemples de code

Arrêter un Producer

Il est possible d'arrêter un Producer

```
// blocks until all previously sent requests complete
producer.close();

// wait until complete or given timeout expires
producer.close(timeout,timeUnit);
```

Exemples de code

Envoyer un Message - Izivia

Le module commons-eventbus-api expose l'interface MessageProducer permettant de publier des messages dans l'Event Bus :

```
producer.publish(
   ChargingSessionCreateEvent(
       chargingSessionId = it.id!!,
       startDate = it.from,
       chargingStationId = it.chargingInfrastructureData?.chargingStationRef?.id,
       evseId = it.chargingInfrastructureData?.evseRef?.id,
       connectorId = it.chargingInfrastructureData?.chargingConnectorRef?.id,
       currentPhase = it.chargingSessionDetails.currentPhase.name
   )
)
```

Produire des messages

Lab

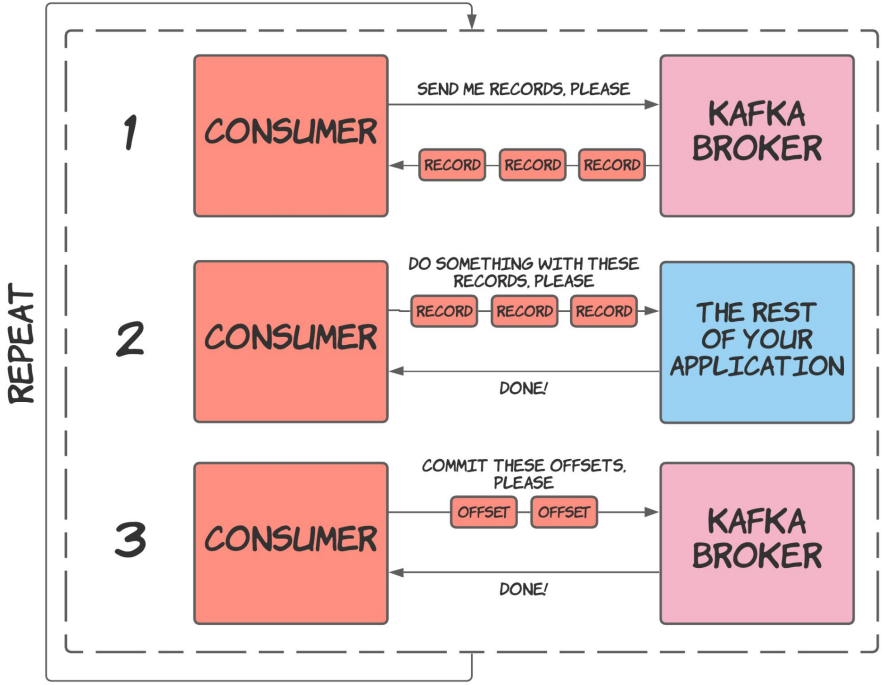
Consommer des messages

Polling

Contrairement à d'autres systèmes de messagerie, les **Brokers** ne "push" pas les **Messages** aux **Consumers**.

Ce sont les **Consumers** qui **Poll** les **Messages** des **Brokers**.

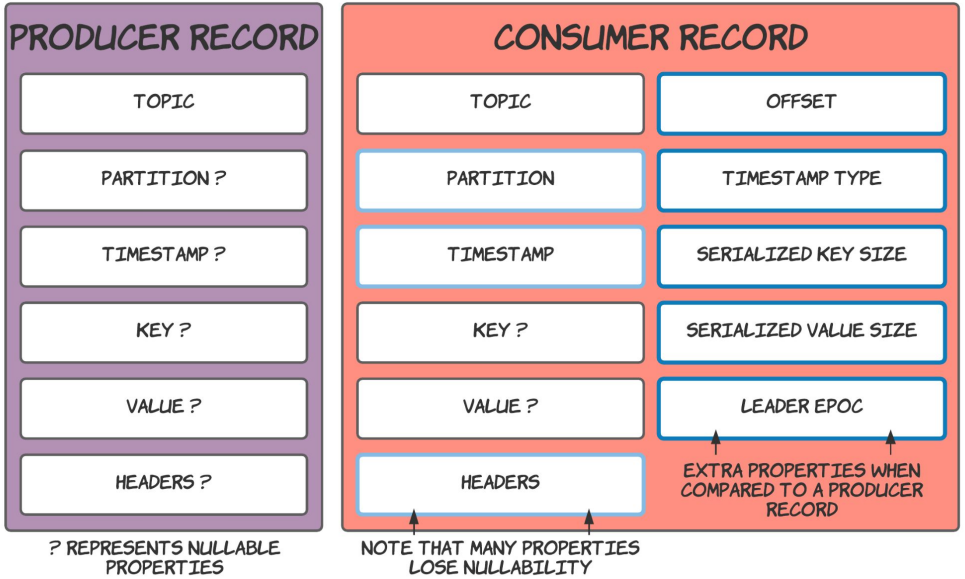
Cela permet aux **Consumers** de process les **Messages** à leur rythme.



Consumer record

Un **ConsumerRecord** contient plus de **Metadata** qu'un **ProducerRecord**.

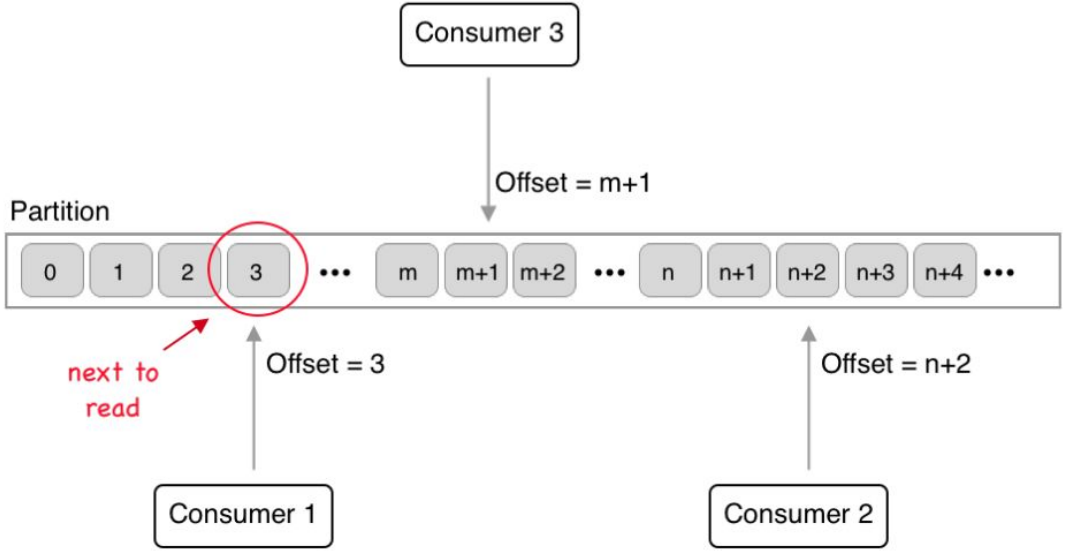
Certaines **Metadata** qui étaient optionnelles au niveau du **ProducerRecord** ne le sont plus côté **ConsumerRecord** (comme les headers, ou le timestamp)



Consumer offset & commit

ℹ️ Le **Topic** interne __consumer_offsets permet de suivre les **Consumer Offsets**

ℹ️ Lorsqu'un **Consumer** commence à lire des **Messages** d'une **Partition** pour la première fois, il peut soit commencer au début, soit à la fin de la auto.offset.reset (auto.offset.reset)



Consumer offset & commit

Auto Commit

- L'auto commit est activé par défaut sur les Consumers
- L'auto commit se fait immédiatement après le poll(), seulement si le temps passé depuis le dernier commit est supérieur à auto.commit.interval.ms (default: 5sec)
- L'**Offset** qui est commit est celui du premier **Message** du **Batch** qui vient d'être **Poll**
- L'auto commit permet de réaliser de l'**At Least Once** delivery, mais pas de l'**Exactly Once**

Consumer offset & commit

Auto Commit

```
Poll finish.. consumer-offset: 1010 - committed-offset: 1000 17:07:05
Poll finish.. consumer-offset: 1020 - committed-offset: 1000 17:07:07
Poll finish.. consumer-offset: 1030 - committed-offset: 1000 17:07:09
Poll finish.. consumer-offset: 1030-1039 - committed-offset: 1030 17:07:11 -> commit when poll finish because of elapsed
time(6 sec) > commit interval(5 sec)
Poll finish.. consumer-offset: 1050 - committed-offset: 1030 17:07:13
Poll finish.. consumer-offset: 1060 - committed-offset: 1030 17:07:15
Poll finish.. consumer-offset: 1070 - committed-offset: 1060 17:07:17 -> auto commit
Poll finish.. consumer-offset: 1080 - committed-offset: 1060 17:07:19
Poll finish.. consumer-offset: 1090 - committed-offset: 1060 17:07:21
Poll finish.. consumer-offset: 1100 - committed-offset: 1090 17:07:23 -> auto commit
```

Manual Commit

Il est possible de désactiver l'auto commit (enable.auto.commit=false) et de commit manuellement.

Un manual commit peut être synchrone ou asynchrone :

- commitAsync()
  - Peut prendre une **Callback** en paramètre
- commitSync()

External offset management

Il est possible de stocker les **Offsets** à l'extérieur de **Kafka** (dans une base de données par exemple).

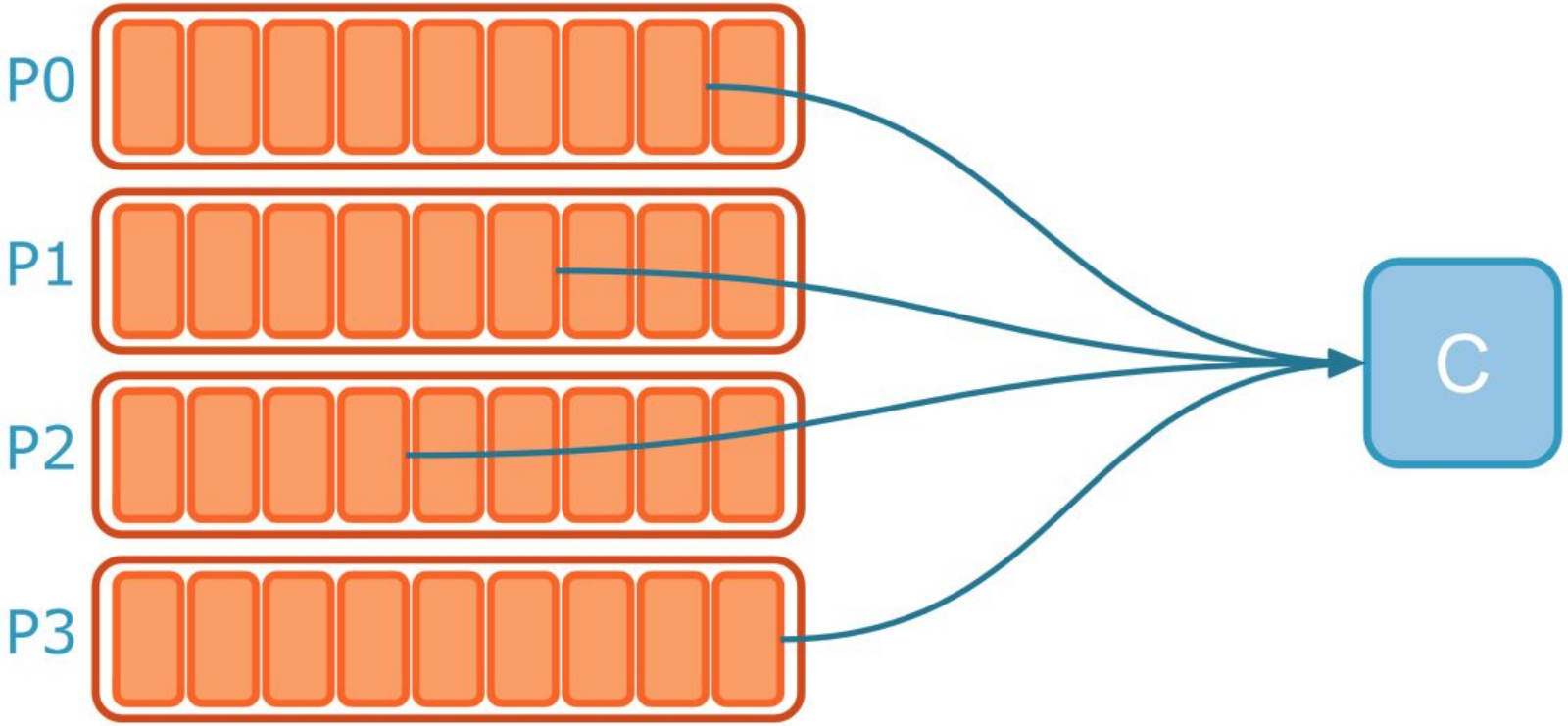
Dans ce cas il faut :

- Désactiver l'auto commit (enable.auto.commit=false)
- Utiliser l'Offset du ConsumerRecord pour stocker la position actuelle
- Avant le prochain Poll, utiliser la méthode seek() pour se position sur l'Offset enregistré précédemment

Plus d'infos ici : https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#rebalancecallback

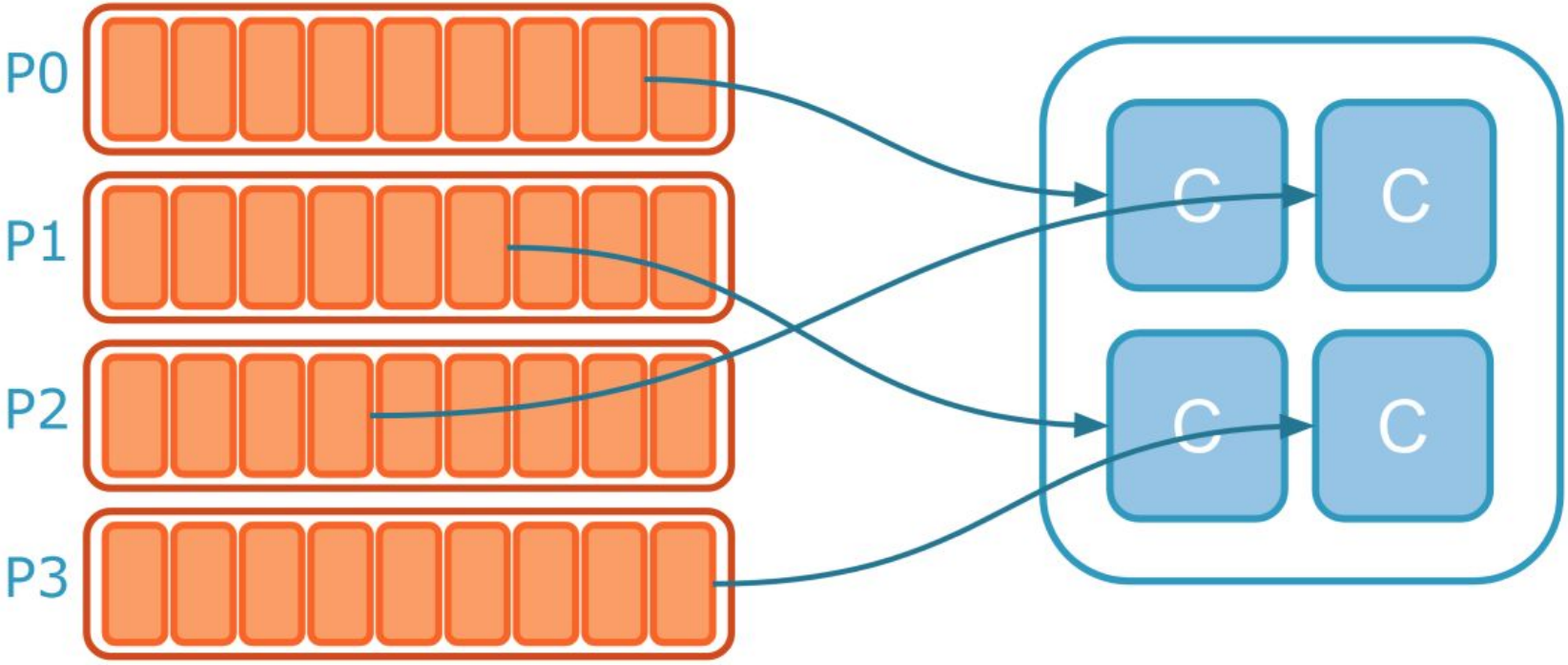
Consumer Groups

Single consumer



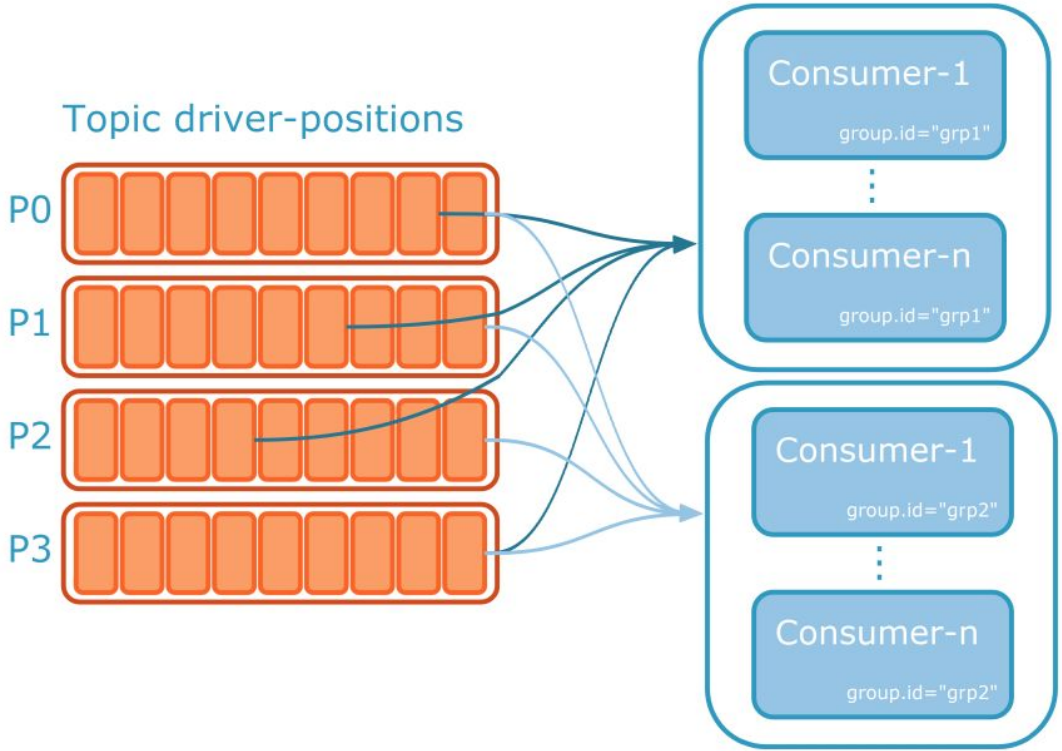
Consumer Groups

Consumer Group



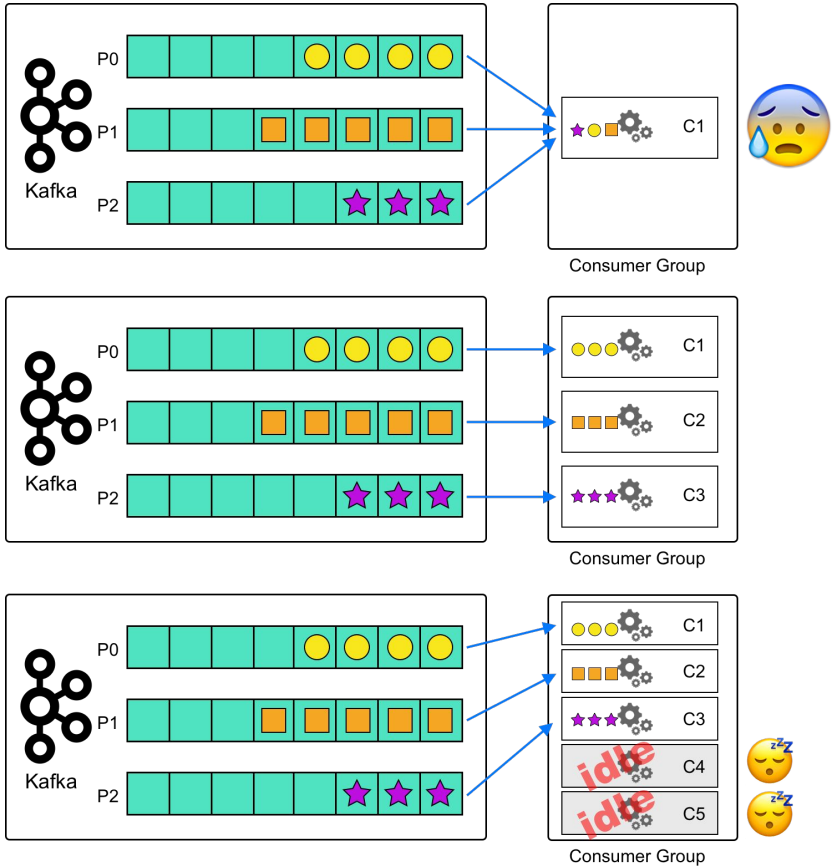
Consumer Groups

Multiple Consumer Groups



Scalability

ℹ️ La scalabilité est limitée par le nombre de Partitions



Consumer properties

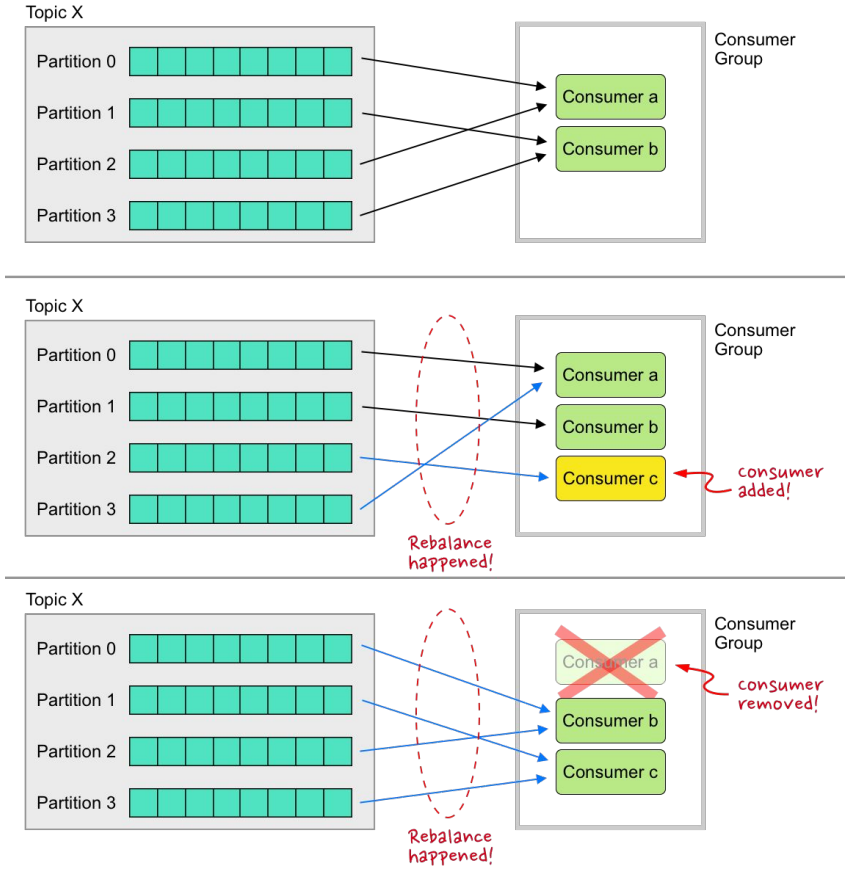
Name	Description
bootstrap.servers	La liste des host:port des **Brokers** utilisés pour la connection initiale
key.deserializer	Classe utilisée pour désérialiser la **Key**. Doit implémenter Deserializer
value.deserializer	Classe utilisée pour désérialiser la **Value**. Doit implémenter Deserializer
partition.assignment.strategy	Détermine la stratégie à utiliser lors de l'assignation des **Partitions** aux **Consumers** d'un groupe. Default : RangeAssignator
heartbeat.interval.ms	Intervalle entre chaque **Heartbeat** envoyé par le **Consumer**. Default : 3sec
session.timeout.ms	Durée pendant laquelle un **Consumer** doit envoyer un **Hearbeat** pour ne pas être considéré comme "dead". Default : 45sec
max.poll.interval.ms	Durée pendant laquelle un **Consumer** doit faire au moins un poll() pour ne pas être considéré comme "dead". Default : 5min
max.poll.records	Nombre maximum de Messages que la méthode poll() peut renvoyer. Default : 500
fetch.max.bytes	Taille maximum (bytes) qu'une fetch request peut renvoyer. Default : 55MB
enable.auto.commit	Active ou non l'auto commit. Default true

