

---

# Soutenance de PFE

Bejaoui Mohamed  
Naudan Quentin

27 Mars 2020



# Cahier des charges

Objectif	Cours associé	Status / Commentaire
Mise en œuvre d'une méthode d'apprentissage et son évaluation (cross val.)	App. aut.	Ok (random forest)
Mise en œuvre de concepts statistiques, modélisation probabiliste	Mod. stat.	Ok (processus de Hawkes, estimation MLE par optimisation)
Implémentation à partir d'un article scientifique		Ok
Richesse du problème et des données → possibilité de faire évoluer d'année en année les challenges		Ok (tweets, NLP)
Mise en œuvre d'un projet de développement logiciel (git, build, test)	Ing. appl. log.	
Intégration dans un framework logiciel avec une problématique de distribution du calcul et de passage à l'échelle	C++ Ing. appl. Log. Big Data ?	Ok (Kafka)
Implémentation d'un algorithme optimisé en C++. Compilation Cmake avec libs	C++	A faire (python non optimisé) Libs = Kafka, IPOPT
Déployer du code dans le Cloud (Docker/Kubernetes)	Ing. appl. log.	A faire. Contact chez Azure



## **Analyse de popularité des Tweets, déployée sur une architecture Apache Kafka**

- ① Présentation du sujet du projet
- ② Application d'une solution issue d'un article scientifique
- ③ Notions de Kafka utilité pour notre projet
- ④ Intégration de notre solution dans une architecture Kafka

# **I. Présentation du sujet du projet**

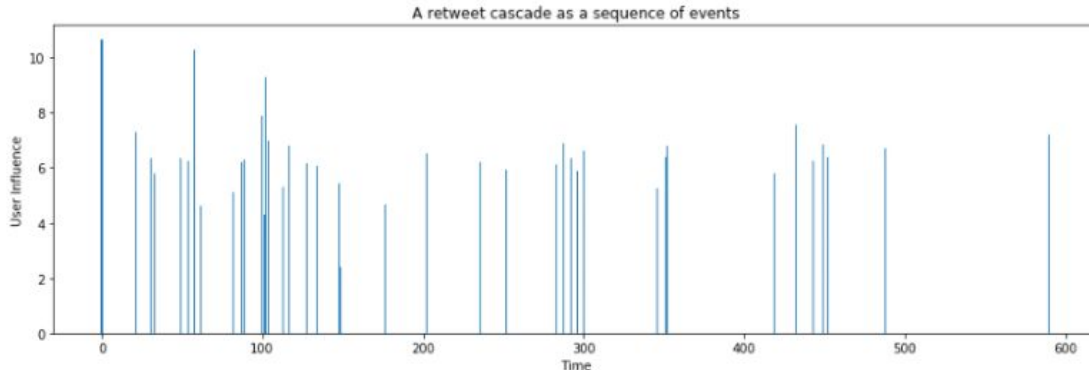
# Présentation

- Comprendre les principes de Kafka et l'évaluer sur des exemples simples
- Développer une source Kafka délivrant et traitant des tweets
- Déployer “à la main” le prototype sur un cluster de machines
- Article intitulé “Feature Driven and Point Process Approaches for Popularity Prediction” publié en Août 2016 par l'université d'Australie

## **II. Application d'une solution issue d'un article scientifique**



- Nous souhaitons un **modèle génératif** pour décrire une cascade de retweets, afin d'en prédire le nombre final.
- Nous utilisons un *self-exciting point process* de Hawkes (une classe de processus stochastique)





## Noyau d'un événement

- Un retweet est un événement à 2 paramètres : le couple (magnitude, time)
- Sa contribution temporelle à trigger un nouveau retweet est modélisé par  $\phi_m(\tau) = \kappa m^\beta (c + \tau)^{-(1+\theta)}$

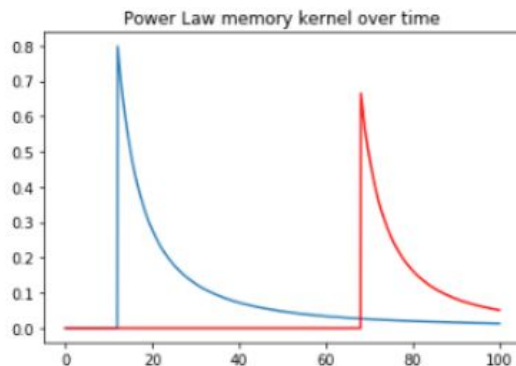


Figure:  $\kappa = 0.8, \beta = 0.6, c = 10, \theta = 0.8$

Bleu : [1000, 12]

Rouge : [750, 68]





- La contribution de tous les événements donne la fonction de taux d'arrivée de nouveaux retweets, notée :

$$\lambda(t) = \sum_{t_i < t} \phi_{m_i}(t - t_i)$$

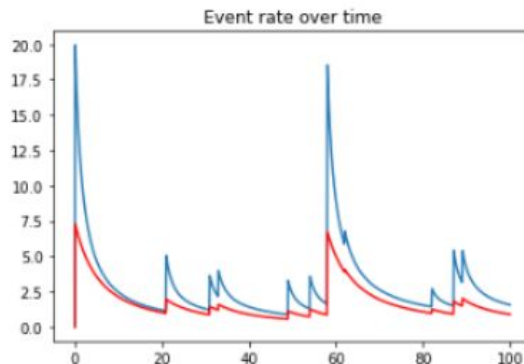


Figure: Bleu :  $K = 0.24$ ,  $\beta = 0.5$ ,  $c = 2$ ,  $\theta = 0.2$

Rouge :  $K = 0.8$ ,  $\beta = 0.6$ ,  $c = 10$ ,  $\theta = 0.8$



- Nous confrontons notre modèle aux données sur un temps d'observation  $T$ , en évaluant la **log-vraisemblance** :

$$L(\kappa, \beta, c, \theta) = \sum_{i=1}^n \log(\lambda(t_i)) - \int_{t=0}^T \lambda(\tau) \, d\tau$$