Consumer properties

Name	Description
fetch.min.bytes	La taille minimale d'un **Fetch**. Default : 1 byte
max.partition.fetch.bytes	La taille maximale d'un **Fetch** pour une **Partition** donnée . Default : 1MB
fetch.max.wait.ms	La durée max qu'un Broker va attendre avant de répondre à un **Fetch** dans le cas où il n'y a pas assez de data pour satisfaire fetch.min.bytes. Default : 500ms
isolation.level	Permet de contrôler comment lire les messages qui ont été écrit dans un contexte transactionnel. Default : read_uncommitted

Consumer Groups

Rebalances



Rebalances

- Une **Rebalance** de l'assignation des **Partitions** au sein d'un **Consumer Group** est déclenchée lorsqu'un **Consumer** quitte ou rejoint le **Consumer Group** ou lorsque le nombre de **Partitions** d'un **Topic** change

- La consommation est arrêtée pendant la **Rebalance**. Aucune données n'est perdue pendant la **Rebalance**

- Les Partitions sont assignées automatiquement par le Consumer Group Protocol en suivant la stratégie (partition.assignment.strategy)

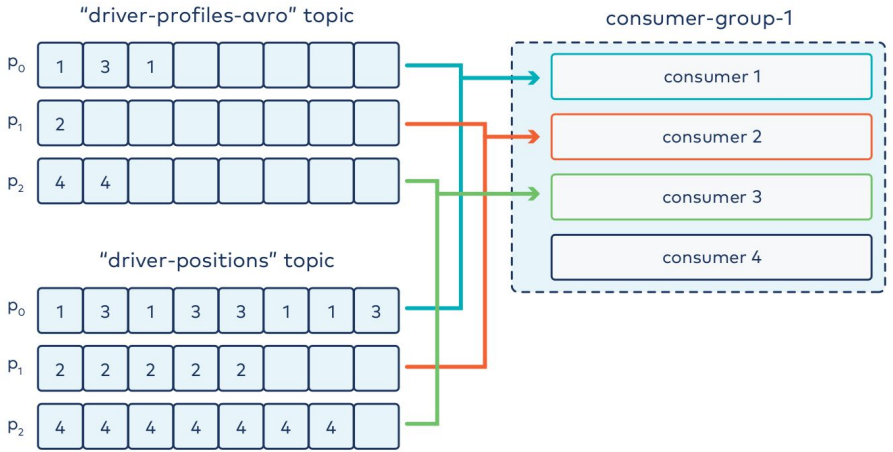
Partitions assignment : Range

La stratégie RangeAssignator permet d'assigner les mêmes numéros de Partitions de Topics différents aux mêmes Consumers.

Tous les **Topics** doivent être "co-partitionnés". Ils doivent avoir :

- Le même nombre de **Partitions**
- Le même **Partitioner**
- Les mêmes **Keys**

C'est la stratégie par défaut.

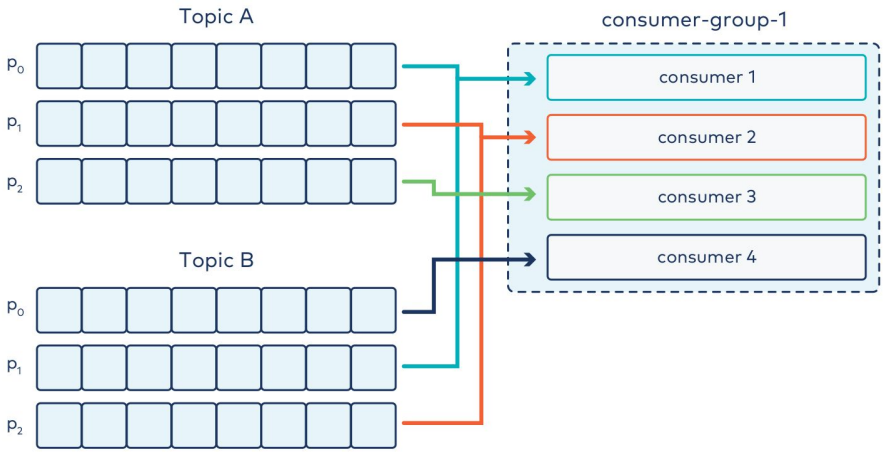


Partitions assignment : Round Robin

La stratégie RoundRobinAssignator permet d'assigner les **Partitions** en suivant l'algorithme Round Robin.

L'objectif est d'assigner le même nombre de **Partitions** à chaque **Consumer**.

Si un Consumer Group n'est abonné qu'à un seul Topic, la stratégie RoundRobinAssignator est la plus adaptée.



Partitions assignment : Sticky & Cooperative Sticky

Les stratégies **Range** et **RoundRobin** ne garantissent pas qu'après une **Rebalance**, les **Consumers** vont récupérer les mêmes **Partitions** qu'auparavant.
Si une **Partitions A-0** est assignée au **Consumer 1**, la **Partition A-0** pourrait être assignée au **Consumer 2** après une **Rebalance**.

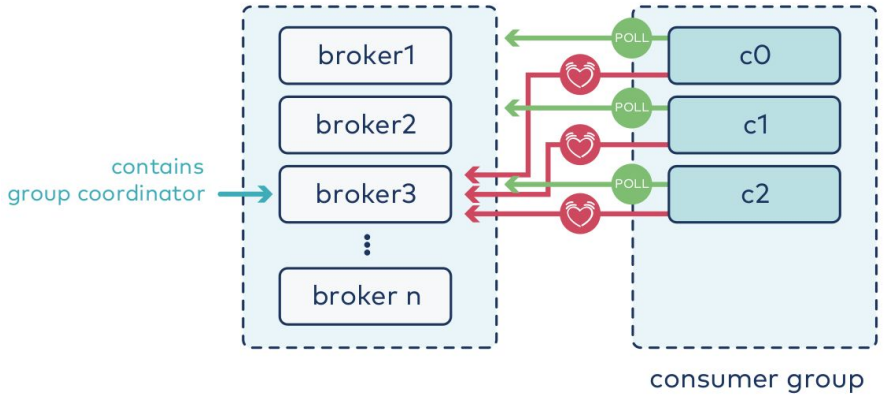
- La stratégie **Sticky** est une extension de la stratégie **RoundRobin** qui en plus préserve les assignations **Partitions**/**Consumers**.

- La stratégie **CooperativeSticky** suit la même logique que la stratégie **Sticky** mais permet de mettre uniquement en pause la consommation des **Consumers** qui sont **Rebalance**, les autres continuant leur consommation.

Dead Consumers

Comment **Kafka** détecte les “Dead Consumers” ?

- Les **Consumers** envoient un **Heartbeat** au **Group Coordinator** (**Broker)** toutes les 3sec (heartbeat.interval.ms)
- Si un **Consumer** n’envoie pas au moins un **Heartbeat** en 45sec (session.timeout.ms), il est considéré comme “dead” et une **Rebalance** est déclenchée
- Si un **Consumer** ne fait pas au moins un poll() en 5min (max.poll.interval.ms), il est considéré comme “dead” et une **Rebalance** est déclenchée



Performance tuning

	High Throughput	Low Lantency
Objectif	Récupérer plus de **Records** par **Batch**	Consommer les **Records** le plus vite possible après qu'ils aient été produits
Tuning	• fetch.min.bytes ↗️ pour indiquer au Broker d'attendre qu'un Batch fasse au moins une certaine taille • fetch.max.wait.ms ↗️ permet toutefois d'indiquer au Broker de ne pas attendre trop longtemps	• fetch.min.bytes=1 ↗️ pour indiquer au Broker de renvoyer des résultats dès que possible

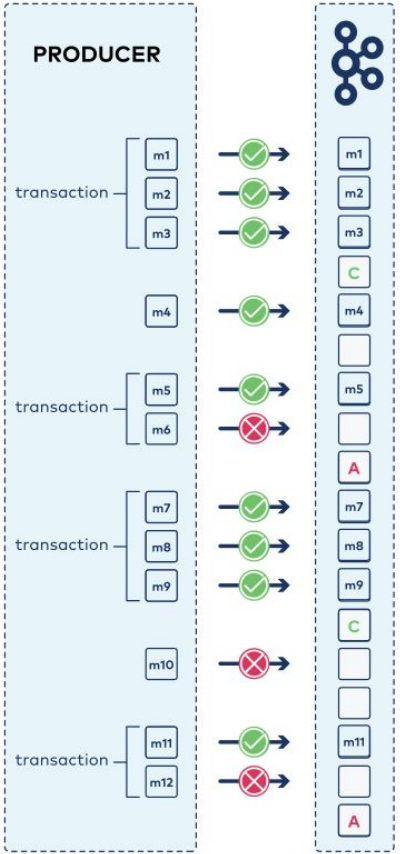
Transactions

La propriété isolation.level permet d'indiquer au **Consumer** s'il doit lire ou non les **Records** transactionnels qui n'ont pas été commit.

- read_uncommitted : le **Consumer** va lire tous les **Records**, même si la **Transaction** est toujours en cours, ou qu'elle a été annulée (abort)

- read_committed : le **Consumer** lit uniquement les **Records** des **Transactions** qui ont été commit. Comme la méthode poll() retourne toujours les **Records** dans l'ordre de l'**Offset**, elle ne retourne les **Records** que jusqu'au dernier **Offset** stable (**L**ast **S**table **O**ffset)

Transactions



Dead Letter Queue

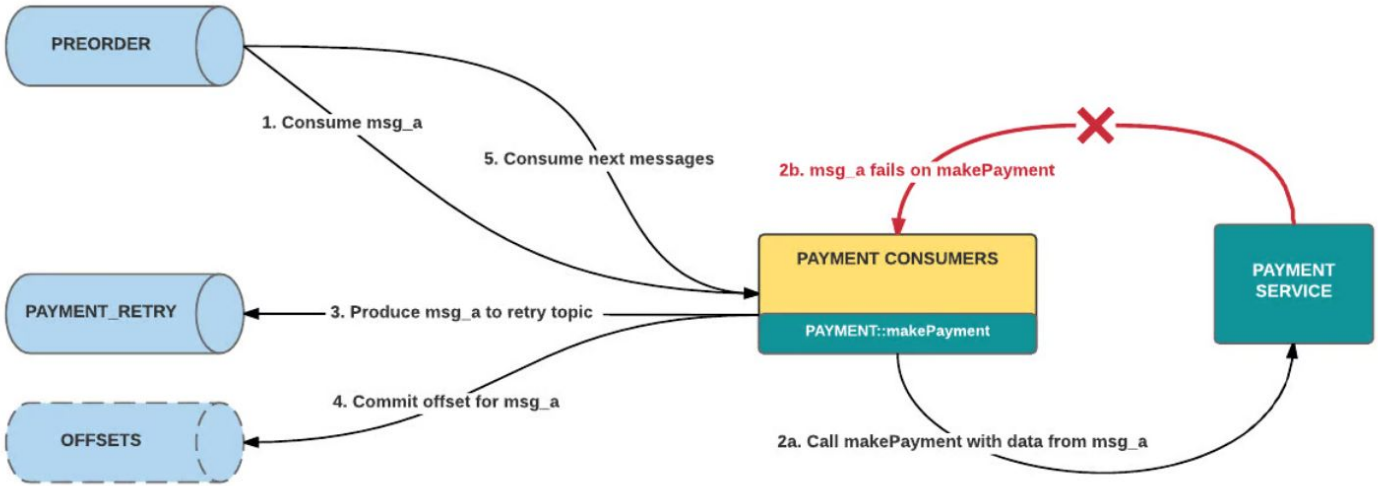
Une **DLQ** (Dead Letter Queue) est un **Topic** dans le cluster **Kafka** qui joue le rôle de destination pour les **Messages** qui n'ont pas pu être process par les **Consumers**.

Les scénarios les plus fréquents expliquant le besoin d'une DLQ sont les suivants :

- Le **Consumer** obtient une erreur lors de la désérialisation du **Message**. Cela pourrait arriver si un **Message** a été sérialisé en Avro alors que le **Consumer** s'attend à du JSON.
- Une erreur est levée lors du processing du **Message**

Retry Topics

Une autre solution courramment utilisée est de publier les **Messages** en échec dans un “retry” **Topic**



Exemples de code

Java

```
props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker:9092");
// other properties

consumer = new KafkaConsumer<>(props);

try {
   consumer.subscribe(Arrays.asList("my_topic", "my_second_topic"));

   while (true) {
       records = consumer.poll(Duration.ofMillis(100));

       for (ConsumerRecord<K, V> record: records) {
           System.out.printf("offset: %d, key: %s, value, %s\n", record.offset(), record.key(), record.value());
       }
   }
} finally {
   consumer.close();
}
```

Exemples de code

Spring

```java
@Component
@KafkaListener(
    containerFactory = "myContainerFactory",
    topics = "my-topic",
    id = "my-group-id",
    clientIdPrefix = "my-consumer-id"
)
public class MyEventsListener {

    @KafkaHandler
    public void listenMyEvents(
        MyRecordValue record,
        @Header(KafkaHeaders.RECEIVED_KEY) MyRecordKey key
    ) {
        // Do some stuff
    }

    @KafkaHandler
    public void listenMyOtherEvents(
        MyOtherRecordValue record,
        @Header(KafkaHeaders.RECEIVED_KEY) MyRecordKey key
    ) {
        // Do some other stuff
    }
}
```

Exemples de code

Java : consumer-process-produce

```
producer.initTransactions();
while (true) {
   ConsumerRecords<K, V> inputRecords = consumer.poll( ... );
   Map<TopicPartition, OffsetAndMetadata> offsetMap = getOffsets(inputRecords);

   try {
       producer.beginTransaction();
       for (ConsumerRecord<K, V> inputRecord : inputRecords) {
           List<ProducerRecord<K, V>> outputRecords = doStuff(inputRecord);
           for (ProducerRecord<K, V> outputRecord : outputRecords) {
               producer.send(outputRecord);
           }
       }
       producer.sendOffsetsToTxn(offsetMap, "consumer-group-id");
       producer.commitTransaction();
   } catch (ProducerFencedException e) {
       producer.close();
   } catch (KafkaException e) {
       producer.abortTransaction();
   }
}
```

Consommer des Messages

Le module commons-eventbus-api expose l'interface MessageBroker permettant de s'abonner à des types de Messages d'un **Topic** Applicatif :

```
init {
    broker.subscribe<TriggerMessageCommandService>().apply {
        onConsume<HardwareCSMSTriggerMessageResponseEvent> { message →
            handleResponseEvent(message.payload)
        }

        onConsume<HardwareCSMSTriggerMessageBootNotificationEvent> { message →
            handleNotificationEvent(message.payload)
        }

        …
    }
}
```

Izivia - Consommer des Messages

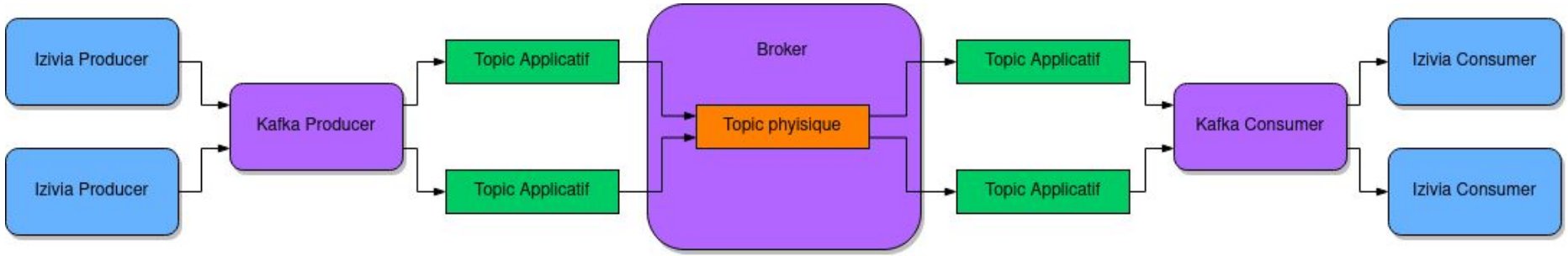
onBroadcast

```kotlin
/**
* Subscribes for a given message type, broadcasted to all replicas (pods).
*
* Each time a message of the given type is received on the bus, it will be delivered to the consumer callback.
*
* The subscription is done on all replicas for all events - therefore the consumption is broadcasted to all replicas.
*
* It means that if multiple replicas subscribe on the same topic with the same consumerId, all replicas will get all events.
*
* If an exception is raised in the callback during the processing of a message, no retry will be performed.
*
* Implementation note:
* This consumer will use a single thread to poll for events, guarantiying the order of processing. If the processing
* of events is too long and need to be done in parallel, consider handling it with an executor of splitting the
* work in multiple small units of work.
*
* In case of error in the callback call, it will  be logged with all message details, making it possible
* to reprocess it manually.
*
* @param consumer the callback invoked with each message received. The message has the event + additional data
* such as the headers
* @param <T> the type of events
**/
inline fun <reified T> onBroadcast(
   noinline consumer: Consumer<Message<T>>
) = eventBus.subscribeBroadcast(subscriberClass, T::class.java, consumer)
```

onConsume

```kotlin
/**
* Subscribes for a given message type, to consume the message once and process it.
*
* WARNING: the consumption has at least once guaranty. It means the processing may be done more than once.
*
* Each time a message of the given type is received on the bus, it will be delivered to the consumer callback once per event.
*
* The subscription is done so that only one pod get the event - sharing the events among the pods per consumerId (consumer class).
*
* It means that if multiple pods subscribe on the same topic with the same subscriberClass, each event will be sent to
* one callback on one pod (qith at least once guaranty). Therefore the event is consumed only once for a single subscription.
*
* If an exception is raised in the callback during the processing of a message, no retry will be performed.
*
* Implementation note:
* This consumer will use a single thread to poll for events, guarantiying the order of processing. If the processing
* of events is too long and need to be done in parallel, consider handling it with an executor of splitting the
* work in multiple small units of work.
*
* In case of error in the callback call, it will  be logged with all message details, making it possible
* to reprocess it manually.
*
* @param consumer the callback invoked with each message received. The message has the event + additional data such as the headers
* @param <T> the type of events
**/
inline fun <reified T> onConsume(
   noinline consumer: Consumer<Message<T>>
) = eventBus.subscribeConsume(subscriberClass, T::class.java, consumer)
```

Rappel


Izivia Producer
Izivia Producer
Kafka Producer
Topic Applicatif
Topic Applicatif
Broker
Topic phyisique
Topic Applicatif
Topic Applicatif
Kafka Consumer
Izivia Consumer
Izivia Consumer

Tradeoff

- Consumers lents
  - Les **Kafka Consumers** sont partagés entre plusieurs **Izivia Consumers**
  - Les **Kafka Consumers** s'abonnent à plusieurs **Topics** physiques
  - Un warning dans les logs est émis si la durée de processing est supérieur au threshold
  - Si un **Izivia Consumer** est bloqué, il bloque les autres **Izivia Consumers** qui sont abonnés au même **Topic** applicatif

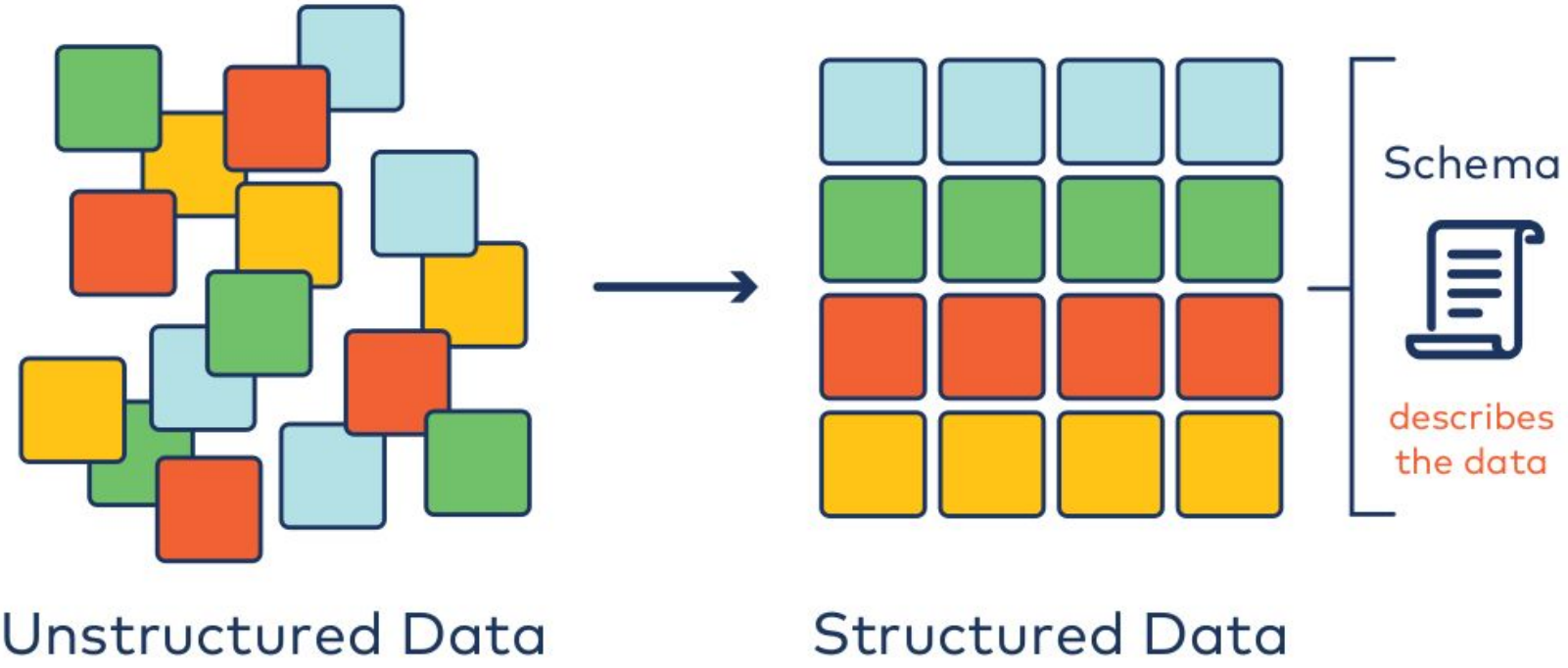
- L'ordre des Messages n'est pas toujours garanti :
  - Le processing des Messages est délégué à des Threads dédiés
  - Si le processing est trop long, une TimeoutException est levée
    - Un check est fait sur le Thread pour savoir s'il est toujours "healthy"
      - Si c'est le cas, on le laisse continuer ses traitements et on démarre un nouveau **Poll**
      - Si ce n'est pas le cas, le Thread est tué => il y a de forte chances que des Messages soient "perdus"

Consommer des messages

Lab

Schema management

Pourquoi des schémas ?



Pourquoi des schémas ?

Imaginez le schéma d'un BusinessCustomer

1980s	1990s	now
• company name	• company name	• company name
• contact person	• contact person	• contact person
• physical address	• physical address	• physical address
• phone	• phone	• phone
• fax number	• fax number	• ~~**fax number**~~
	• **email address**	• email address

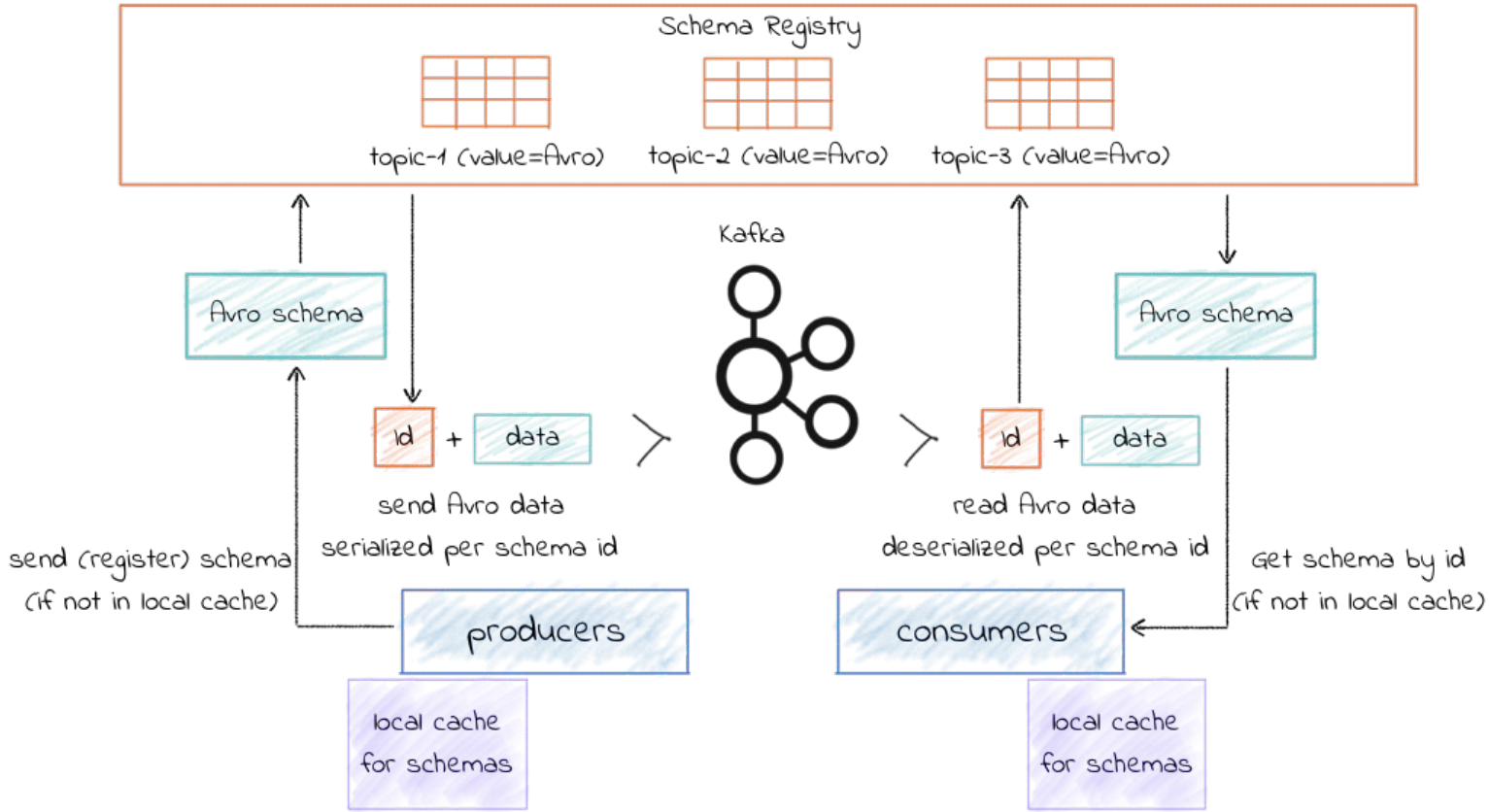
C’est quoi le Schema Registry ?

- Développé par Confluent (Confluent Community License)
- Gestion centralisée des **Schemas**
  - Stock un historique des versions de tous les **Schemas**
  - Fournit des API REST et Java pour stocker et récupérer les **Schemas**
  - Permet de checker la compatibilité des **Records** avec les **Schemas** et lever des erreurs
  - Permet l’évolution des **Schemas**, en fonction du mode de compatibilité choisi
- Il stocke toutes les informations des **Schemas** dans un topic interne (_schemas)
- Il associe un ID à chaque **Schema**. Cet ID de **Schema** est envoyée avec chaque **Record**
- Compatible avec les schémas JSON, Avro et Protobuf

Integration with Schema Registry

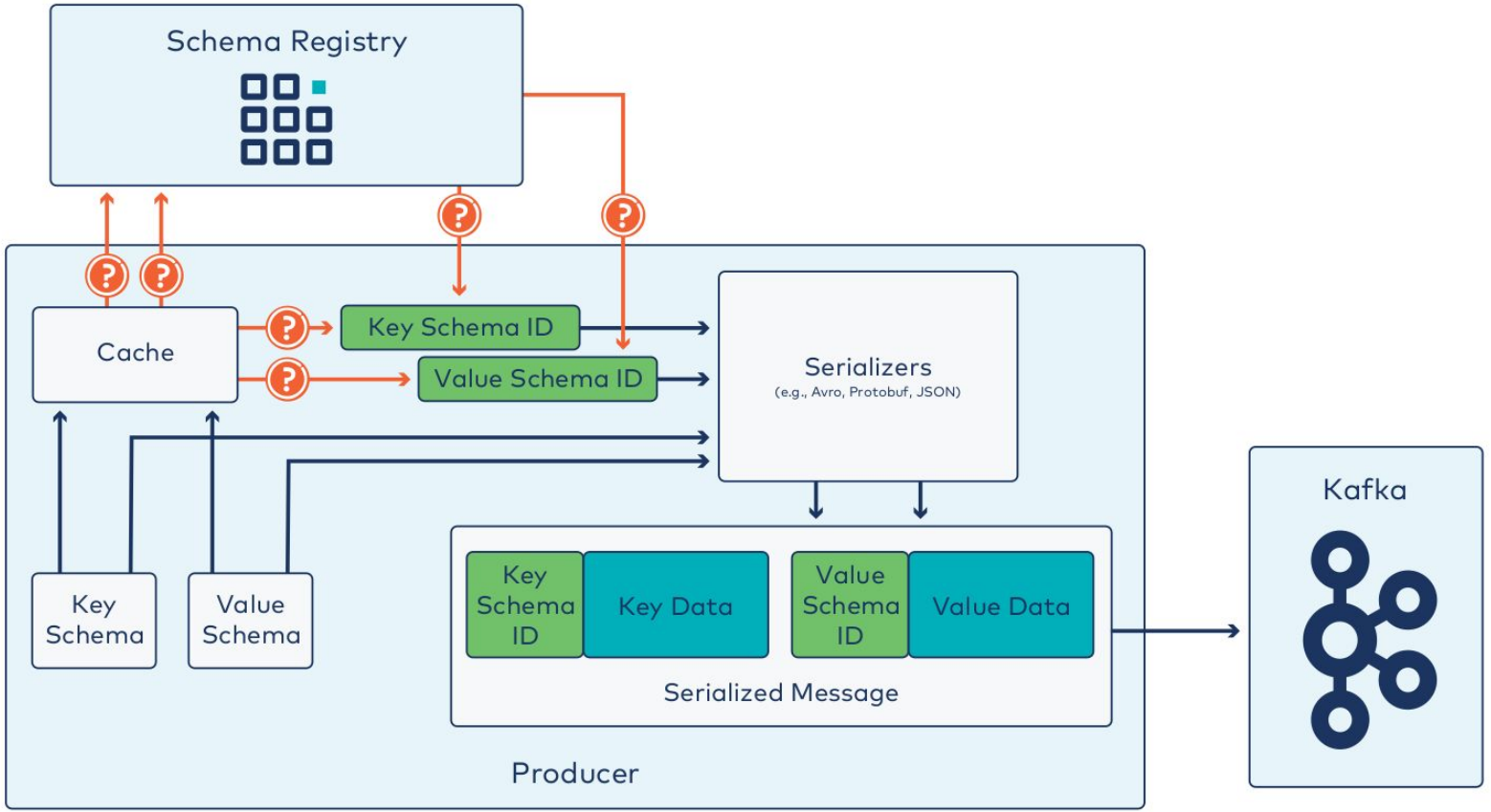
- JVM Clients
- ksqlDB
- Kafka Streams
- Kafka Connect
- Confluent REST Proxy
- Non-Java clients based on librdkafka

Schema registry



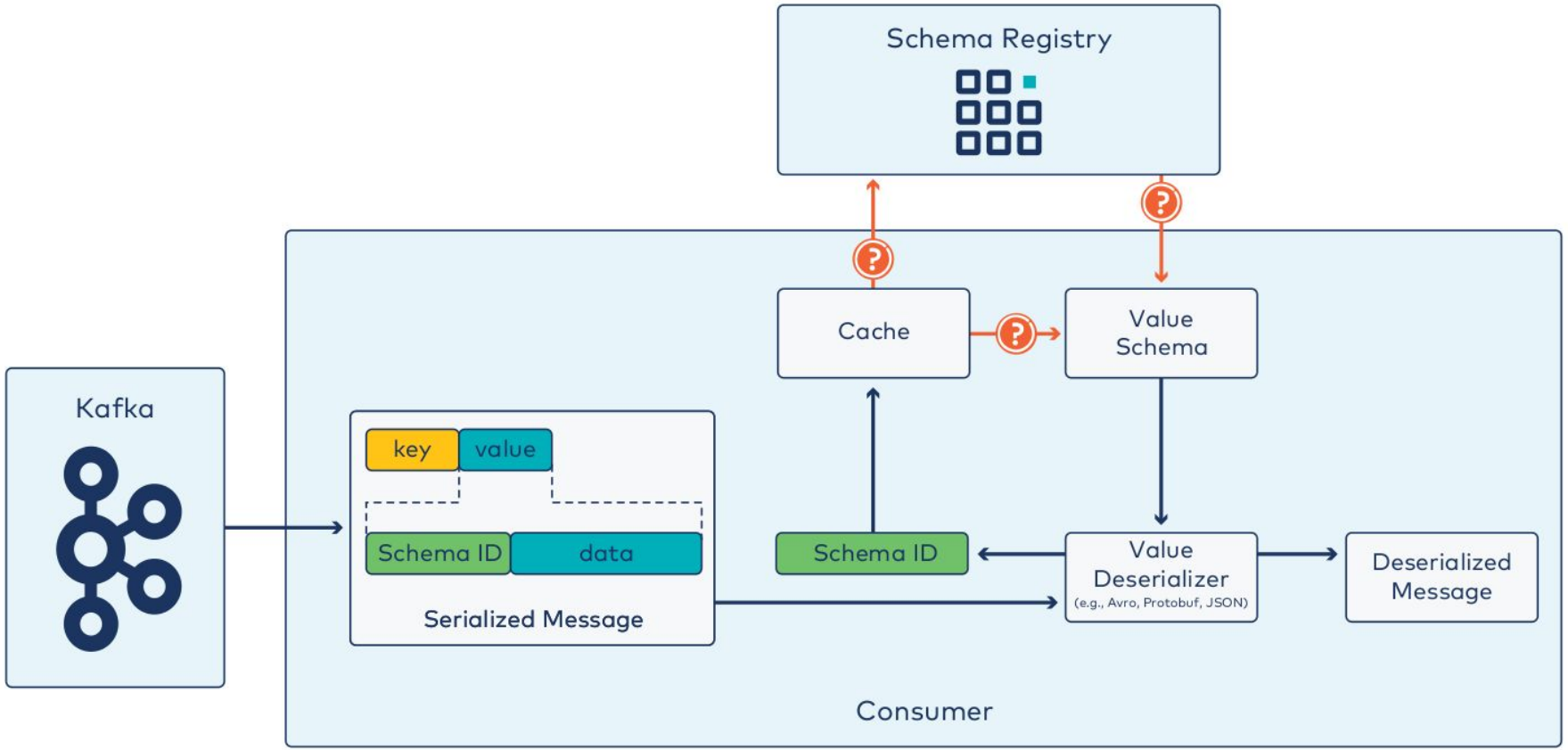
Schema registry

Producers



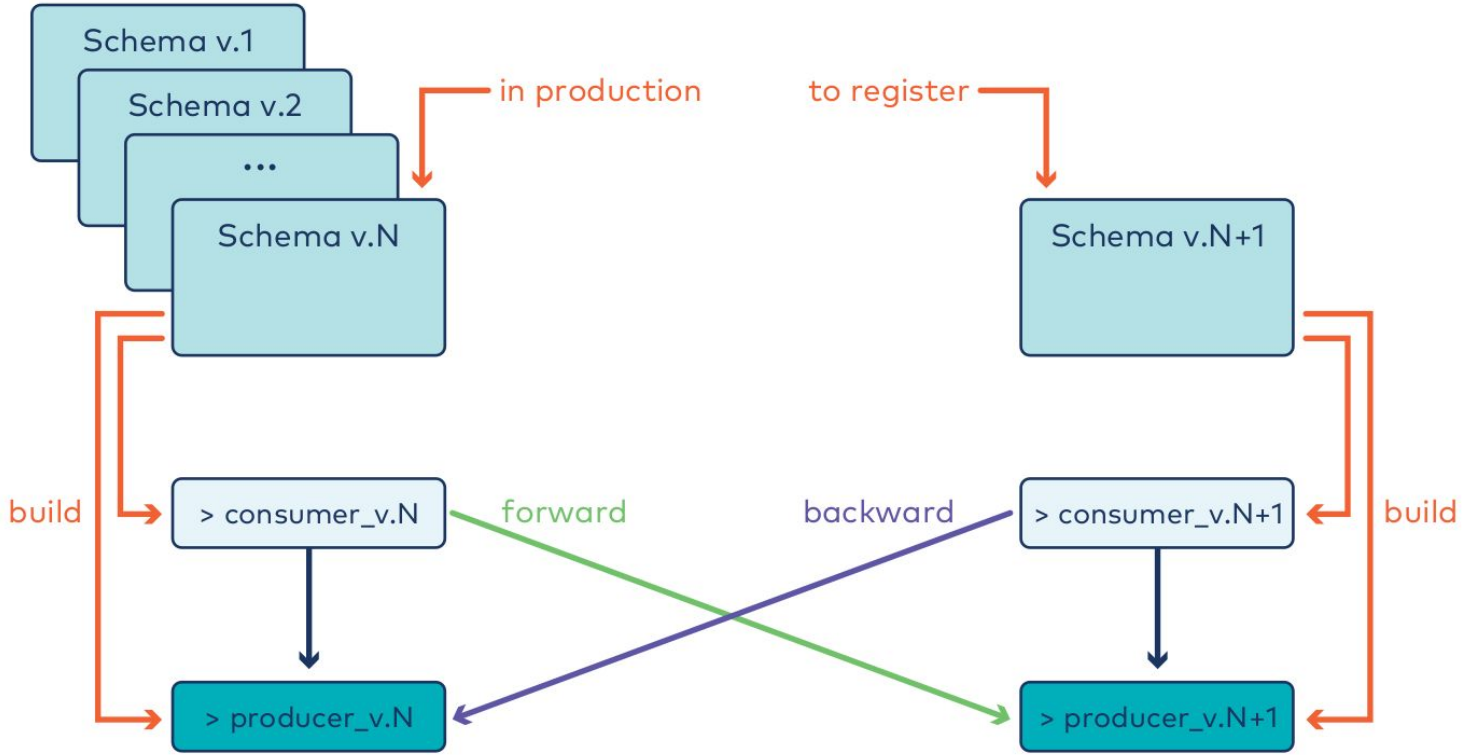
Schema registry

Consumers



Schema registry

Compatibility



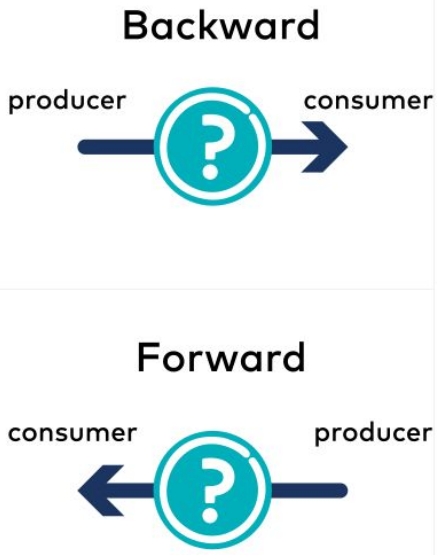
Compatibility

Compatibility Type	Schema Resolution	Upgrade first
BACKWARD	Les **Consumers** utilisant le nouveau **Schema** peuvent lire des **Messages** produits avec le **Schema** précédent. C'est la valeur par défaut	Consumers
BACKWARD_TRANSITIVE	Les **Consumers** utilisant le nouveau **Schema** peuvent lire des **Messages** produits avec tous les **Schemas** précédents	Consumers
FORWARD	Les **Consumers** utilisant le **Schema** précédent peuvent lire des **Messages** produits avec le nouveau **Schema**	Producers
FORWARD_TRANSITIVE	Les **Consumers** utilisant le **Schema** précédent peuvent lire des **Messages** produits avec tous les nouveaux **Schemas**	Producers
FULL	Combinaison de BACKWARD et FORWARD	N'importe lequel
FULL_TRANSITIVE	Combinaison de BACKWARD_TRANSITIVE et FORWARD_TRANSITIVE	N'importe lequel
NONE	Check de compatibilité désactivé	N'importe lequel