- Nous cherchons à maximiser cette fonction, nous utilisons une librairie open source : **Ipopt** (Interior Point OPTimizer)



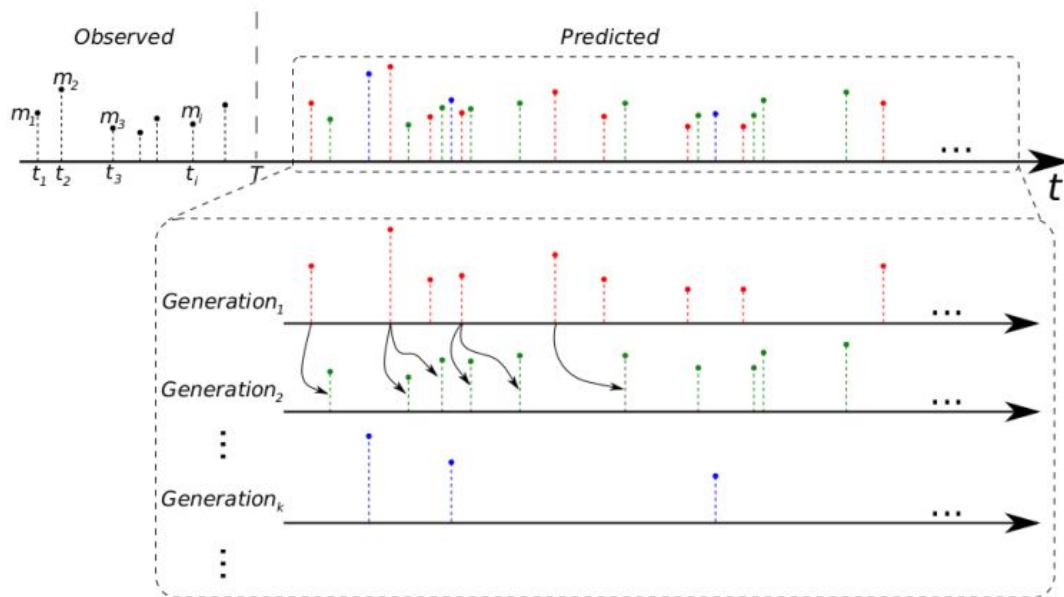
- Le **branching factor** est le nombre d'enfants attendus d'un seul événement : 
$$n^* = \int_{m=1}^{\infty} \int_{\tau=0}^{\infty} P(m) \phi_m(\tau) \, d\tau dm$$
avec la distribution d'influence commune à tous les utilisateurs Twitter :  $P(m) = (\alpha - 1)m^{-\alpha}$
- Le modèle ne prédit pas pour les cascades de tweets en régime super-critique  $n^* \geq 1$



- $n^* = \kappa \frac{\alpha - 1}{(\alpha - \beta - 1)\theta c^\theta}$  avec  $\theta > 0$  et  $0 < \beta < \alpha - 1$   
ce qui apporte une **contrainte d'optimisation** sur les variables.



- A partir d'un jeu de données, on note  $A_1$  le nombre d'événements enfantés par la 1ère génération de retweets (en rouge sur le graphe).





- $A_1 = \int_T^{\infty} \lambda(t) dt$
- Nous estimons chaque génération par la récurrence  
 $A_{i+1} = A_i * n^*$
- D'où la prédiction finale s'obtient, quand  $n^* < 1$ , par :  
$$N_{\infty} = n + \sum_{i=1}^{\infty} A_i = n + \frac{A_1}{n^* - 1}$$



- Nous ajoutons une couche de prédiction au modèle, car la modélisation mathématiques à elle-seule ne peut pas donner des résultats performants.
- Nous souhaitons réaliser l'apprentissage supervisé d'un **scaling factor**  $w$  pour améliorer la prédiction finale de retweets :  $\hat{N}_{\infty} = n + w \frac{A_1}{n^* - 1}$