Compatibility Example 1

Schema V1 fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



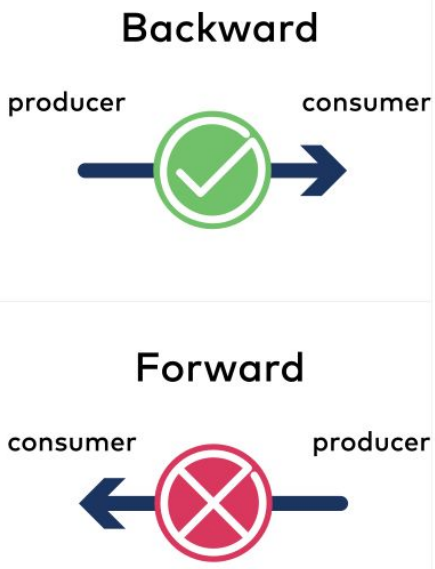
Schema V2 fields

```
"fields":
[
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

Compatibility Example 1

Schema V1 fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



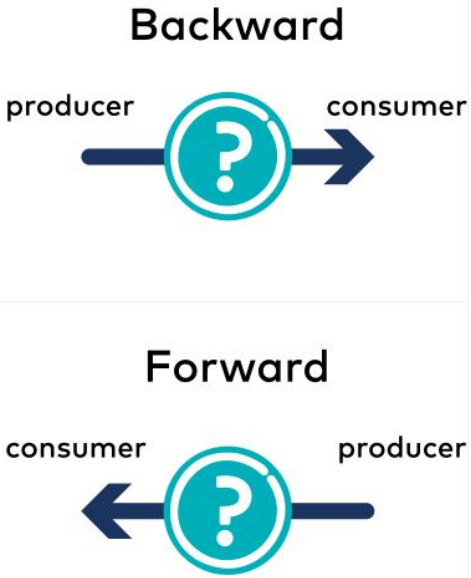
Schema V2 fields

```
"fields":
[
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

Compatibility Example 2

Schema V1 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



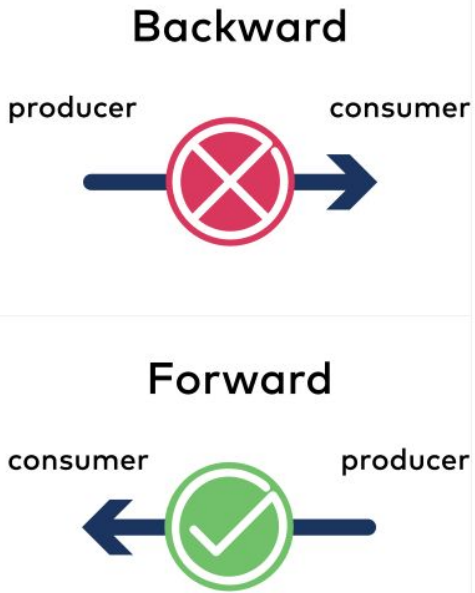
Schema V2 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "hobby",
        "type": "string"
    }
]
```

Compatibility Example 2

Schema V1 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



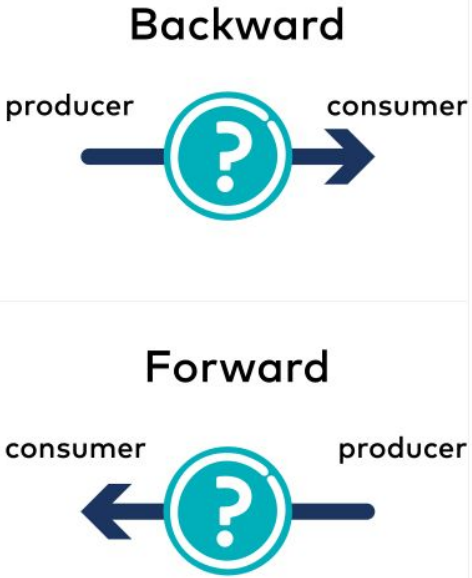
Schema V2 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "hobby",
        "type": "string"
    }
]
```

Compatibility Example 3

Schema V1 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



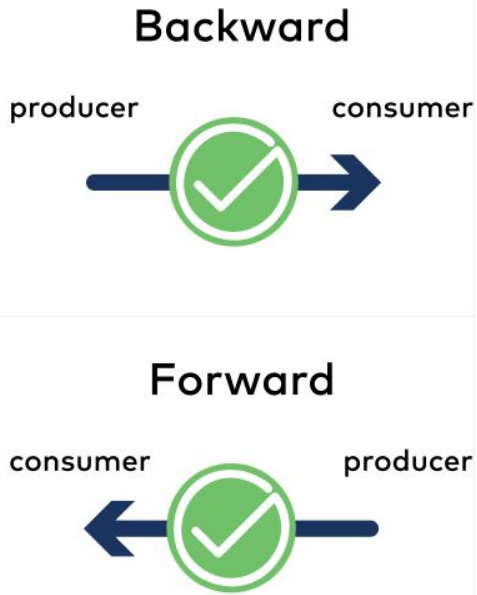
Schema V2 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

Compatibility Example 3

Schema V1 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "age",
        "type": "int",
        "default": -1
    }
]
```



Schema V2 Fields

```
"fields":
[
    {
        "name": "firstname",
        "type": "string"
    },
    {
        "name": "lastname",
        "type": "string"
    },
    {
        "name": "hobby",
        "type": "string",
        "default": ""
    }
]
```

Schema registry

Formats : JSON

```
{
 "record:PositionValue" : {
   "type" : "object",
   "required" : [ "latitude", "longitude" ],
   "additionalProperties" : false,
   "properties" : {
     "latitude" : {"type" : "number"},
     "longitude" : {"type" : "number"}
   }
 }
}
```

Formats : Avro

```
{
 "namespace": "clients.avro",
 "type": "record",
 "name": "PositionValue",
 "fields": [
   { "name": "latitude", "type": "double" },
   { "name": "longitude", "type": "double" }
 ]
}
```

- Apache project
- Schémas au format JSON
- Permet de générer les classes Java associées
- File extension : .avsc
- Sérialisation efficace

Formats : Protobuf

```
syntax = "proto3";

option java_package = "clients.proto";
message PositionValue {
 double latitude = 1;
 double longitude = 2;
}
```

- Développé par Google, licence BSD
- Syntaxe maison pour les schémas
- Permet de générer les classes Java associées
- File extension : .proto
- Sérialisation efficace

Subject Naming Strategies

- **Subject** : scope dans lequel les **Schemas** peuvent évoluer dans le **Schema Registry**
  - Topic : driver-positions-avro
  - Subjects :
    - driver-positions-avro-key => <topic>-key
    - driver-positions-avro-value => <topic>-value
  - C'est la stratégie TopicNameStrategy . C'est celle par défaut
- La stratégie se configure au niveau du **Topic** et des **Clients** :
  - key.subject.name.strategy
  - value.subject.name.strategy
- Il existe d'autres stratégies :
  - TopicRecordNameStrategy
    - <topic>-<recordName>-key et <topic>-<recordName>-value
    - Permet d'avoir plusieurs **Schemas** pour un même **Topic**
  - RecordNameStrategy
    - <recordName>-key et <recordName>-value
    - Permet de partager des **Schemas** entre différents **Topics**

Records Validation

- Par défaut, la validation des **Records** se fait côté **Client**
- Avec la distribution **Confluent** de **Kafka**, il est possible d'activer la validation au niveau du **Broker** :
  - confluent.key.schema.validation : Active la validation des **Keys** des **Records** (**Broker** config)
  - confluent.value.schema.validation : Active la validation des **Values** des **Records** (**Broker** config)
  - confluent.key.subject.name.strategy : Indique la stratégie à utiliser pour la **Key** des **Records** (**Topic** config)
  - confluent.value.subject.name.strategy : Indique la stratégie à utiliser pour la **Value** des **Records** (**Topic** config)
- ⚠️ Sur **Confluent Cloud**, la validation des **Records** par le **Broker** n'est disponible qu'avec des **Dedicated Clusters**

Exemples de code

Subject referencing other Subjects

```
[
 "com.4sh.kafka.OrderCreatedEvent",
 "com.4sh.kafka.OrderUpdatedEvent",
 "com.4sh.kafka.OrderCanceledEvent",
 "com.4sh.kafka.OrderShippedEvent"
]
```

Exemples de code

Java Producer

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");

// Configure serializer classes
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);

// Configure schema repository server
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schema-registry1:8081");

// Create the producer, which expects PositionValue object generated from Avro schema
KafkaProducer<String, PositionValue> producer = new KafkaProducer<>(props);

// Create the key (String) and value (PositionValue object generated from Avro schema)
String key = "driver-1";
PositionValue value = new PositionValue(47.618580396045445, -122.35454111509547);

// Create the ProducerRecord and send it
ProducerRecord<String, PositionValue> record = new ProducerRecord<>("driver-positions-avro", key, value);
producer.send(record);
```

Exemples de code

Java Consumer

```
Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class);
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schemaregistry1:8081");
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");

KafkaConsumer<String, PositionValue> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("driver-positions-avro"));

while (true) {
   ConsumerRecords<String, PositionValue> records = consumer.poll(Duration.ofMillis(100));

   for (ConsumerRecord<String, PositionValue> record : records) {
       System.out.printf("Key:%s Latitude:%s Longitude:%s [partition %s]\n",
                         record.key(), record.value().getLatitude(),
                         record.value().getLongitude(), record.partition()
       );
   }
}
```

Exemples de code

Spring Consumer

```java
@Component
@KafkaListener(
    containerFactory = "myContainerFactory",
    topics = "my-topic",
    id = "my-group-id",
    clientIdPrefix = "my-consumer-id"
)
public class MyEventsListener {

    @KafkaHandler
    public void listenMyEvents(
        MyRecordValue record,
        @Header(KafkaHeaders.RECEIVED_KEY) MyRecordKey key
    ) {
        // Do some stuff
    }

    @KafkaHandler
    public void listenMyOtherEvents(
        MyOtherRecordValue record,
        @Header(KafkaHeaders.RECEIVED_KEY) MyRecordKey key
    ) {
        // Do some other stuff
    }
}
```

Exemples de code

REST API (GET)

```
$ curl -X GET http://schemaregistry1:8081/schemas/ids/1

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json
{"schema": "{\"type\": \"string\"}"}
```

Exemples de code

REST API (POST)

```
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
   --data '{"schema": "{\"type\": \"string\"}"}' \
   http://schemaregistry1:8081/subjects/<topic-name>-value/versions

HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json
{"id":1}
```

Izivia - Schema Management

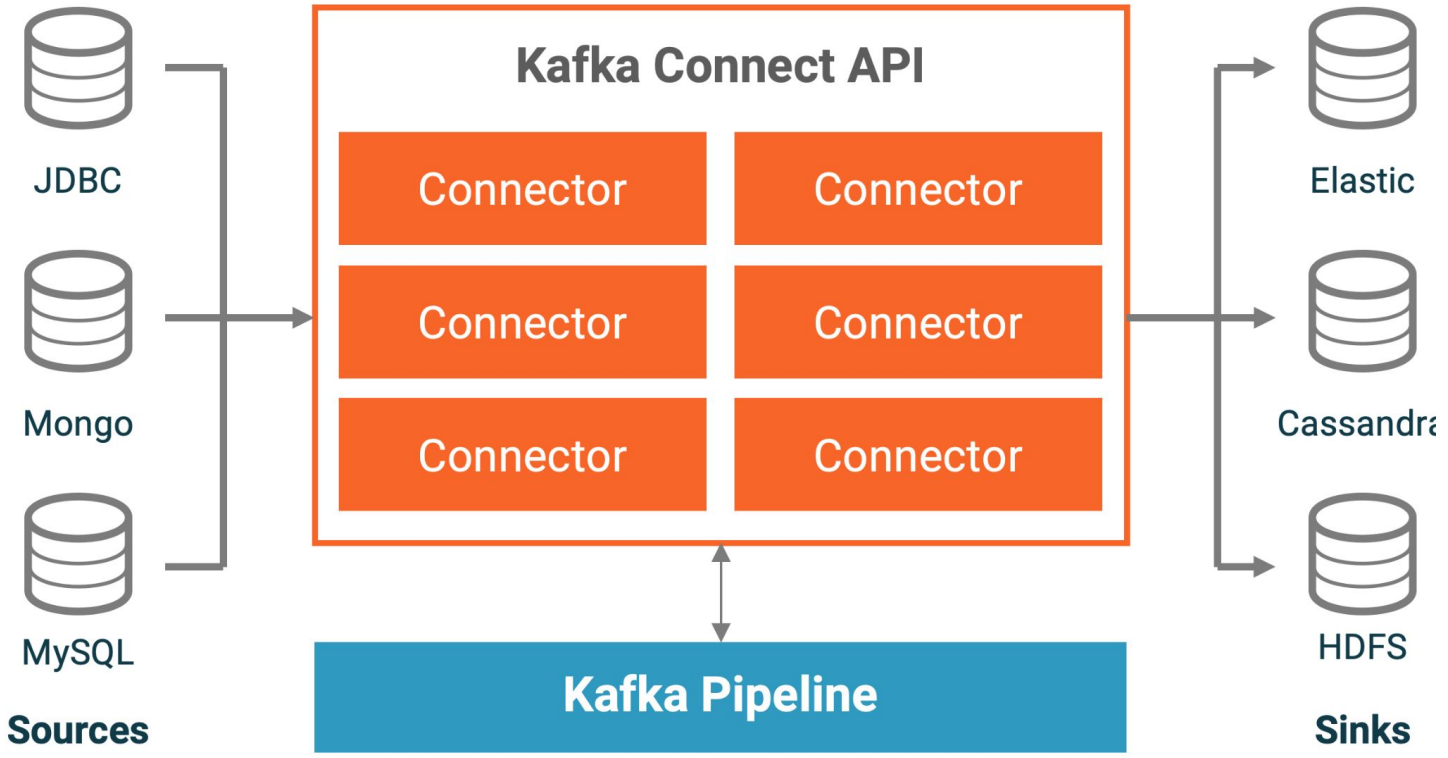
- Le **Schema Registry** n'est pas utilisé sur Izivia
- Les **Messages** sont sérialisés/désérialisés au format JSON
  - Il est envisagé de passer un jour à Protobuf
- Si le schéma d'un **Message** n'est pas compatible avec le précédent, alors il faut créer un nouveau **Topic**

Schema management

Lab

Kafka Connect

Kafka Connect



C’est quoi Kafka Connect ?

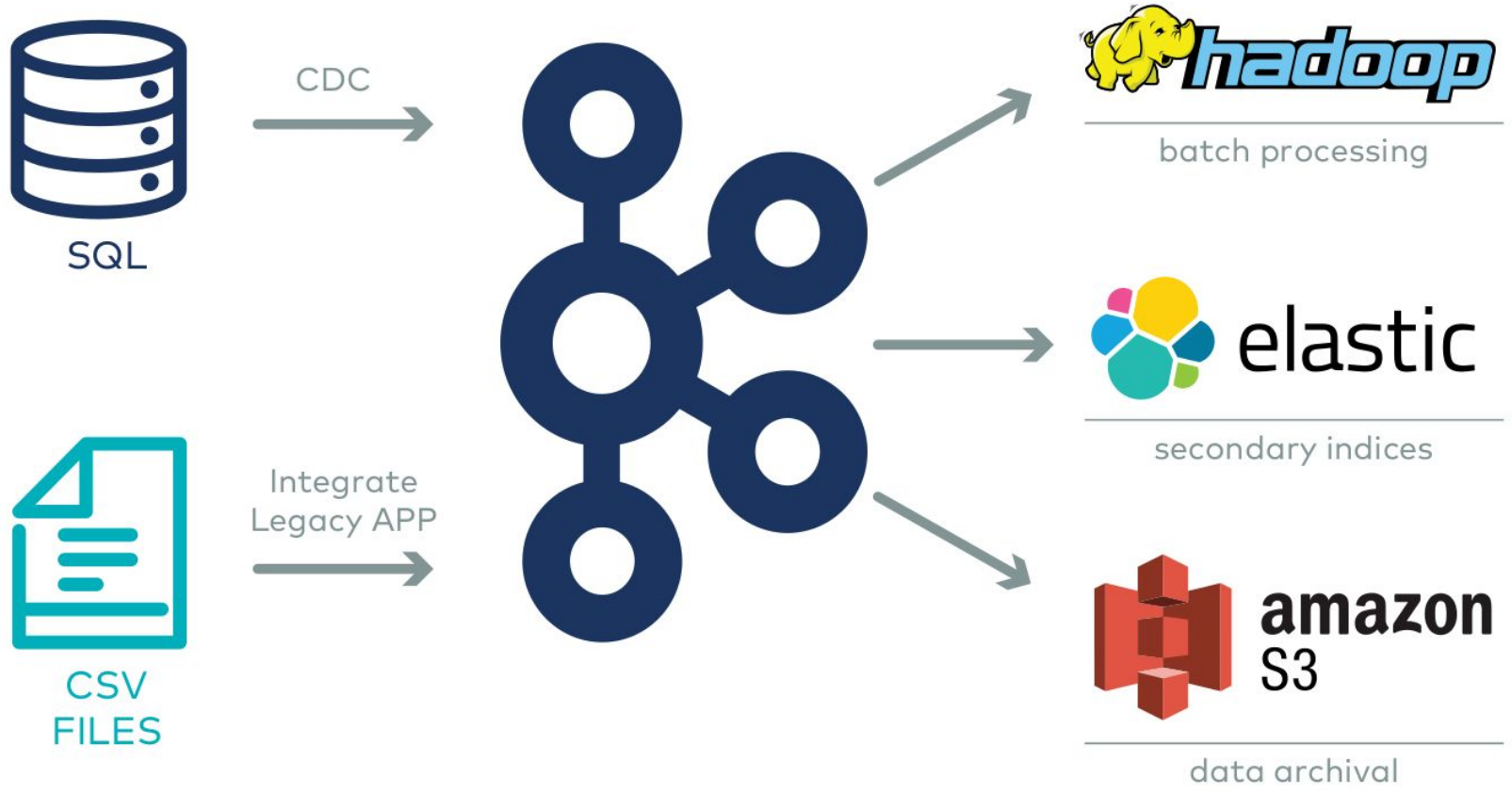
- Fait parti du projet **Apache Kafka**
- Framework pour importer/exporter des data depuis **Kafka**
- On parle de **Source** et de **Sink Connectors**
  - **Source Connectors** : permettent d’importer des données dans **Kafka**
  - **Sink Connectors** : permettent d’exporter des données depuis **Kafka**
- **Kafka Connect** est scalable et tolérant aux pannes de manière intrinsèque
- **Kafka Connect** utilise les API de Producer et Consumer
  - Chaque **Connector** gère ses **Offsets** (Source ou Sink). Certains utilisent des **Topics** à l’image des **Consumers**, d’autres utilisent des solutions externes
- La plupart des systèmes de données possèdent un **Source** et un **Sink Connector** (JDBC, MongoDB, Cassandra, etc)

Use cases

- Stream une base de données SQL dans un **Topic**
- Stream un **Topic** vers une base de données
- Stream des **Topics** vers Elasticsearch
- Lire des fichiers CSV sur un FTP et envoyer le contenu dans un **Topic**
- Etc

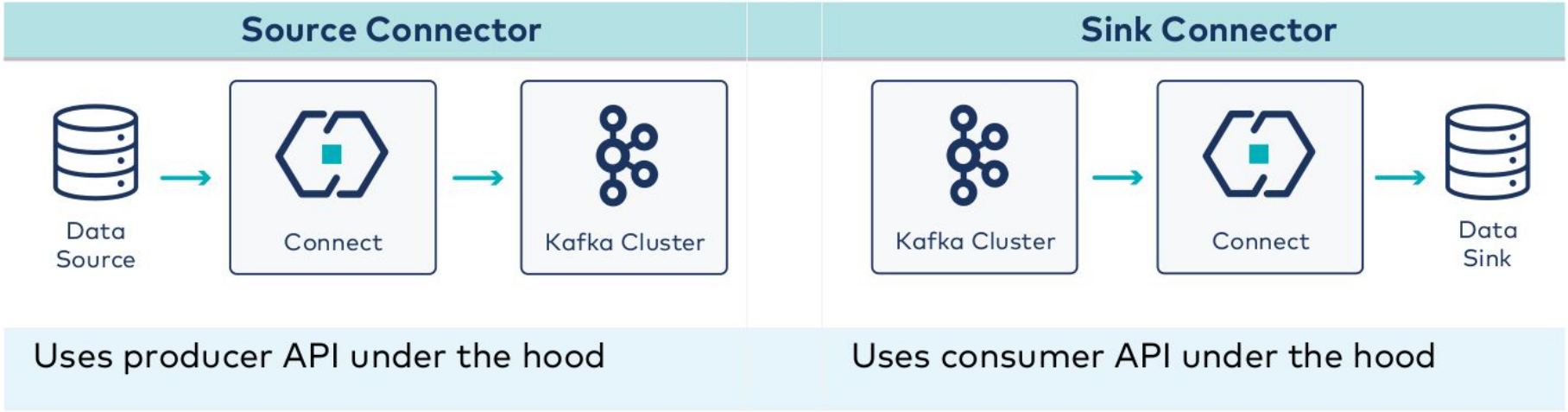
Liste des **Connectors** sur https://confluent.io/hub

Uses cases


SQL
CDC
CSV
FILES
Integrate
Legacy APP
hadoop
batch processing
elastic
secondary indices
amazon
S3
data archival

Source & Sink Connectors

Il y a 2 types de **Connectors**



Distributed vs Standalone Mode

- **Distributed** :
  - Adapté pour la production dans la plupart des cas
  - Permet de paralléliser les **Tasks** et **Connectors** grâce aux **Workers**
  - Fault-tolerance out-of-the-box
  - Ne requiert pas d'orchestrateur (utilise le même mécanisme que les **Consumer Groups**)