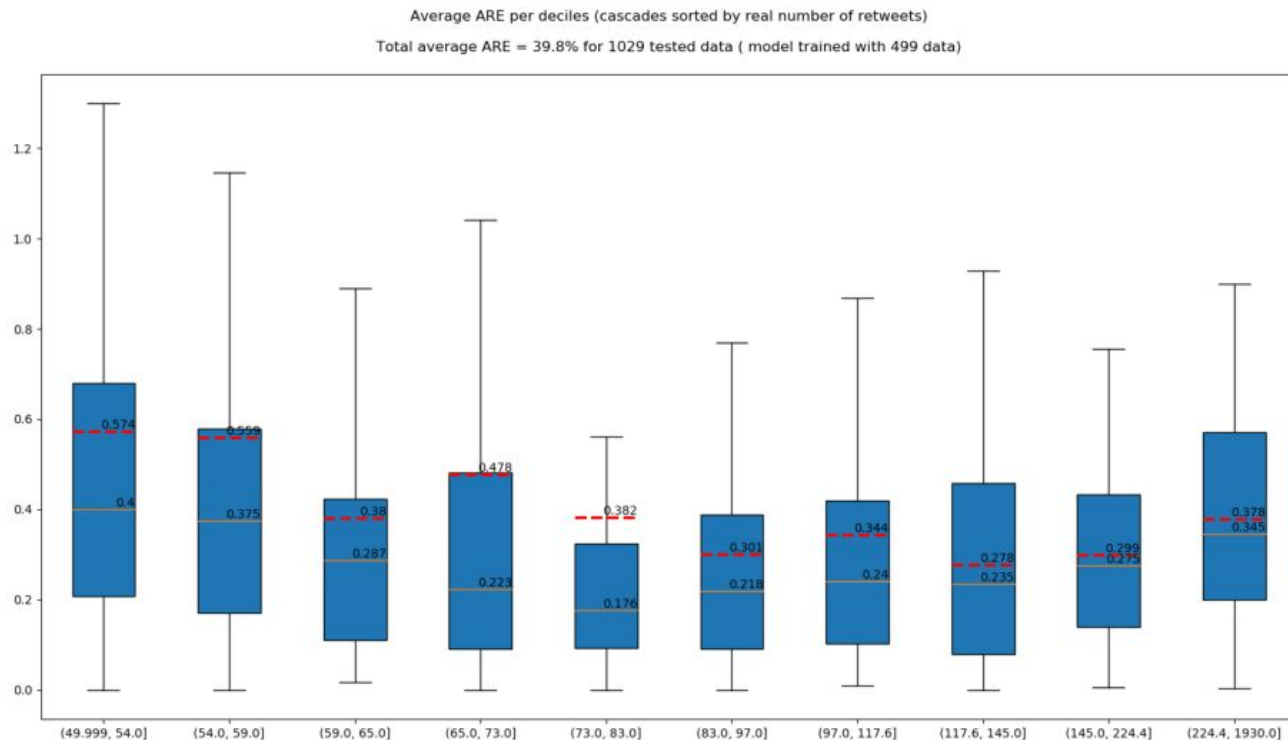


- Nous utilisons une **Random Forest** dont 4 features suffisants sont  $\{ c, \theta, A1, n* \}$
- Performances de la régression, nous utilisons l'**absolute relative error** :  $ARE = \frac{\|N_{real} - \hat{N}_{\infty}\|}{N_{real}}$





# Random Forest



### **III. Notions de Kafka (utilité pour notre projet)**

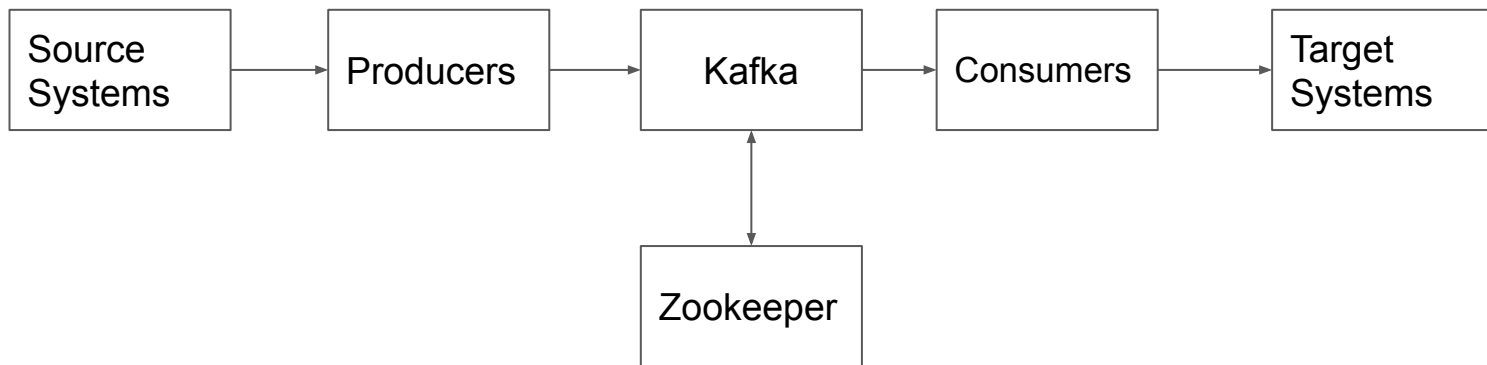
# Pourquoi l'utilisation de Kafka?

- Kafka est une plateforme de streaming **distribuée** utilisée pour publier et s'abonner à des flux de données. Elle est **rapide**, **scalable**, **durable** et **tolérante aux défaillances**.
- Notre solution peut bénéficier de:
  - l'utilisation d'événements
  - l'envoi, la réception et le traitement de flux de données de manière distribuée
  - la modularité
  - le traitement en temps réel



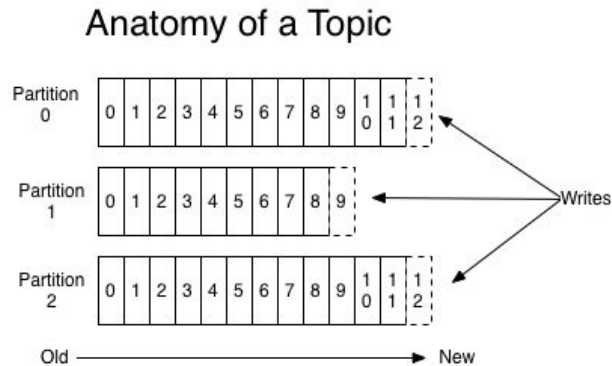
→ intégration de notre solution dans une architecture Kafka