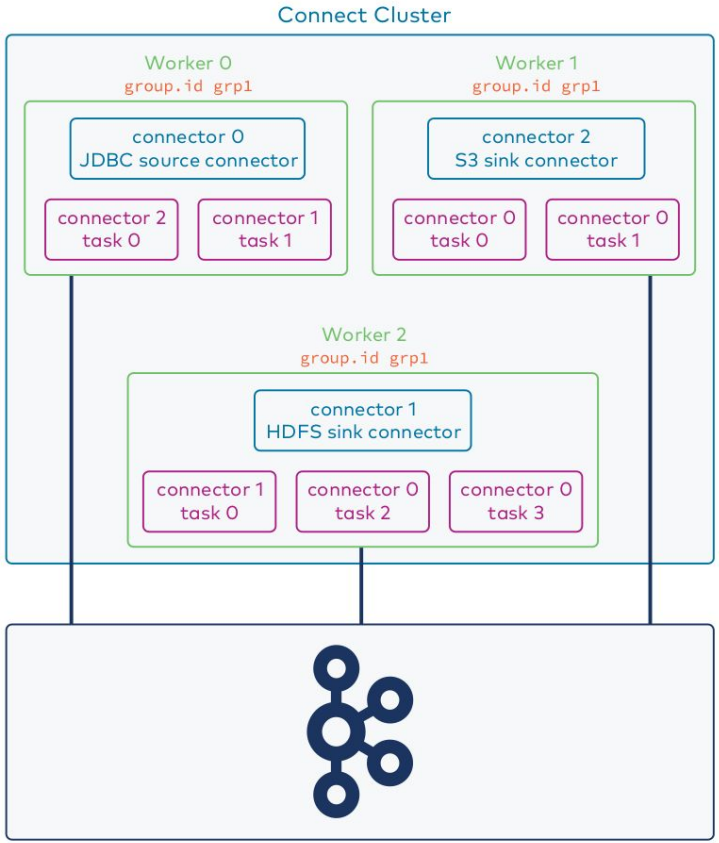
- **Standalone** :
  - 1 seul process exécute tous les **Connectors** et toutes les **Tasks**
  - Pas de fault-tolerance out-of-the-box
  - Adapté pour les phases de développement
  - Certains **Connectors** ne sont compatible qu'avec le mode **Standalone** (Syslog Source Connector)

Kafka Connect décomposition

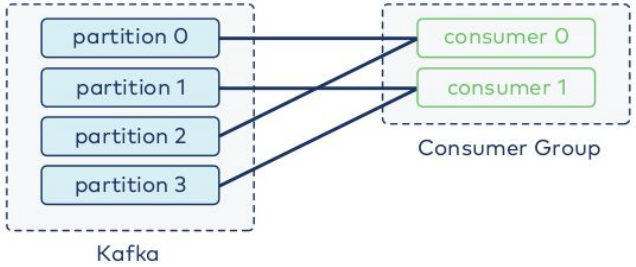
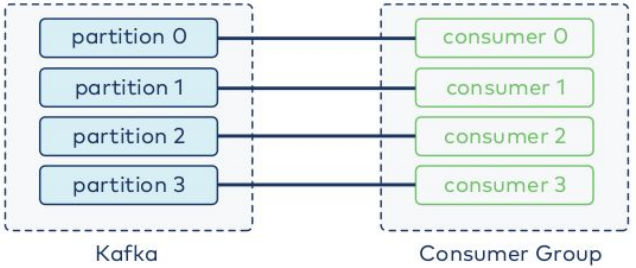
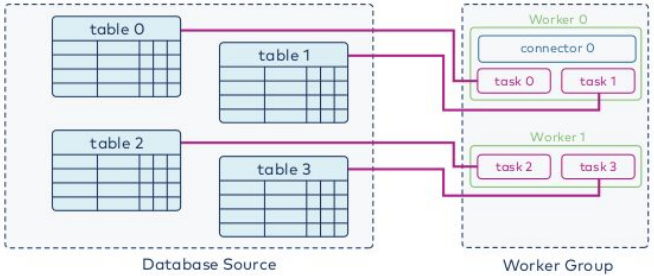
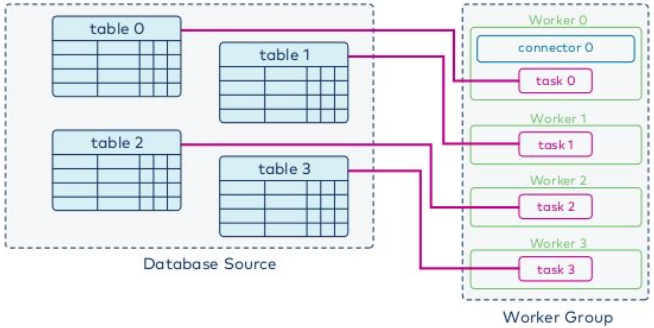
- **Connector** : logique visant à importer/exporter des données depuis ou vers **Kafka**
- **Task** : un **Connector** est découpé en N Tasks permettant de paralléliser les traitements
- **Worker** : Process qui permet d'exécuter des **Connectors** et/ou des **Tasks**. Les Workers ne sont pas gérés par **Kafka**.
  Une pratique courante est d'utiliser **Kubernetes** ou **Confluent Cloud** pour faire tourner des clusters de **Kafka Connect**

- Un **Connector** a :
  - 1..N **Tasks**
- Un **Worker** exécute :
  - 0..N **Connectors**
  - 0..N **Tasks**