# Notions de Kafka



# Topics et partitions

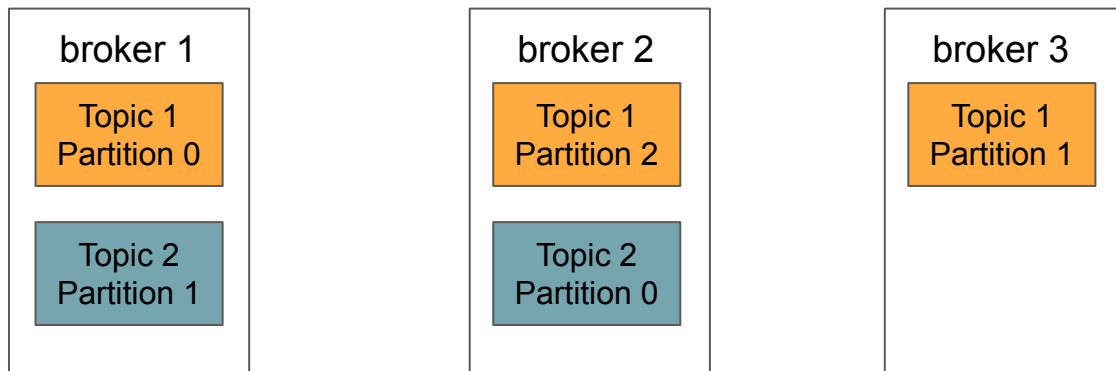
- Topics: un flux de données particulier
  - On peut avoir autant de topics que l'on veut
  - Un topic est identifié par son nom
- Les topics sont répartis en des partitions
  - Chaque partition est ordonnée
  - Chaque message dans la partition reçoit un id incrémental, nommé offset



- Un offset n'a de sens que pour une partition spécifique
- L'ordre n'est garantie qu'à l'intérieur d'une partition (non pas à travers les partitions)
- Les données ne sont conservées que pour une durée limitée (2 semaines par défaut)
- Une fois que les données sont écrites sur une partition, elles ne peuvent plus être modifiées (immutabilité)
- Les données sont assignées aléatoirement à une partition, à moins qu'une clé ne soit fournie
- On peut avoir autant de partitions par topic que l'on souhaite (plus de partitions → plus de parallélisme)

# Brokers

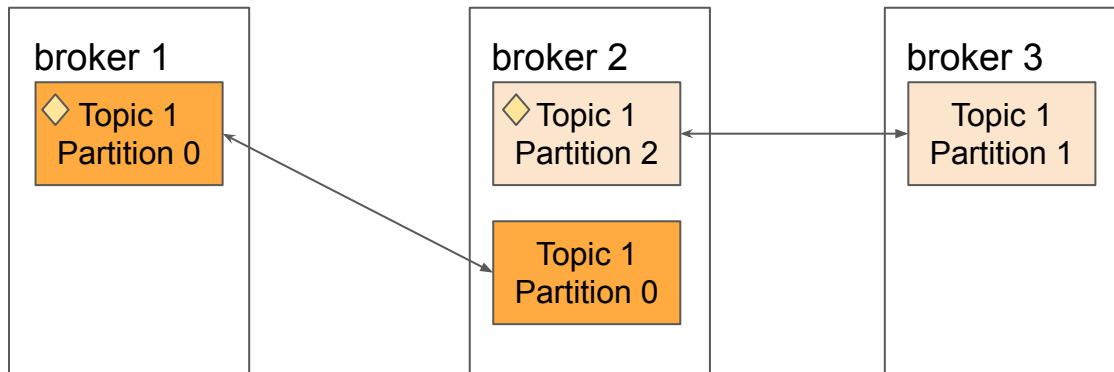
- Un cluster kafka est composé de plusieurs brokers (serveurs)
- Chaque broker est identifié par son id (integer)
- Chaque broker contient certaines partitions de certains topics
- Après être connecté à n'importe quel broker, on sera connecté à l'ensemble du cluster
- Un bon nombre pour commencer est 3, mais certains grands clusters ont plus de 100 brokers (scalabilité horizontale)



# Topic replication factor / Leader pour une partition

- Les topics doivent avoir un replication factor > 1 (souvent 2 ou 3)
- De cette façon, si un broker est en panne, un autre broker peut servir les données
- Un seul broker peut être un leader pour une certaine partition
- Seul ce leader peut recevoir et servir des données pour une partition
- Les autres brokers synchronisent les données
- → Chaque partition a un leader et plusieurs ISR (in-sync replica)