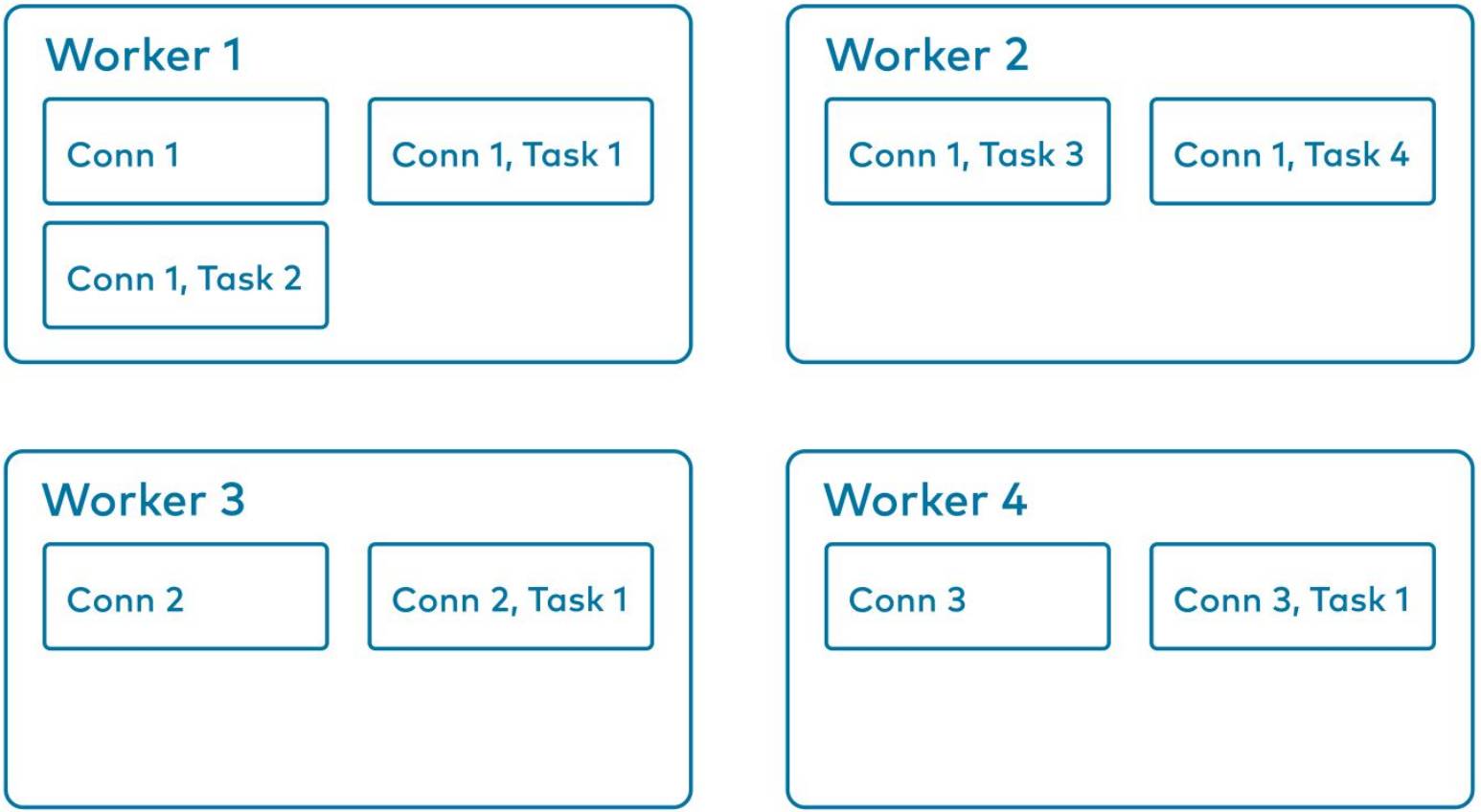
Connect Cluster



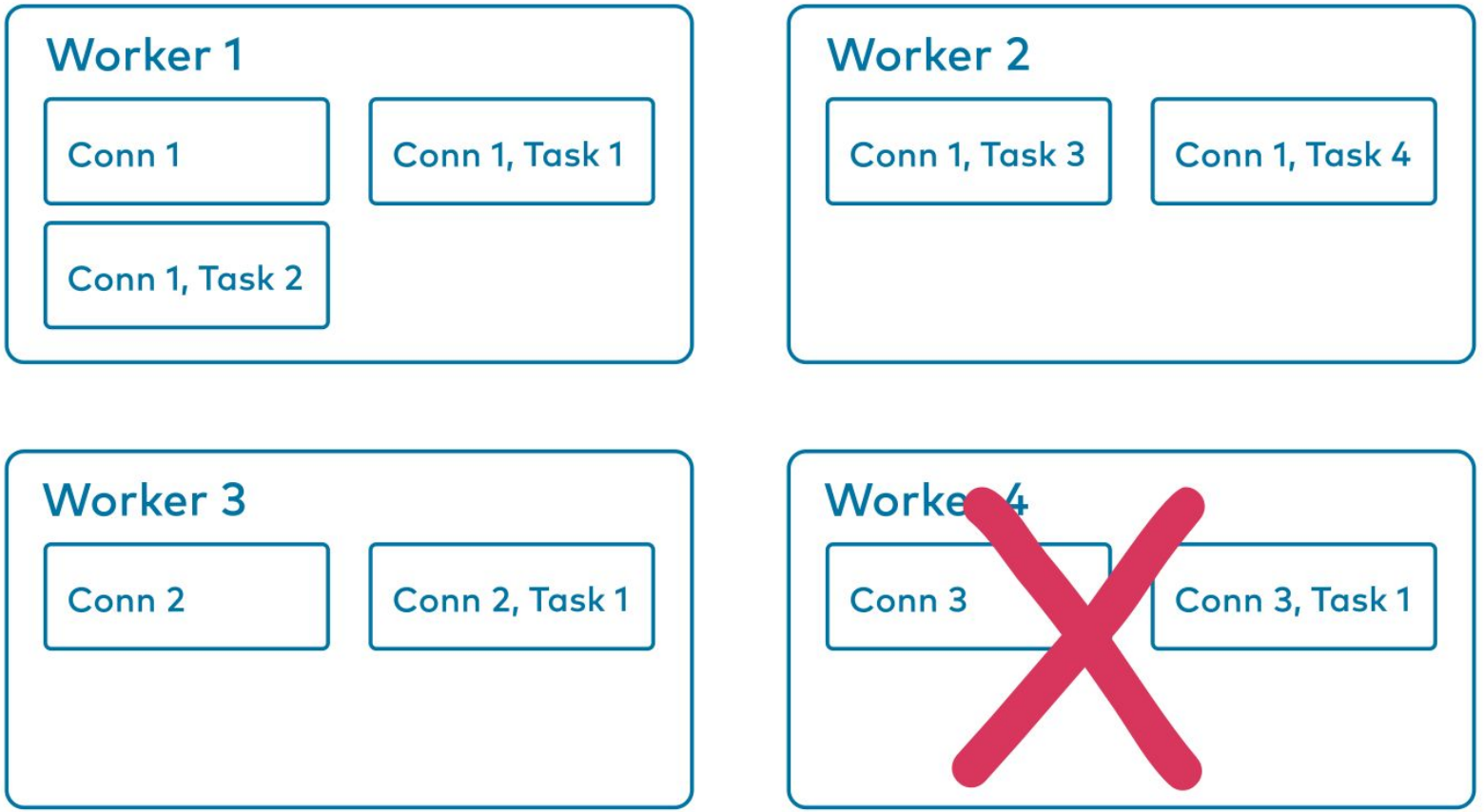
Groups

	Before Scaling	After Scaling
Consumer Group		
Worker Group		

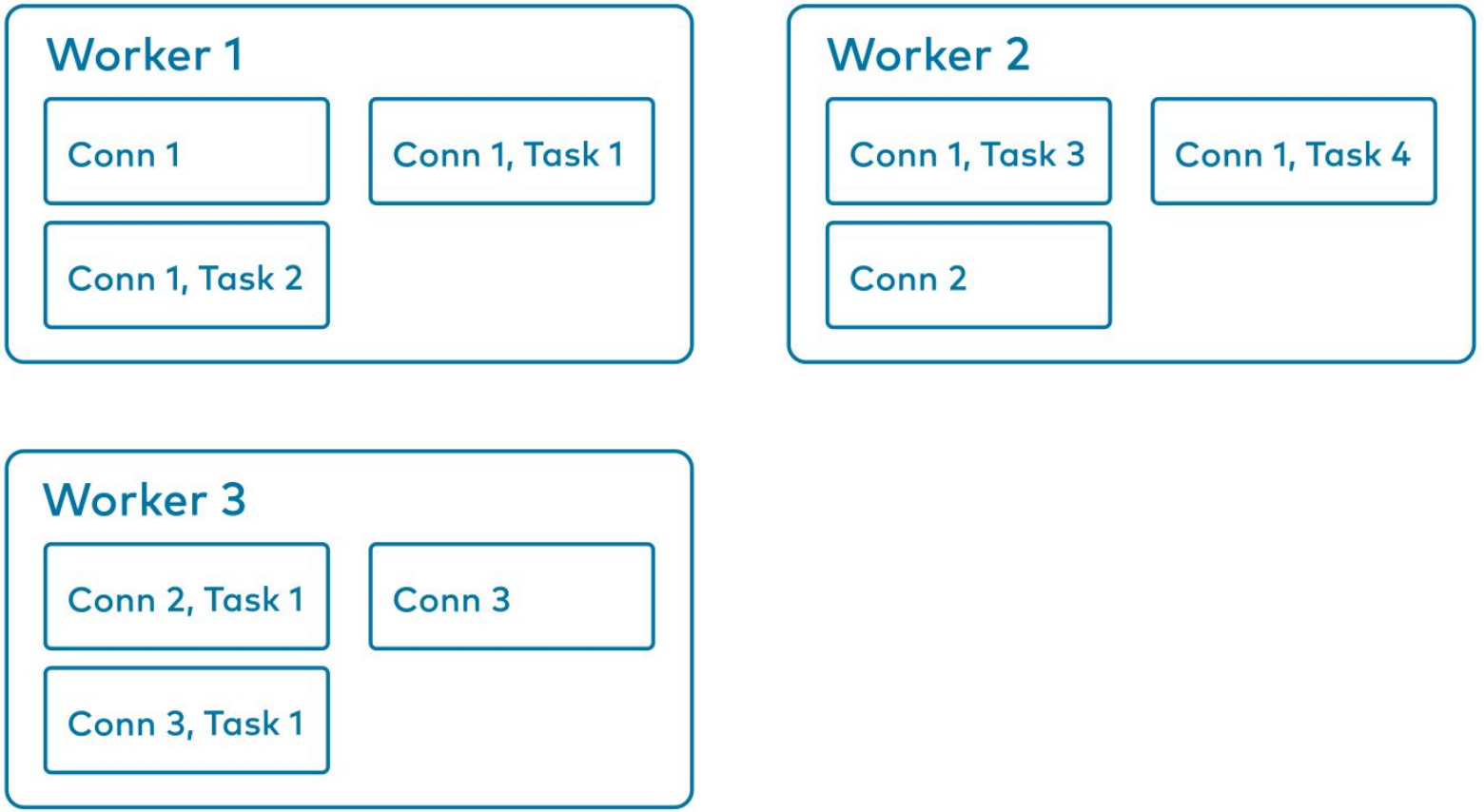
Worker Failure Example



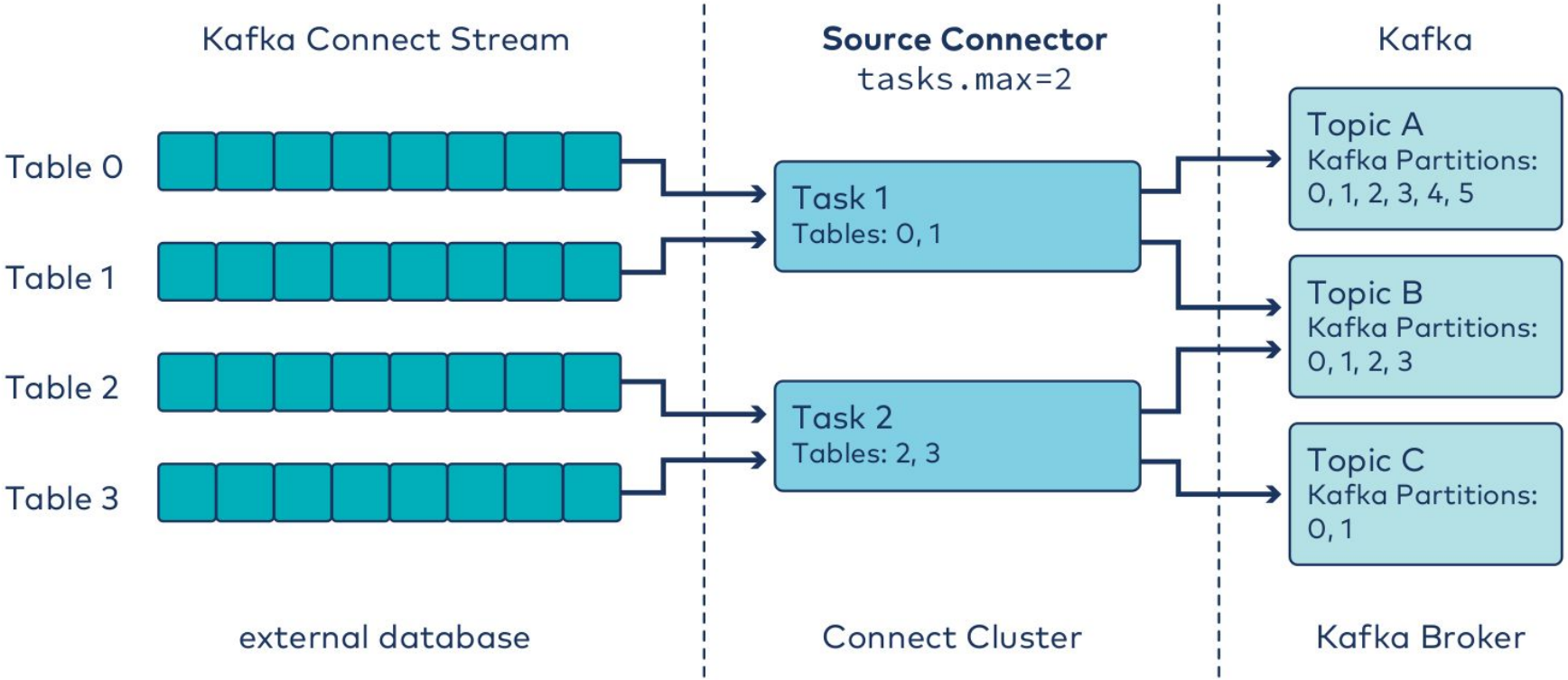
Worker Failure Example



Worker Failure Example



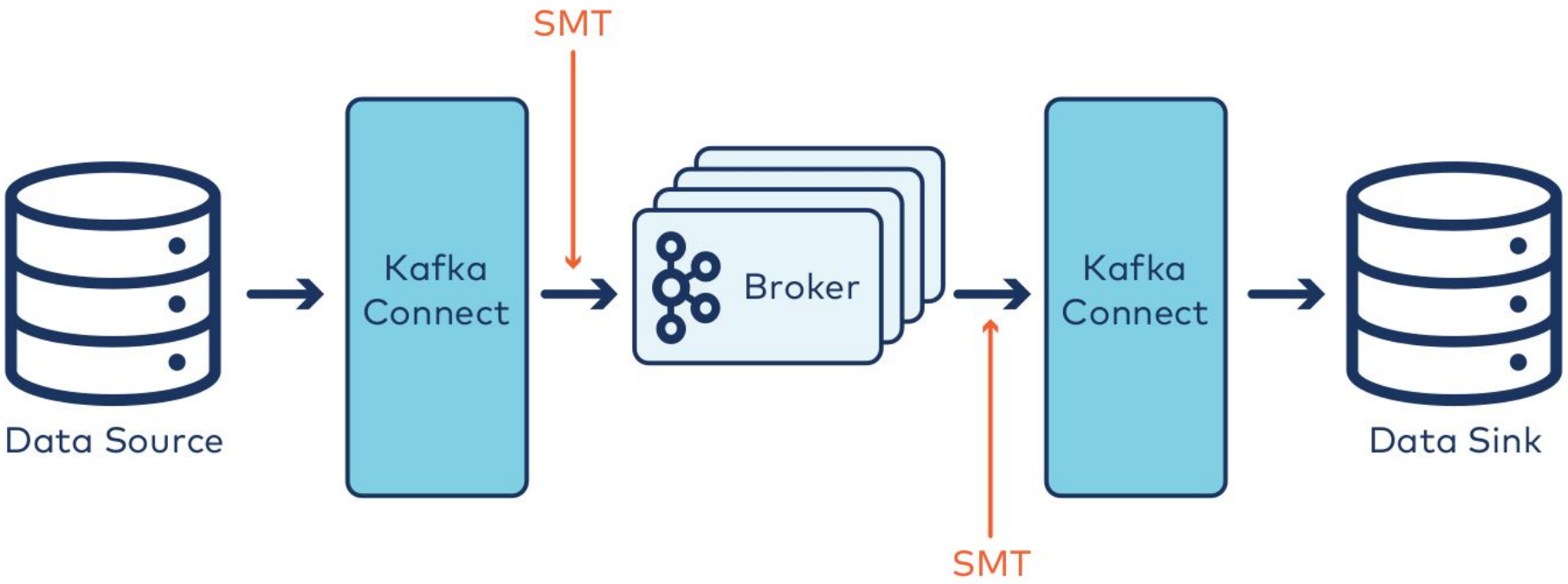
Source Connector Example



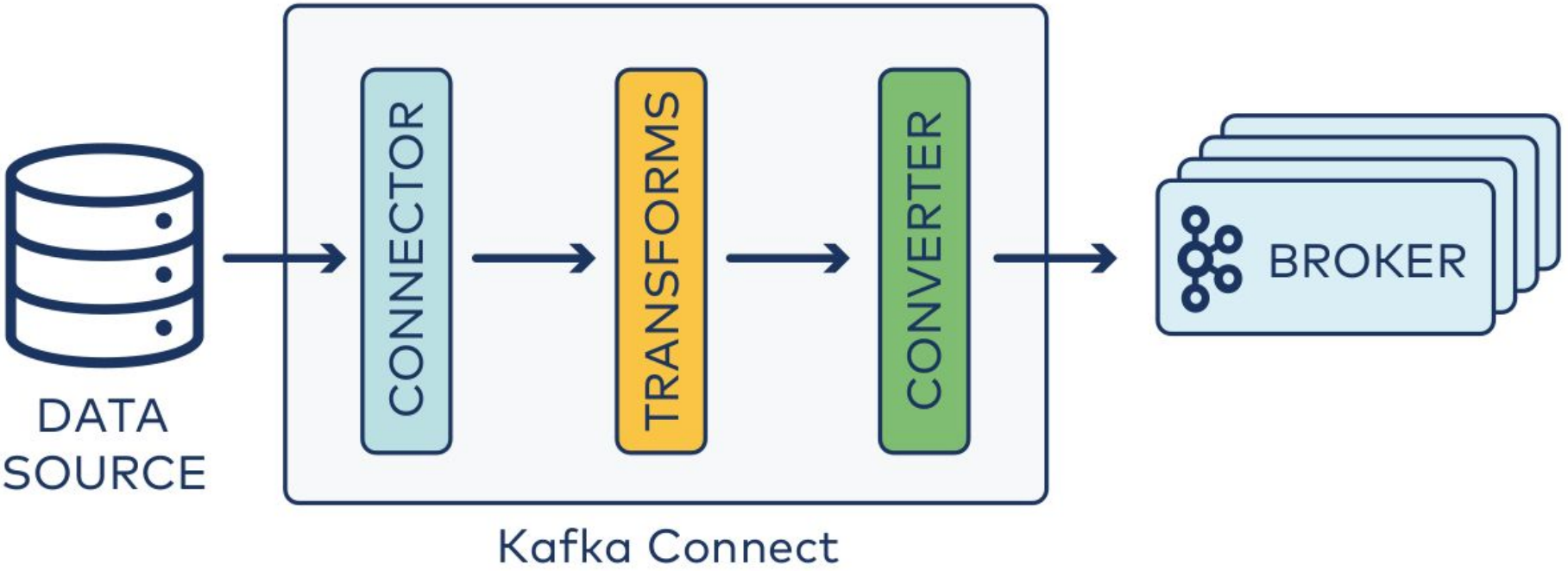
Important Properties

Name	Description
bootstrap.servers	La liste des host:port des **Brokers** utilisés pour la connection initiale
group.id	ID du groupe dont le **Connector** fait parti
heartbeat.interval.ms	Intervalle entre chaque **Heartbeat** envoyé par le **Consumer**. Default : 3sec
session.timeout.ms	Durée pendant laquelle un **Consumer** doit envoyer un **Hearbeat** pour ne pas être considéré comme “dead”. Default : 10sec ⚠️ La valeur par défaut est différente des **Consumers**
topic.creation.enable	Indique si le **Connector** est autorisé à créer des **Topics**

Simple Message Transforms (SMT)



Simple Message Transforms (SMT)



Simple Message Transforms (SMT)

Transform	Description
InsertField	Insère un nouveau champ depuis une valeur statique ou la valeur d'un champ du **Message** en input
ReplaceField	Renomme un champ
MaskField	Remplace le champ par une valeur vide (empty string, 0, etc)
ValueToKey	Remplace la **Key** par une nouvelle **Key** formée d'un ensemble de champs issus du **Message** en input
HoistField	Wrap le champ dans une Struct ou une Map
ExtractField	Extrait un champ d'une Struct ou d'une Map dans un champ dédié

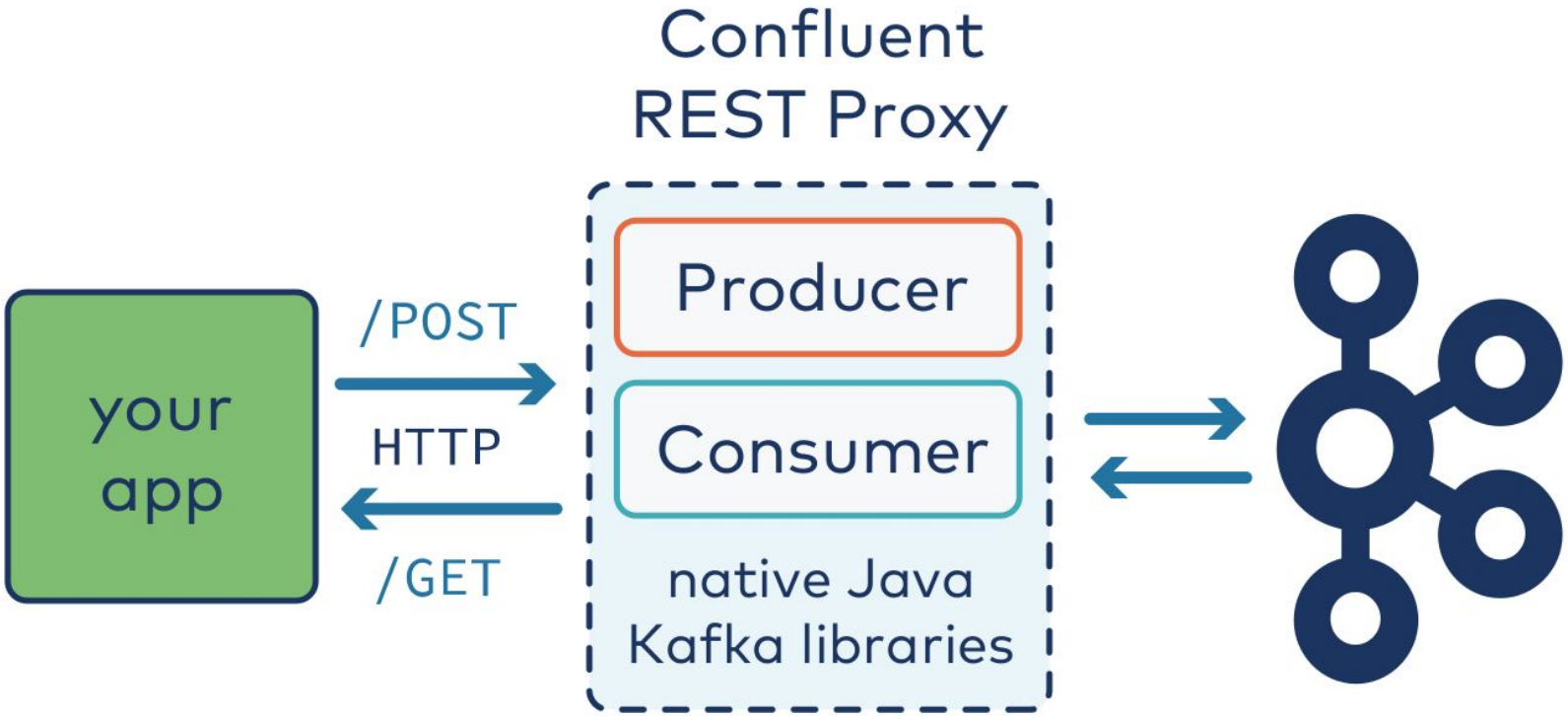
Plus d'informations et d'opération de **Transform** sur https://docs.confluent.io/platform/current/connect/transforms/overview.html

JDBC Source Connector Config Example

```
{
 "name": "Driver-Connector",
 "config": {
   "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
   "connection.url": "jdbc:postgresql://postgres:5432/postgres",
   "connection.user": "postgres",
   "table.whitelist": "driver",
   "topic.prefix": "",
   "mode": "timestamp+incrementing",
   "incrementing.column.name": "id",
   "timestamp.column.name": "timestamp",
   "table.types": "TABLE",
   "numeric.mapping": "best_fit"
 }
}
```

REST Proxy

REST Proxy


Confluent
REST Proxy
your
app
/POST
HTTP
/GET
Producer
Consumer
native Java
Kafka libraries

REST Proxy

- Développé par Confluent (Confluent Community License)
- Expose des API REST pour produire et consommer des **Messages**
- Utilise les API Java **Producer** et **Consumer** et **Admin**
- API référence : https://docs.confluent.io/current/kafka-rest/api.html
- Compatible avec le Schema Registry (Avro, Protobuf, JSON, binary, etc)
- Utile pour les applications développées dans des langages non supportés par les clients Kafka officiels

Python Producer example

```
import requests
import json

url = "http://restproxy:8082/topics/my_topic"
headers = {"Content-Type": "application/vnd.kafka.json.v2+json"}
# Create one or more messages
payload = {"records":
   [{
       "key": "firstkey",
       "value": "firstvalue"
   }]}
# Send the message
r = requests.post(url, data=json.dumps(payload), headers=headers)
if r.status_code ≠ 200:
   print "Status Code: " + str(r.status_code)
   print r.text
```

Python Consumer

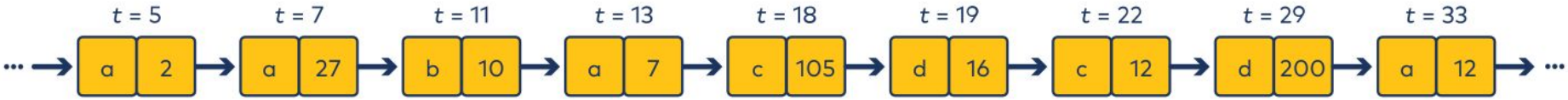
```
import requests
import json
import sys

FORMAT = "application/vnd.kafka.v2+json"
POST_HEADERS = {"Content-Type": FORMAT}
GET_HEADERS = {"Accept": FORMAT}

base_uri = create_consumer_instance("group1", "my_consumer")
subscribe_to_topic(base_uri, "hello_world_topic")
consume_messages(base_uri)
delete_consumer(base_uri)

def create_consumer_instance(group_name, instance_name):
   url = f'http://rest-proxy:8082/consumers/{group_name}'
   payload = {
       "name": instance_name,
       "format": "json"
   }
   r = requests.post(url, data=json.dumps(payload), headers=POST_HEADERS)

   if r.status_code ≠ 200:
       print ("Status Code: " + str(r.status_code))

   return r.json()["base_uri"]
```

Kafka Streams

C’est quoi Kafka Streams ?

- **Kafka Streams** est une API Java au même titre que les API **Producer** et **Consumer**
- Exactly Once processing semantics
- Highly scalable, fault-tolerant
- Les **Streams** sont immutables et infinis
    - Lorsque l’on effectue une opération de filtering par exemple, c’est un nouveau **Stream** qui est créé
- Les **Tables** sont mutables et finis

A Sample Stream

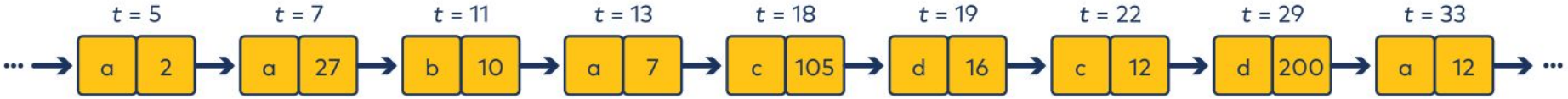


- Les événements sont des paires de **Key**/**Value** (la **Key** est aussi optionnelle avec **Kafka Streams**)
- Nos **Events** proviennent des **Topics**
- La notion de **Temps** est importante (plus que les **Offsets**)

Stateless Operations

On peut effectuer des opération Stateless sur les streams, comme par exemple une opération de filtrage.

Input stream :

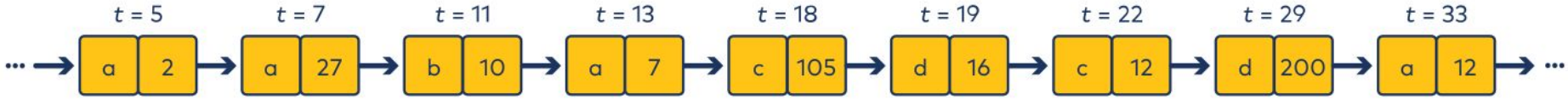


Output stream si l'on ne garde que les **Events** qui ont une **Value** supérieure à 50 :

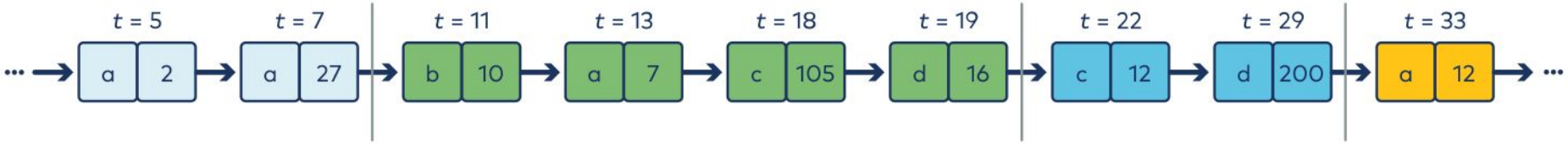
Windowing Operations

On peut effectuer des opérations de Windowing sur les streams, comme par exemple regrouper par fenêtre de temps.

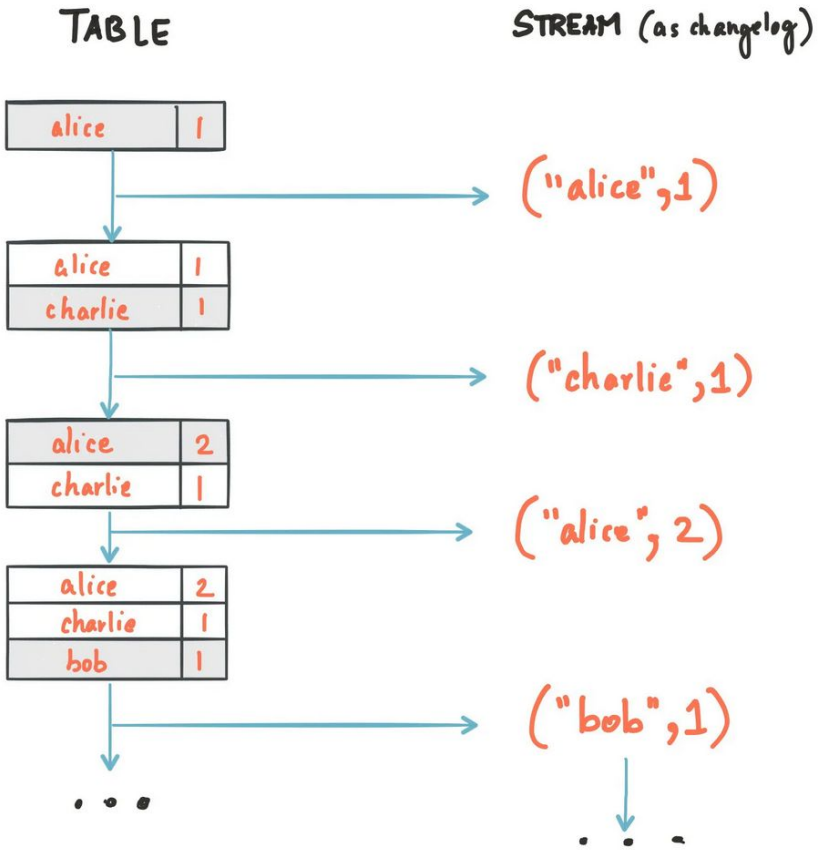
Input stream :



Output stream si fait des windows de 10 :



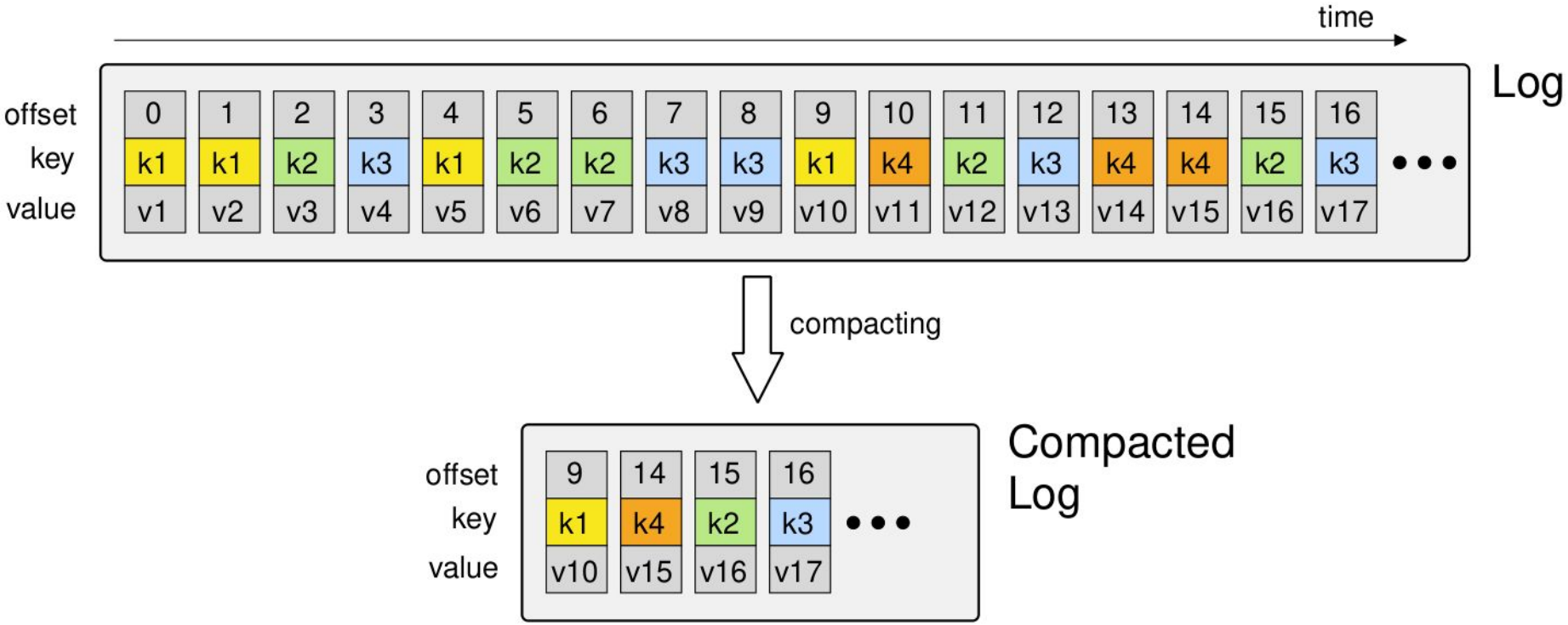
Duality Stream - Table



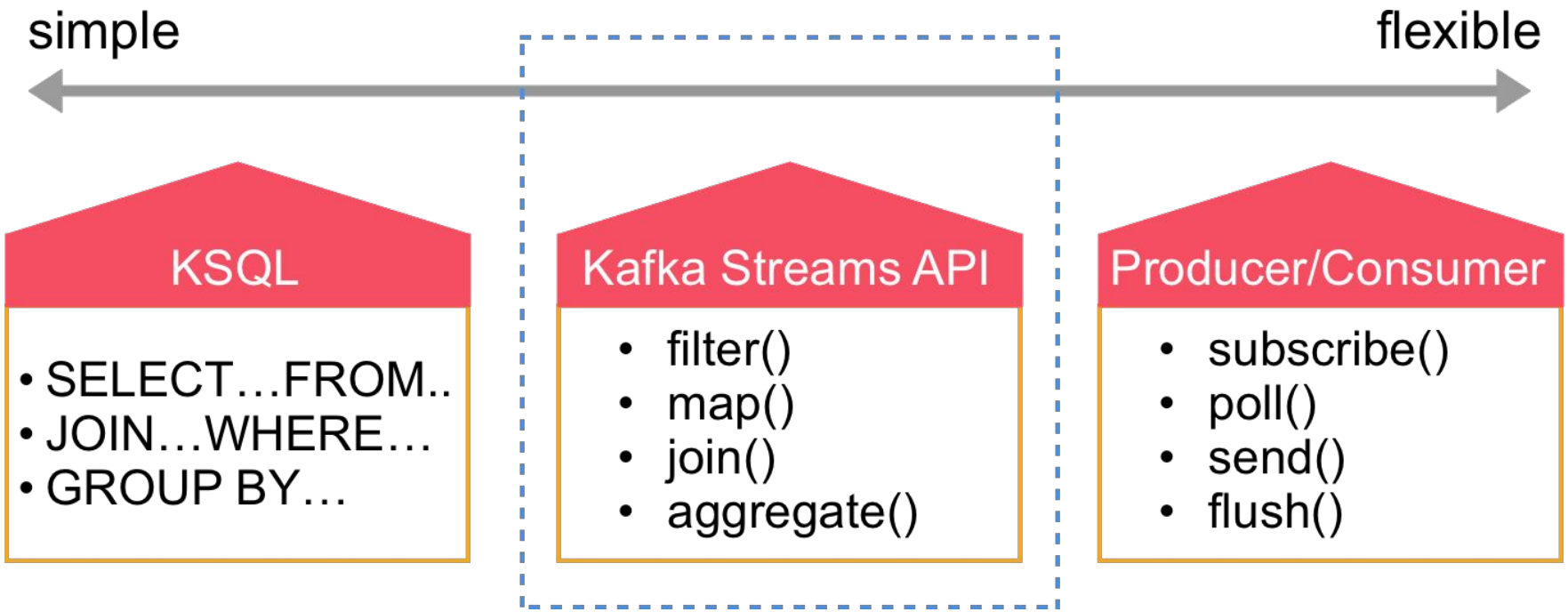
Streams vs Tables

	Stream	Table
Use case	Working with events in time/history	Storing status/state
Exemple 1	Driver positions	Driver profiles
Exemple 2	Order status changes	Orders, Customers

Tables - Compacted Topics



Streams vs ksqlDB vs Consumer/Producer



Processors

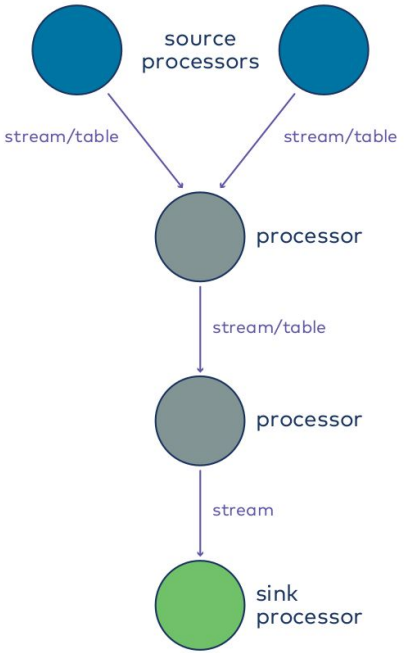
- Un **Stream Processor** est une opération sur un **Stream** ou une **Table**
  - Ex : filtrer un **Stream**
- Special **Processors** :
  - **Source Processor** :
    - Lit les Events depuis un **Topic** existant
    - Génère un KStream ou une KTable
  - **Sink Processor** :
    - Prend un KStream
    - Produit chaque **Event** du **Stream** vers un **Topic**

Processor Topology

Les **Processors** sont assemblés pour former une **Processor Topology**.