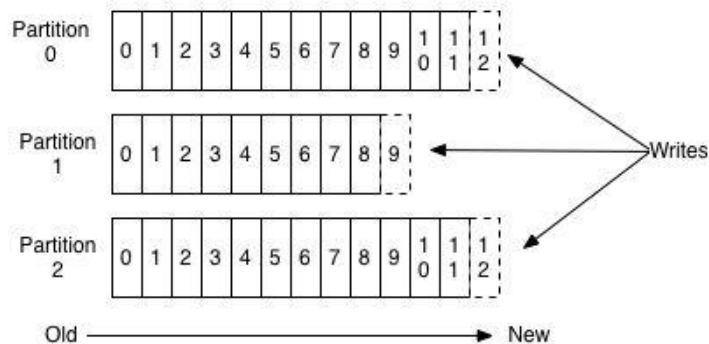
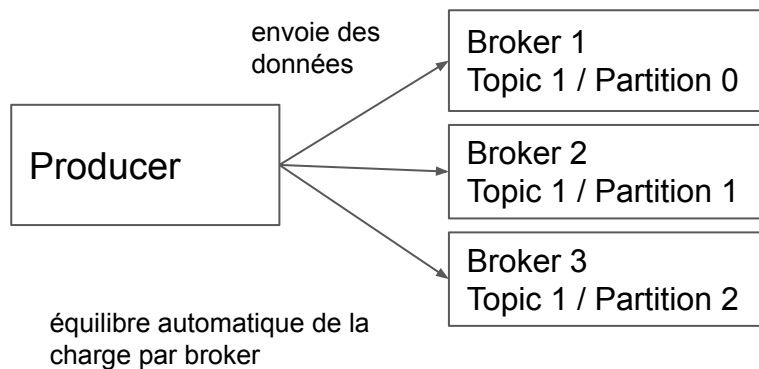
◆ : Leader





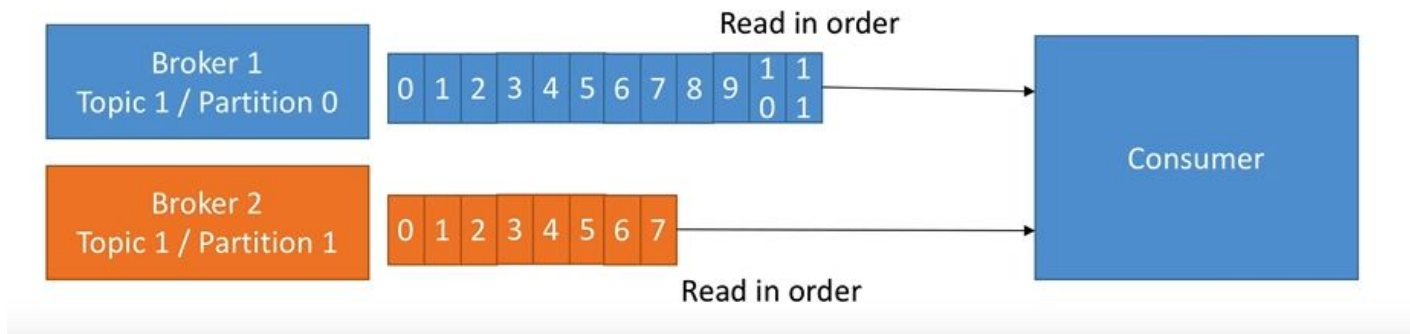
# Producers

- Les producers écrivent des données sur des topics
- Il suffit de spécifier le nom du topic et un broker auquel se connecter, et Kafka se chargera automatiquement d'acheminer les données vers les brokers convenables
- les producers peuvent choisir de recevoir un accusé de réception des données écrites
  - Acks = 0: pas d'accusé
  - Acks = 1: accusé de la part du leader
  - Acks = 2: accusé leader + replicas
- Les producers peuvent envoyer une clé avec le message
- Les messages avec la même clé vont vers la même partition → garantie d'ordre pour la même clé



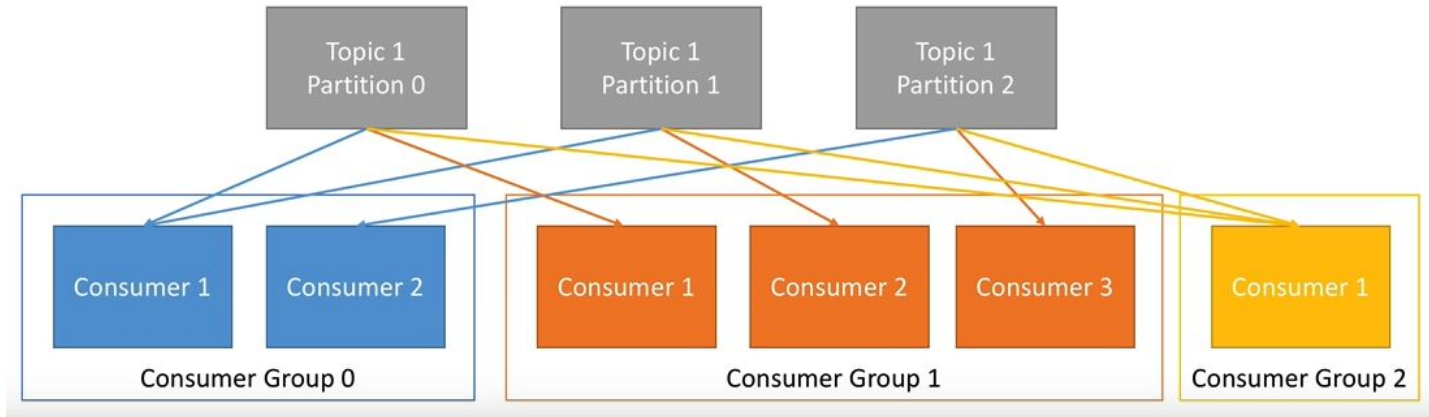
# Consumers

- Les consumers lisent les données des topics
- il leur suffit de spécifier le nom du topic et un broker auquel se connecter, et Kafka se chargera automatiquement de récupérer les données auprès des bons brokers
- Les données sont lues dans l'ordre pour chaque partition



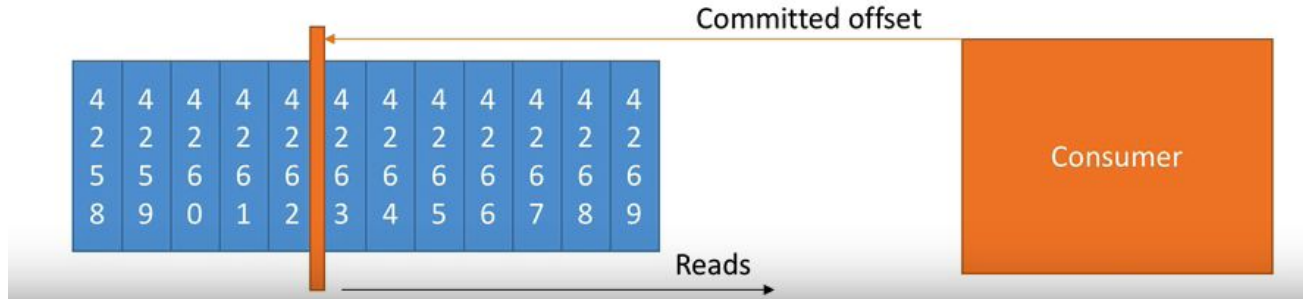
# Consumer Groups

- Les Consumers lisent les données dans des consumer groups
- Chaque consommateur au sein d'un groupe lit à partir de partitions exclusives
- Il est inutile d'avoir plus de consumers que de partitions (sinon certains seront inactifs)