Les **Topologies** peuvent être représentés sous forme de **DAG** (Directed Acyclic Graph) :

- Les noeuds représentent les **Processors**
- Les flèches représentent les **Streams** et **Tables**



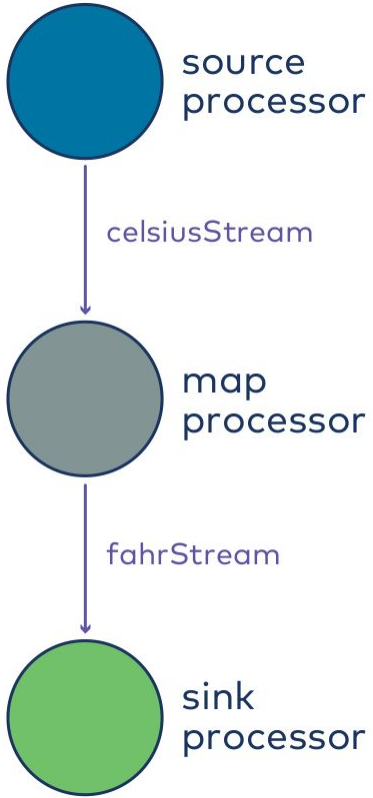
Processor Topology Example 1

Kafka cluster has 1 **Topic** :

- temp_readings
  - keys : postal codes
  - values : temperatures in degrees C

Topology :

- **Source processor** creates KStream celsiusStream from temp_readings
- **Processor** maps each temperature in degrees C to
  - Creates KStream fahrStream
- **Sink Processor** writes fahrStream to **Kafka** topic temp_readings_fahr



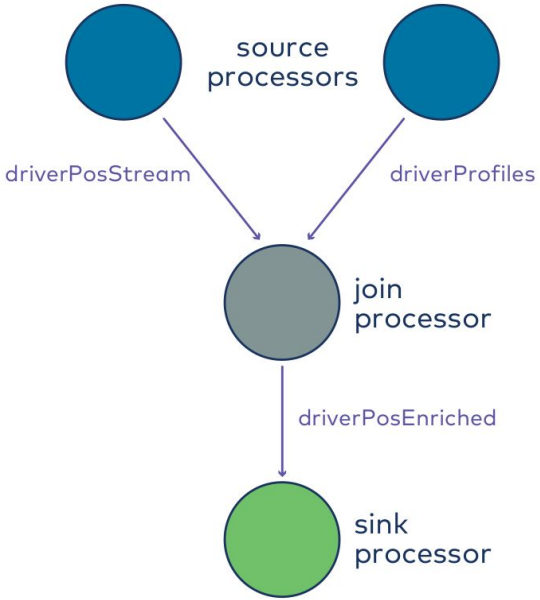
Processor Topology Example 2

Kafka cluster has **Topics** :

- driver_profiles
- driver_positions

Topology :

- **Source processors** create :
  - KStream driverPosStream from driver_positions
  - KTable driverProfiles from driver_profiles
- **Processor** join driverPosStream with driverProfiles
  - Creates KStream driverPosEnriched
- **Sink Processor** writes driverPosEnriched to **Kafka** topic driver_positions_enriched



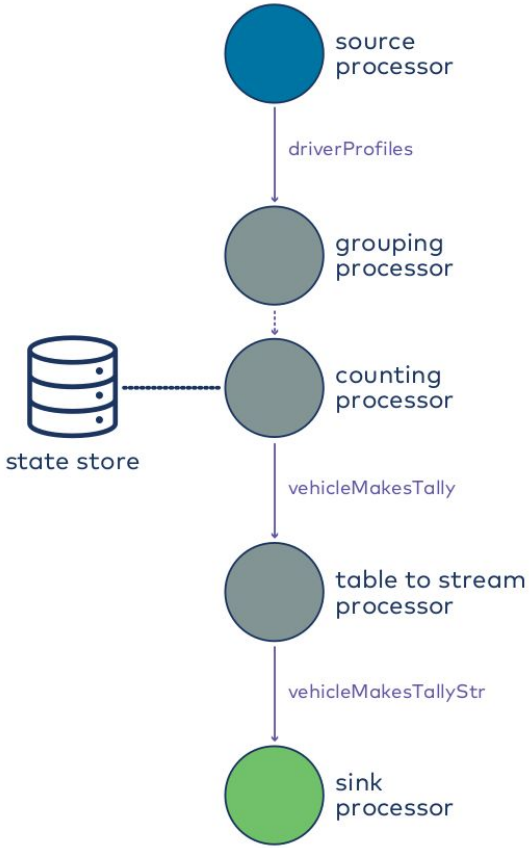
Processor Topology Example 3

Kafka cluster has **Topic** :

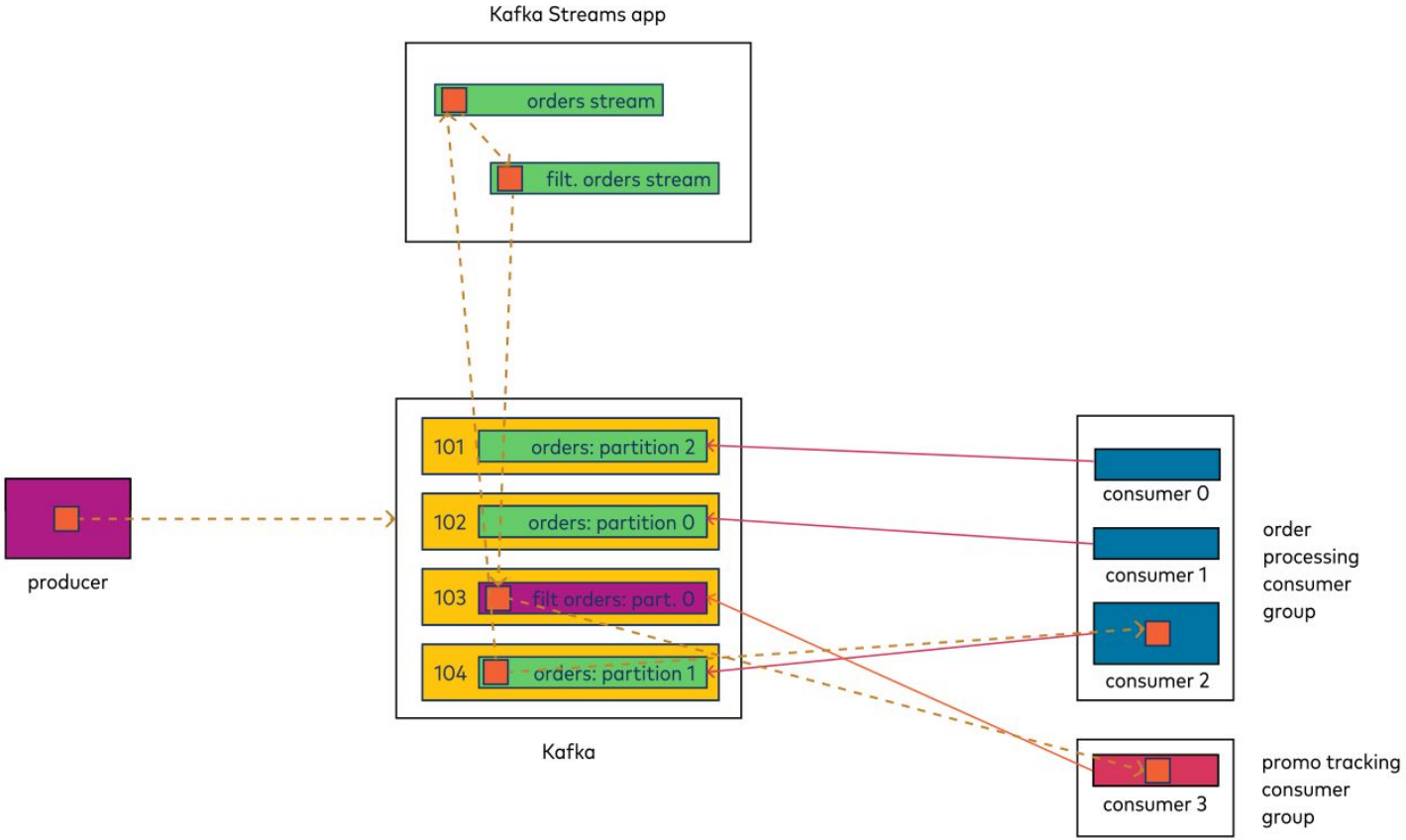
- driver_profiles

Topology :

- **Source processors** creates :
  - KTable driverProfiles from driver_profiles
- **Processors** group driverProfiles by make of vehicle and count number in each group
  - Creates KTable vehicleMakesTally
- **Processor** generates equivalent KStream vehicleMakesTallyStr from vehicleMakesTally
  - Creates KStream vehicleMakesTallyStr
- **Sink Processor** writes vehicleMakesTallyStr to **Kafka** topic vehicle_makes_tally_topic



Data Flow with Kafka Streams Apps



Kafka Streams Code Example

Kafka Streams DSL : Configuration

```
public static Properties getConfig() {
   Properties config = new Properties();
   // Give the application a name, which must be unique in the cluster
   config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-streams-example");

   // Connect to cluster
   config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");

   // Specify default (de)serializers for record keys and for record values.
   config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.Integer().getClass());
   config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
   return config;
}
```

Kafka Streams Code Example

Kafka Streams DSL : Stateless Topology

```
public static Topology getTopology() {
    StreamsBuilder builder = new StreamsBuilder();

    // Source processor: get stream from Kafka topic
    KStream<Integer, String> delimTransStream = builder.stream("delim_transactions_topic");

    // Internal processor: break up the stream
    KStream<Integer, String> indTransStream = delimTransStream.flatMapValues(value → split(value, "|"));

    // Sink processor: new stream back to new Kafka topic
    indTransStream.to("individual_transactions_topic");

    // Generate and return topology
    Topology result = builder.build();

    return result;
}
```

Kafka Streams Code Example

Kafka Streams DSL : Main Program

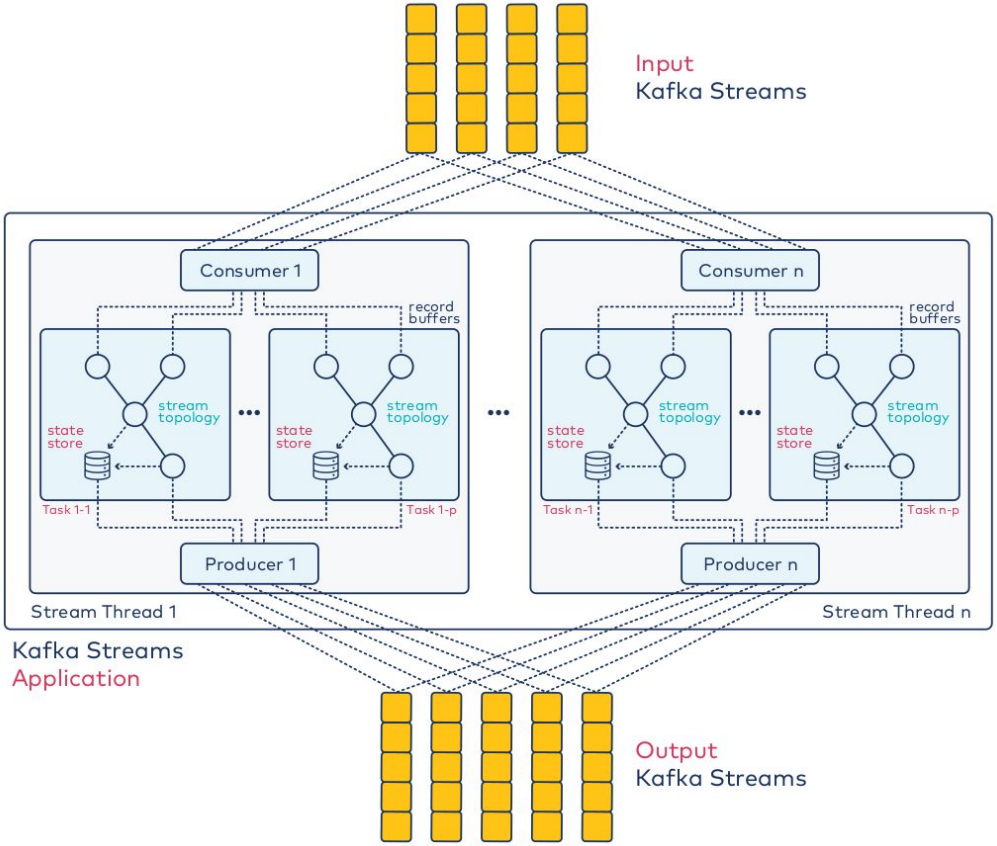
```
public static void main(String[] args) {
    // Create a streams application based on config & topology defined already
    KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());

    // Run the Streams application via `start()`
    streams.start();

    // Stop the application gracefully
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

Kafka Streams Code Example

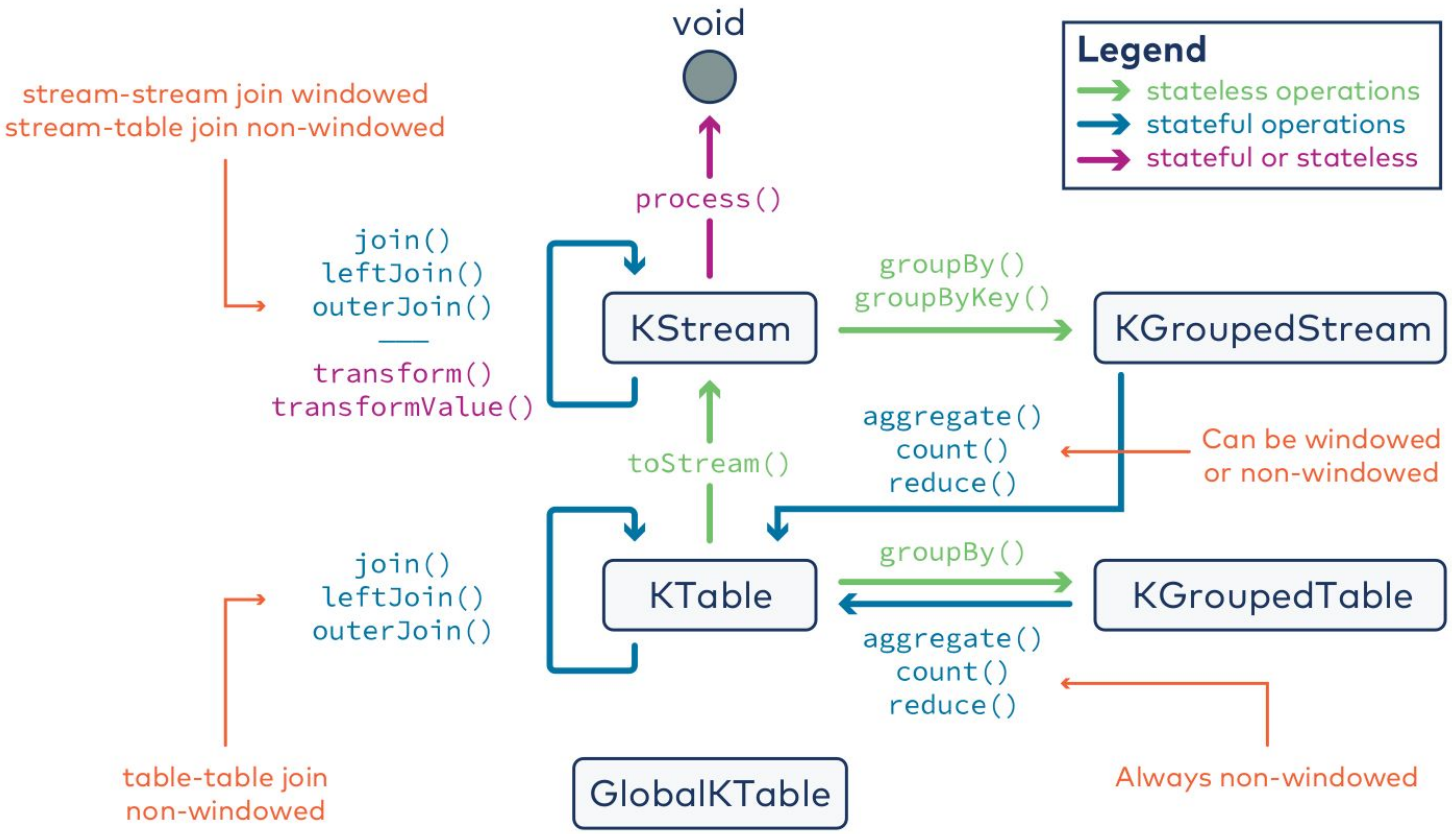
Kafka Streams DSL : Stateful Topology

```
public static Topology getTopology() {
    StreamsBuilder builder = new StreamsBuilder();

    // Source processor: get stream from Kafka topic
    KStream<Integer, String> delimTransStream = builder.stream("delim_transactions_topic");

    // Internal processor: break up the stream
    KStream<Integer, String> indTransStream = delimTransStream.flatMapValues(value → split(value, "|"));

    // Group transactions by account number and count
    KTable<Integer, Long> transByAcctTally = indTransStream.groupByKey()
        .count();

    // Sink processor: convert table to stream, then write to new Kafka topic
    transByAcctTally.toStream()
        .to("acct_activity_tally_topic", Produced.with(Serdes.Integer(), Serdes.Long()));

    // Generate and return topology
    Topology result = builder.build();
    return result;
}
```

Stateful Stores



Stateful vs Stateless Operations



Lab

ksqlDB

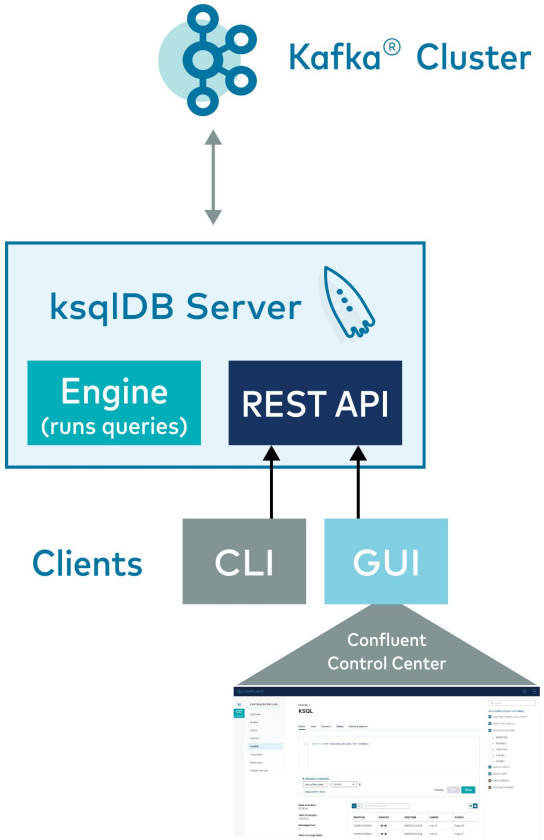
C’est quoi ksqlDB ?

ksqlDB permet de développer des real-time stream processing apps avec du SQL uniquement :

```
ksql> CREATE STREAM vip_actions AS
      SELECT userid, page, action
      FROM clickstream c
      LEFT JOIN users u ON c.userid = u.user_id
      WHERE u.level = 'Platinum';
```

C'est quoi ksqlDB ?

- **ksqlDB** exploite l'API **Kafka Streams**
- Open source (Confluent Community License)
- ksqlDB est scalable, fault-tolerant et real-time
- Connect via Control Center UI, CLI ou REST
- Extensible via des UDF (User Defined Functions) en Java
- Use cases :
  - Streaming ETL
  - Anomaly detection
  - Event monitoring



ksqlDB

Remember Kafka Streams...