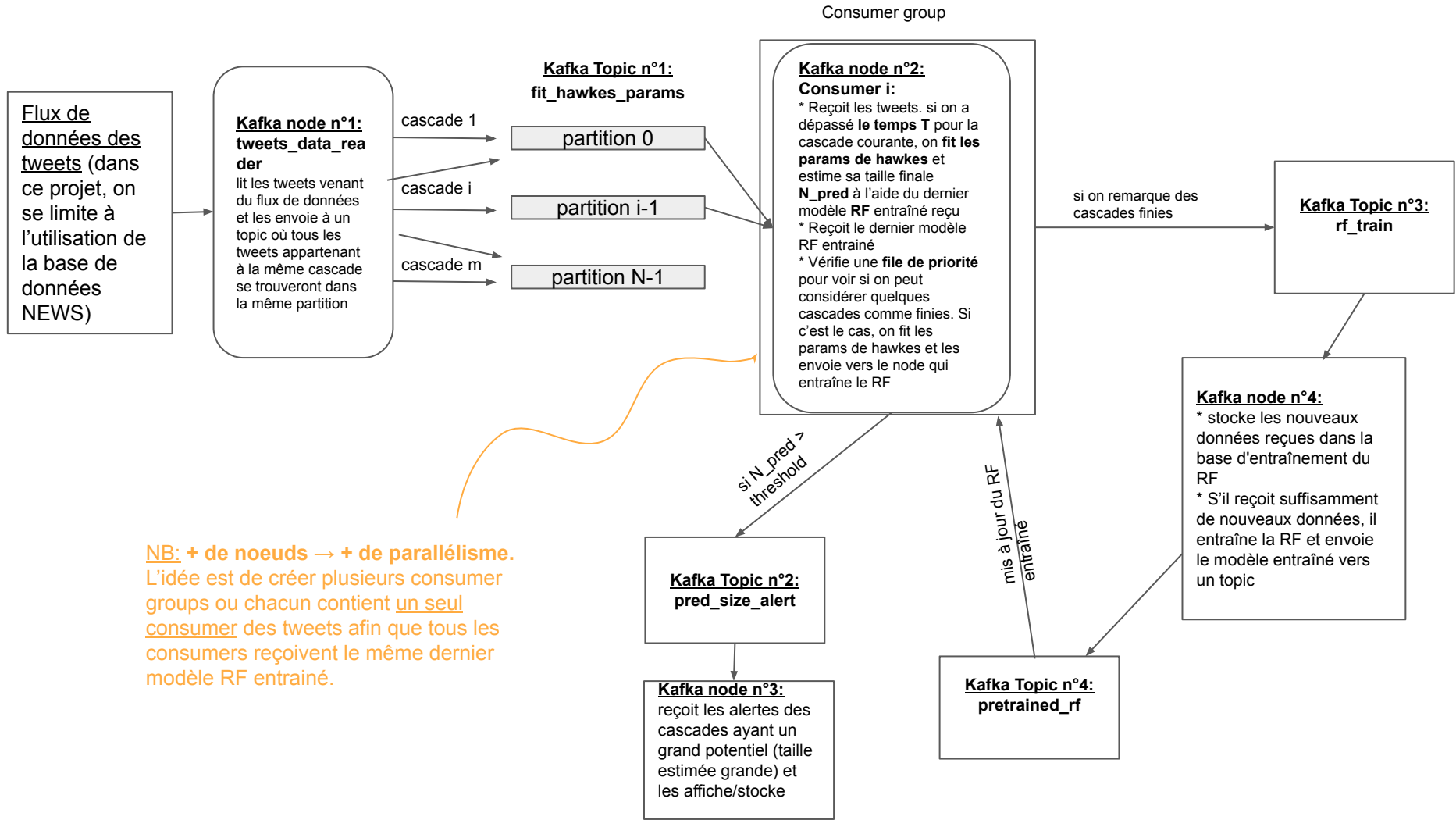


# Consumer offsets

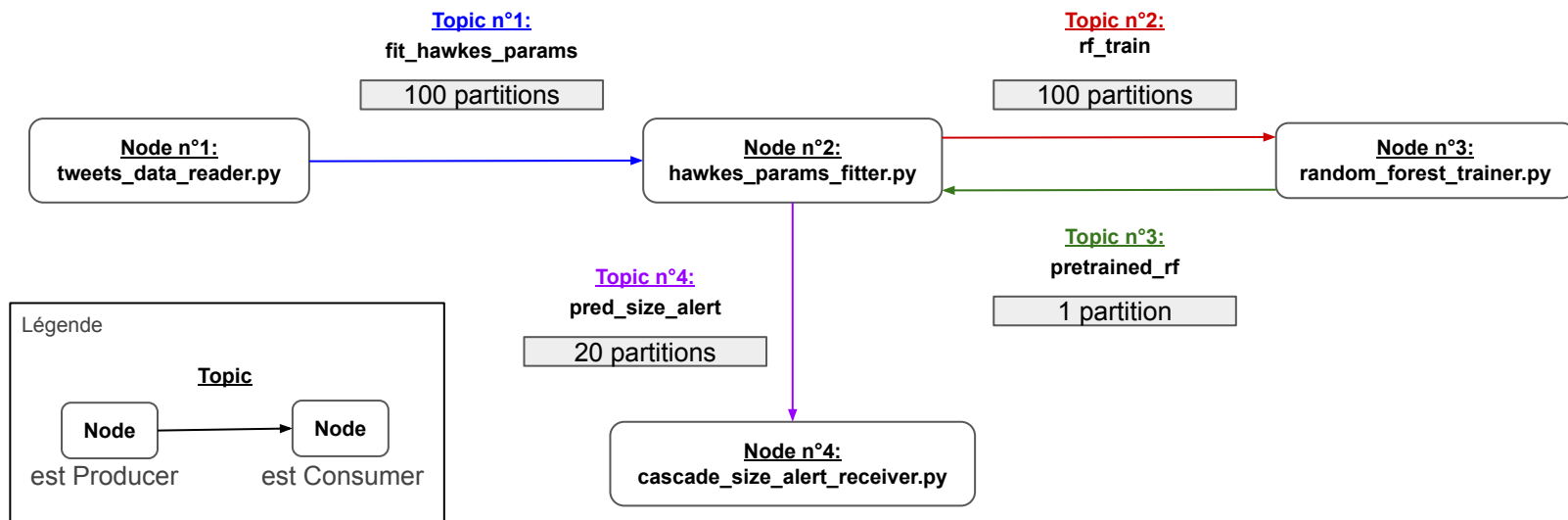
- Kafka stocke les offsets auxquelles un consumer group a lu les données
- Quand un consumer a traité des données issues de Kafka, il est préférable de commit les offsets
- Si le processus d'un consumer meurt, il pourra reprendre là où il s'est arrêté grâce au consumer offsets.



## **IV. Intégration de notre solution dans une architecture Kafka**



**NB: + de noeuds → + de parallélisme.**  
L'idée est de créer plusieurs consumer groups ou chacun contient un seul consumer des tweets afin que tous les consumers reçoivent le même dernier modèle RF entraîné.



# Flux de données des tweets

## Base de données **NEWS**

index.csv

	A	B
1	start ind	end ind
2	1	63
3	64	126
4	127	242
5	243	295
6	296	365
7	366	452
8	453	793
9	794	935
10	936	1087
11	1088	1161
12	1162	1250
13	1251	1316
14	1317	1395
15	1396	1628

data.csv

	A	B
1	time	magnitude
2	0	2086
3	136	357
4	153	340
5	168	724
6	191	5553
7	221	345
8	232	10631
9	365	374
10	368	114
11	394	35
12	409	2711
13	472	420
63	97050	356
64	123921	112
65	0	330
66	4	1010
67	26	1058
68	57	374

- 20093 cascades
- données prétraitées

\* Pour aller plus loin: utiliser L'API de streaming de Twitter gratuite (<https://dev.twitter.com/streaming/overview>) et créer un script pour faire le prétraitement (estimation de l'influence de l'utilisateur pour chaque cascade et création des fichiers csv)



## 1er noeud Kafka


- lecture des données présentes dans les deux fichiers csv

- choix aléatoire d'un tweet à envoyer
- envoie des paramètres qui décrivent le tweet ('time' et 'magnitude') au topic 'fit\_hawkes\_params', la clé du message étant le numéro de la cascade
- mis en pause du noeud d'une durée aléatoire (suivant une loi normale)



en boucle

mutex pour protéger les  
variables globales partagées



## 2ème noeud Kafka (3 Threads)

### Thread 1:

Dès qu'un nouveau modèle random forest entraîné est reçu, on met à jour une variable globale correspondant au modèle utilisé pour les prédictions du scaling factor.

NB: le scaling factor vaut 1 par défaut au cas où aucun modèle n'est encore reçu

### Thread 2:

- reçoit les tweets et les groupe par cascade

- si le temps d'observation (5min, 10min ou 1h) est dépassé, on procède à la prédiction:

- on estime les paramètres de hawkes
- on prédit, à partir du modèle RF, le scaling factor  
→ estimation de la taille de la cascade

- si la taille est > thresh, on envoie une alerte

### Thread 3:

- stocke les cascades reçues par ordre d'ancienneté

- chaque 5 secondes, parcourt toutes les cascades reçues. Si on remarque que pour une cascade, on n'a pas reçu un tweet pendant une durée définie (1h par exemple), on considère que la cascade est finie

- on envoie les cascades finies pour être stockés dans la base d'entraînement de la RF

## 3ème noeud Kafka

- reçoit les cascades considérées finies et les ajoute à la base de données d'entraînement de la RF.
- Quand le nombre de nouvelles cascades reçues dépasse un seuil défini, on entraîne la RF et envoie le modèle entraîné au 2ème noeud

### \* Contrainte actuelle:

- la **longueur maximale** des messages envoyés aux topics kafka est de **~1M caractères**
  - avant envoi, le modèle est sérialisé avec **compress\_pickle**.
  - La longueur du message est **proportionnelle** à la taille de la base de données (500 données → ~1M, 2000 données → ~4M)
- **On se limite à 500 données d'entraînement**

### Solutions:

- on augmente la taille maximale des messages, mais pas scalable
- trouver une alternative du diffusion du modèle (e.g: app Flask)

# script de création des topics

```
from kafka.admin import KafkaAdminClient, NewTopic
from kafka.errors import TopicAlreadyExistsError
import time

admin_client = KafkaAdminClient(bootstrap_servers="localhost:9092", client_id='pfe2019')

# for now, we only have one broker
# when possibility to have more brokers is available, change replication factor to 2!!
topics_list = [
    NewTopic(name='fit_hawkes_params', num_partitions=100, replication_factor=1),
    NewTopic(name='rf_train', num_partitions=100, replication_factor=1),
    NewTopic(name='pred_size_alert', num_partitions=20, replication_factor=1),
    NewTopic(name='pretrained_rf', num_partitions=1, replication_factor=1)
]

# if topic doesn't exist create it
for topic in topics_list:
    try:
        admin_client.create_topics(new_topics=[topic])
    except TopicAlreadyExistsError:
        pass
```

Quand on dispose de plusieurs machines, on peut augmenter le nombre de brokers.

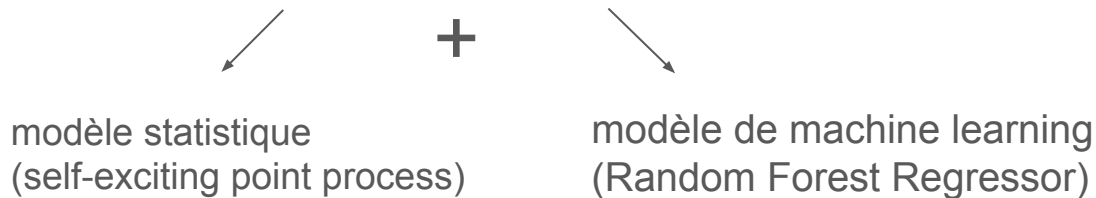
→ Distribuer le traitement des tweets sur les différentes machines

**\* Il est très important dans ce cas, d'augmenter le replication factor à 2 ou 3 lors de la création des topics pour limiter la perte des données.**

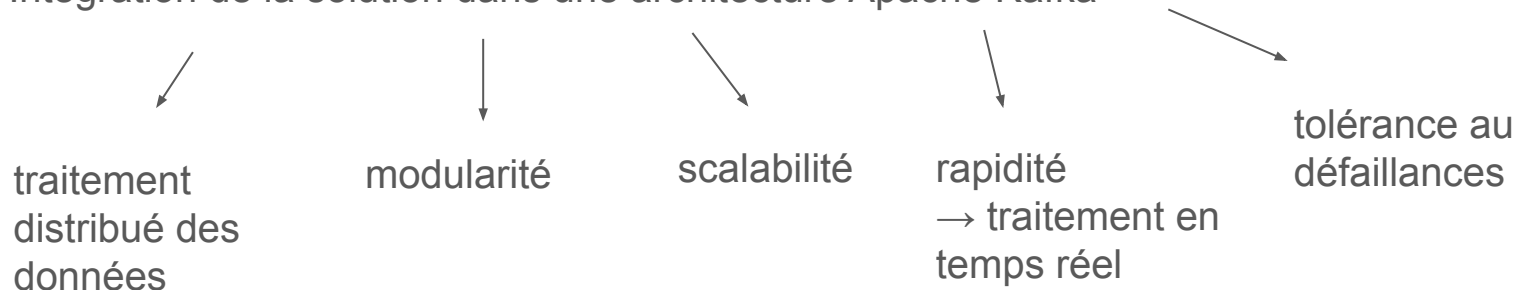
# Conclusion

## Pour résumer

- Application d'une solution issue d'un article scientifique



- Intégration de la solution dans une architecture Apache Kafka



## Pour aller plus loin

- Accroître le parallélisme en distribuant le traitement sur plusieurs machines.  
→ augmenter le nombre de [brokers](#), de [consumer groups](#), de [consumers](#) et le [replication factor](#) des topics pour profiter de la tolérance aux défaillances de Kafka.
- Pour l'instant le flux de données de tweets est simulé. L'idée est de traiter des tweets publiés en temps réel → Utiliser l'API de streaming de Twitter et automatiser le prétraitement.
- Combiner cette solution avec l'approche axée sur les features de l'article pour améliorer la qualité de l'estimation.
- Conteneuriser la solution et la déployer sur le cloud → utiliser Kubernetes
- Connecter l'architecture avec des systèmes externes comme les databases → utiliser Kafka Connect

**Merci beaucoup pour votre attention!**