```
public static void main(String[] args) {
   // Create a streams application based on config & topology defined
already
   KafkaStreams streams = new KafkaStreams(getTopology(), getConfig());

   // Run the Streams application via `start()`
   streams.start();

   // Stop the application gracefully
   Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}

public static Properties getConfig() {
   Properties config = new Properties();
   // Give the application a name, which must be unique in the cluster
   config.put(StreamsConfig.APPLICATION_ID_CONFIG,
"simple-streams-example");

   // Connect to cluster
   config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker-1:9092");

   // Specify default (de)serializers for record keys and for record
values.
   config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.Integer().getClass());
   config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
   return config;
}
```

```
public static Topology getTopology() {
   StreamsBuilder builder = new StreamsBuilder();

   // Source processor: get stream from Kafka topic
   KStream<Integer, String> indTransStream =
builder.stream("transactions_topic");

   // Group transactions by account number and count
   KTable<Integer, Long> transByAcctTally = indTransStream.groupByKey()
       .count();

   // Sink processor: convert table to stream, then write to new Kafka
topic
   transByAcctTally.toStream()
       .to("acct_activity_tally_topic",
           Produced.with(Serdes.Integer(), Serdes.Long()));

   // Generate and return topology
   Topology result = builder.build();
   return result;
}
```

Syntax - Create Stream

Pour commencer, il faut créer un **Stream** de notre **Topic** source

```
CREATE STREAM ind_trans_stream (acct_id INT KEY, amount DOUBLE, details VARCHAR)
   WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'transactions_topic');
```

Un STREAM correspond à un KStream et une TABLE à une KTable

CREATE STREAM ... WITH permet de faire la même chose que StreamsBuilder.stream()

Syntax - SQL Query

Ensuite on peut commencer à exécuter des requêtes SQL :

```
SELECT acct_id, count(*)
FROM ind_trans_stream
GROUP BY acct_id;
```

On peut également stocker le résultat de cette requête dans une TABLE

```
CREATE TABLE trans_by_acct_tally AS
SELECT acct_id, count(*)
FROM ind_trans_stream
GROUP BY acct_id;
```

Syntax - SQL Query

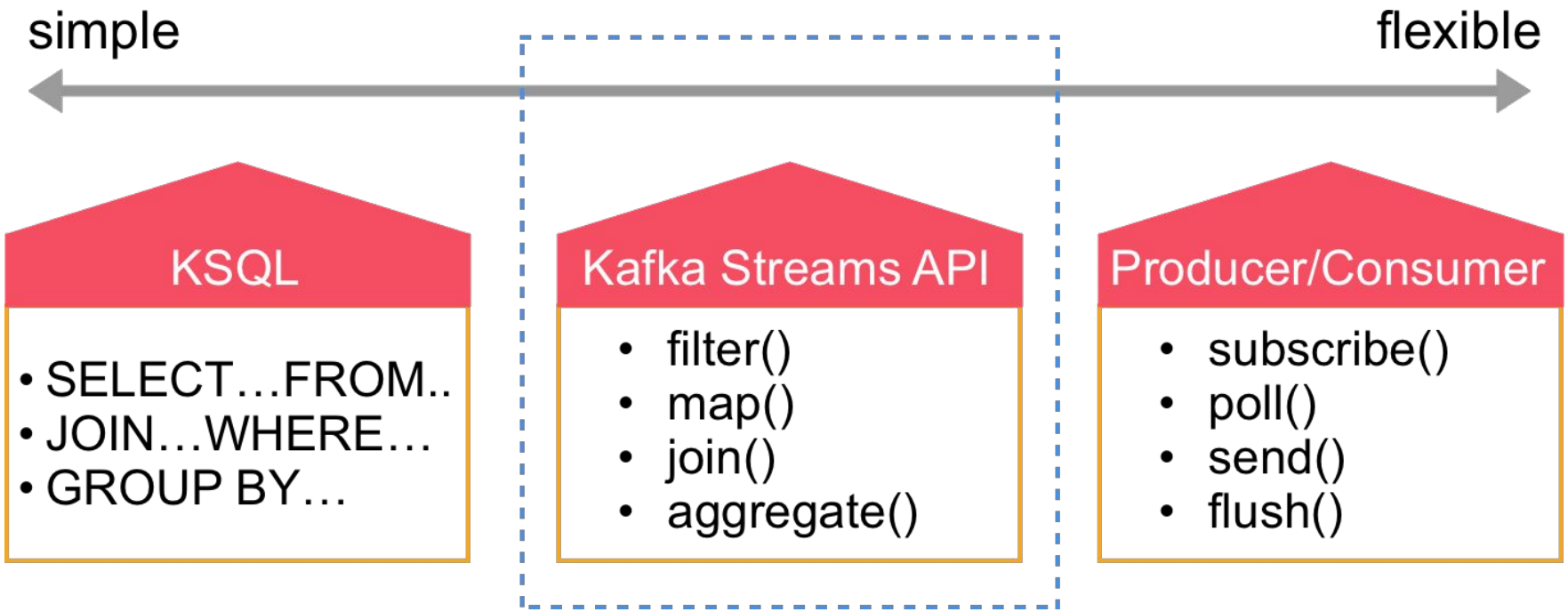
Voici un exemple de filtering

```
CREATE STREAM big_transaction_amounts AS
SELECT acct_id, amount
FROM ind_trans_stream
WHERE amount > 100
```

Et un exemple de filtering + aggregation

```
CREATE TABLE trans_by_acct_tally AS
SELECT acct_id, count(*)
FROM ind_trans_stream
WHERE amount > 100
GROUP BY acct_id;
```

Streams vs ksqlDB vs Consumer/Producer



SQL

- Conçu dans le but d'être accessible pour les personnes ayant fait du SQL
- Standard TABLE, FROM clauses
- Standard aggregation with GROUP BY
- Standard filtering with WHERE and HAVING
- Many popular scalar functions (CONCAT, SUBSTRING, FLOOR, ...)
- Windowing capabilities
- Join capabilities
- And more on full ksqlDB documentation

ksqlDB

Persistent vs Non-Persistent Queries

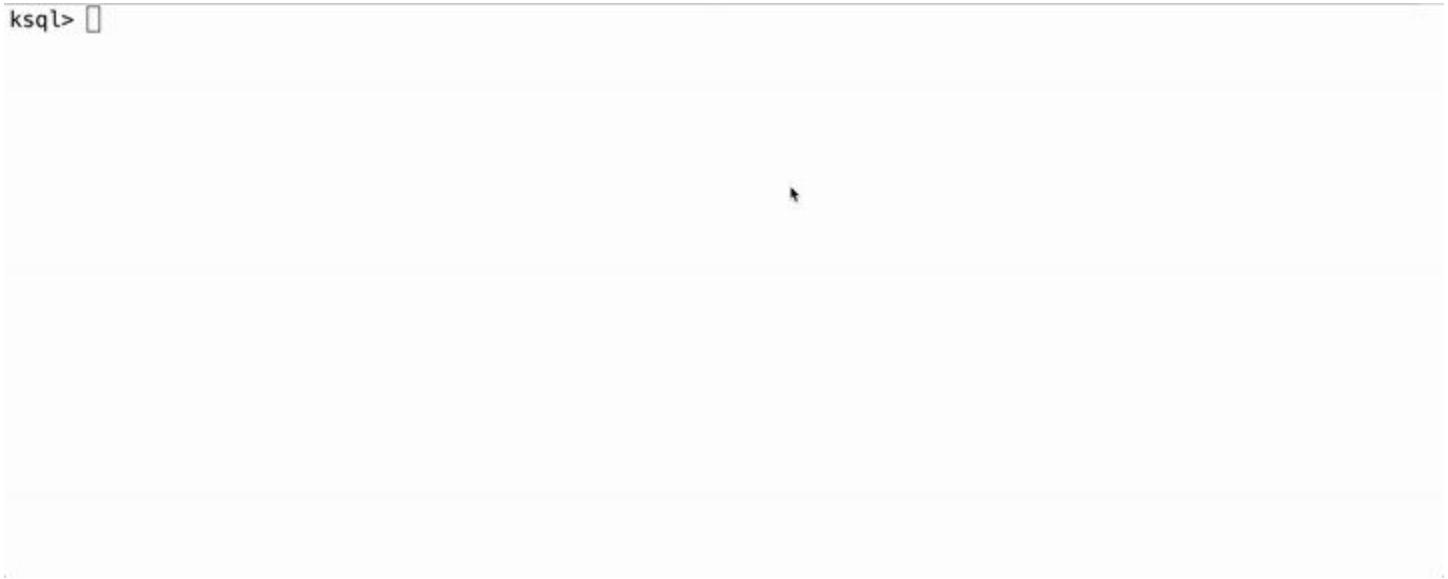
Non-Persistent Queries	Persistent Queries
Correspond aux résultats visibles dans le CLI ou dans Confluent Control Center	Les résultats sont persistés dans un STREAM ou une TABLE
Les résultats ne sont pas persistés	Créés grâce à : • CREATE STREAM … AS SELECT • CREATE TABLE … AS SELECT
SELECT …	STREAM ou TABLE : • Peuvent être “queried” depuis une autre requête • Sont stockés dans un **Topic** du même nom

Push vs Pull Queries

Push Queries	Pull Queries
All value changes	Point in time value
Ne se termine jamais	Se termine dès que le résultat est retourné
Disponible avec les STREAM et les TABLE	Disponible uniquement avec les TABLE ayant un “materialized state”
	Nécessite la clause EMIT CHANGES

ksqlDB

Push Queries



ksqlDB

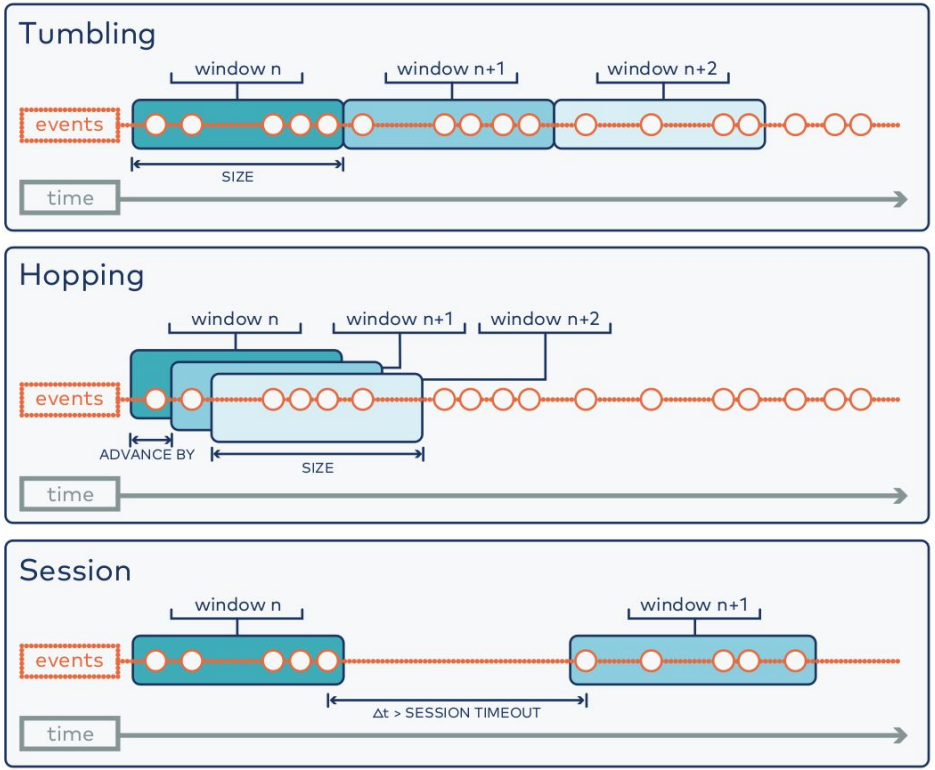
Pull Queries

```
ksql> SELECT TIMESTAMPTOSTRING(WINDOWSTART,'yyyy-MM-dd HH:mm:ss','Europe/London') AS WINDOW_START,
>        TIMESTAMPTOSTRING(WINDOWEND,'HH:mm:ss','Europe/London') AS WINDOW_END,
>        MAKE,
>        ORDER_COUNT,
>        TOTAL_ORDER_VALUE
>  FROM ORDERS_PER_HOUR_BY_MAKE
>  WHERE MAKE = 'Funk Inc';
+-------------------+-------------------+-------------------+-------------------+-------------------+
|WINDOW_START       |WINDOW_END         |MAKE               |ORDER_COUNT        |TOTAL_ORDER_VALUE  |
+-------------------+-------------------+-------------------+-------------------+-------------------+
|2021-02-22 10:00:00 |11:00:00           |Funk Inc           |7                  |297.92             |
|2021-02-22 11:00:00 |12:00:00           |Funk Inc           |120                |4543.28            |
|2021-02-22 12:00:00 |13:00:00           |Funk Inc           |32                 |1375.92            |
|2021-02-22 13:00:00 |14:00:00           |Funk Inc           |131                |5107.76            |
|2021-02-22 14:00:00 |15:00:00           |Funk Inc           |112                |4457.04            |
|2021-02-22 15:00:00 |16:00:00           |Funk Inc           |104                |4088.56            |
|2021-02-22 16:00:00 |17:00:00           |Funk Inc           |110                |4296.32            |
|2021-02-22 17:00:00 |18:00:00           |Funk Inc           |79                 |2857.68            |
Query terminated
ksql>
```

Windowing

- **Tumbling** : on définit une durée fixe de fenêtre qui s'enchaine les unes après les autres
- **Hopping** : on définit une durée fixe de fenêtre ainsi qu'un "pas" d'avancement de chaque fenêtre. Contrairement au Windowing **Tumbling**, les fenêtres peuvent se croiser avec du **Hopping**
- **Session** : on définit une durée de session timeout. La durée d'une Window **Session** est donc variable

ℹ️ Le Windowing **ksqlDB** est similaire au Windowing **Kafka Streams**



ksqlDB

Create Stream from existing Topic example

```
CREATE STREAM NETWORK_TRAFFIC_NESTED
(
  timestamp BIGINT,
  layers STRUCT<
     frame STRUCT<
        frame_frame_time VARCHAR,
        frame_frame_protocols VARCHAR
     >,
     eth STRUCT<
        eth_eth_src VARCHAR,
        eth_eth_dst VARCHAR
     >,
     ip STRUCT<
        ...
     >,
     tcp STRUCT<
        ...
     >,
     http STRUCT<
        ...
        http_http_response_line array<VARCHAR>,
        ...
     >
  >
)
WITH (KAFKA_TOPIC='network-traffic', TIMESTAMP='timestamp', VALUE_FORMAT='JSON');
```

ksqlDB

Create Stream from Stream example

```
CREATE STREAM NETWORK_TRAFFIC_FLAT
AS SELECT
   timestamp,
   layers→frame→frame_frame_protocols as frame_protocols,
   layers→frame→frame_frame_time as frame_time,
   layers→eth→eth_eth_src as eth_addr_source,
   layers→eth→eth_eth_dst as eth_addr_dest,
   layers→ip→ip_ip_src as ip_source,
   ...

   layers→tcp→tcp_tcp_stream as tcp_stream,
   layers→tcp→tcp_tcp_srcport as tcp_port_source,
   layers→tcp→tcp_tcp_dstport as tcp_port_dest,
   ...

   layers→http→http_http_host as http_host,
   layers→http→http_http_request_full_uri as http_request_full_uri,
   layers→http→text_http_request_method as http_request_method,
   ...
   layers→http→http_http_file_data as http_file_data
FROM NETWORK_TRAFFIC_NESTED;
```

ksqlDB

Create Table with Windowing example

```
-- Detect Slowloris attacks
CREATE TABLE potential_slowloris_attacks
AS SELECT ip_dest, count(*) as count_connection_reset
  FROM NETWORK_TRAFFIC_FLAT
  WINDOW TUMBLING (SIZE 60 SECONDS)
  WHERE tcp_flags_ack = '1' AND tcp_flags_reset = '1'
  GROUP BY ip_dest
  HAVING count(*) > 100;
```

ksqlDB

Lab

Confluent Cloud

Izivia

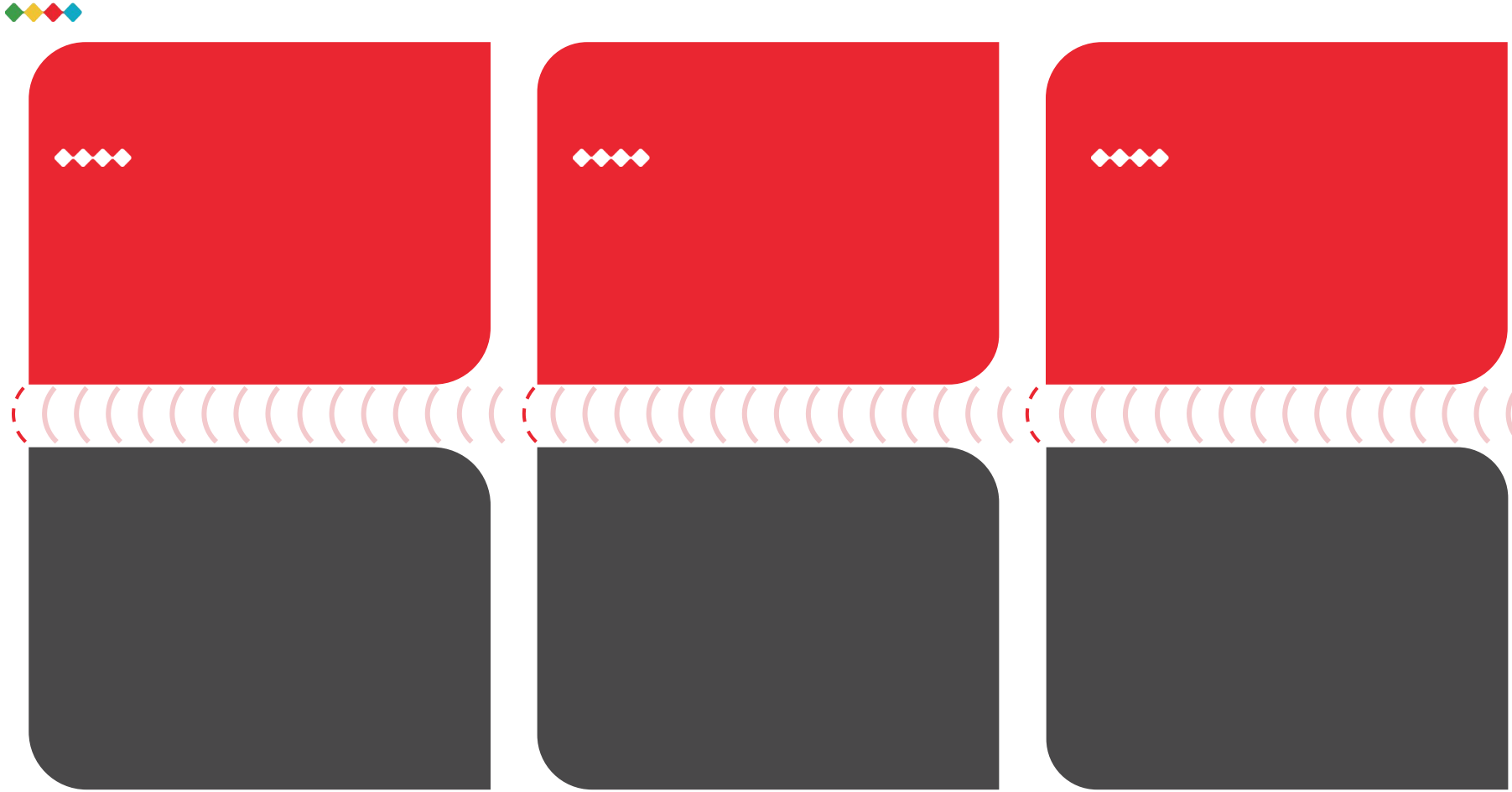
Conclusion

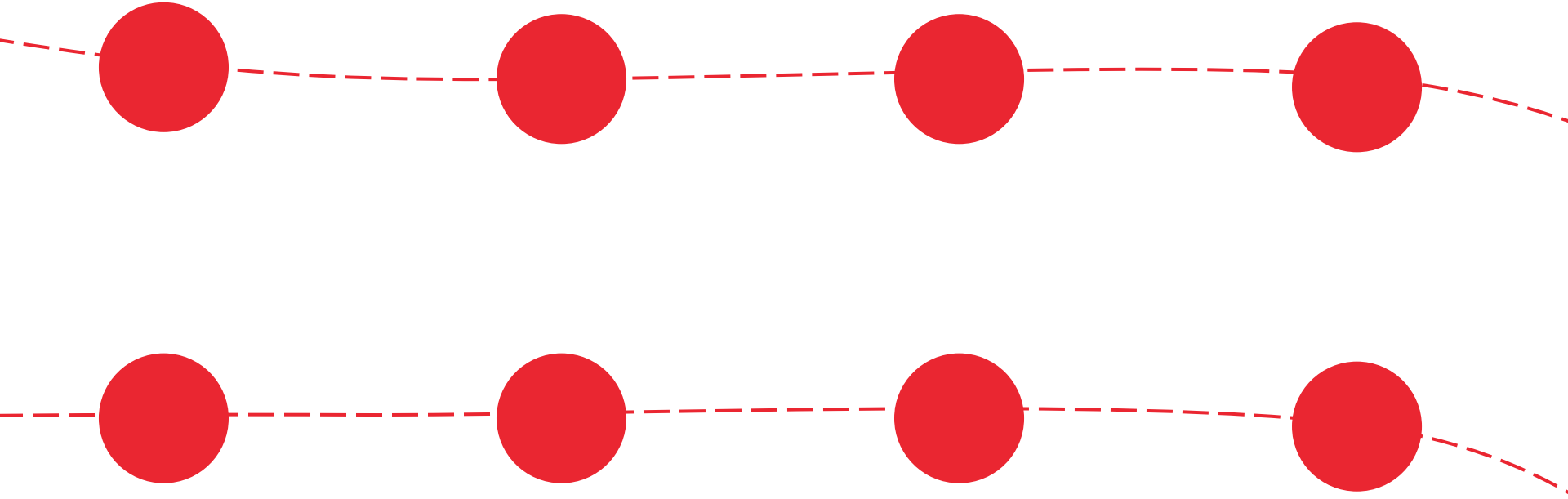
Conclusion

Vous êtes maintenant capable de...

- Décrire les principaux composant d'un cluster **Kafka** et d'expliquer leurs rôles
- Concevoir des **Topics** adapté aux besoins (partitions, retention, replication, batching, etc)
- Développer **Producers** et **Consumers** qui vont interagir avec **Kafka**
- Créer des **Schema** et de les faire évoluer grâce au **Schema Registry**
- Mettre en place des intégrations entre **Kafka** et des systèmes externes grâce à **Kafka Connect**
- Développer des "streaming apps" grâce à **Kafka Streams** et **ksqlDB**

Merci !








4SH

www.4sh.fr

linkedin.com/company/4sh-france/

@4sh_france

Bordeaux

6-8 avenue des satellites
33185 Le Haillan

+33 (0)9 63 28 62 73