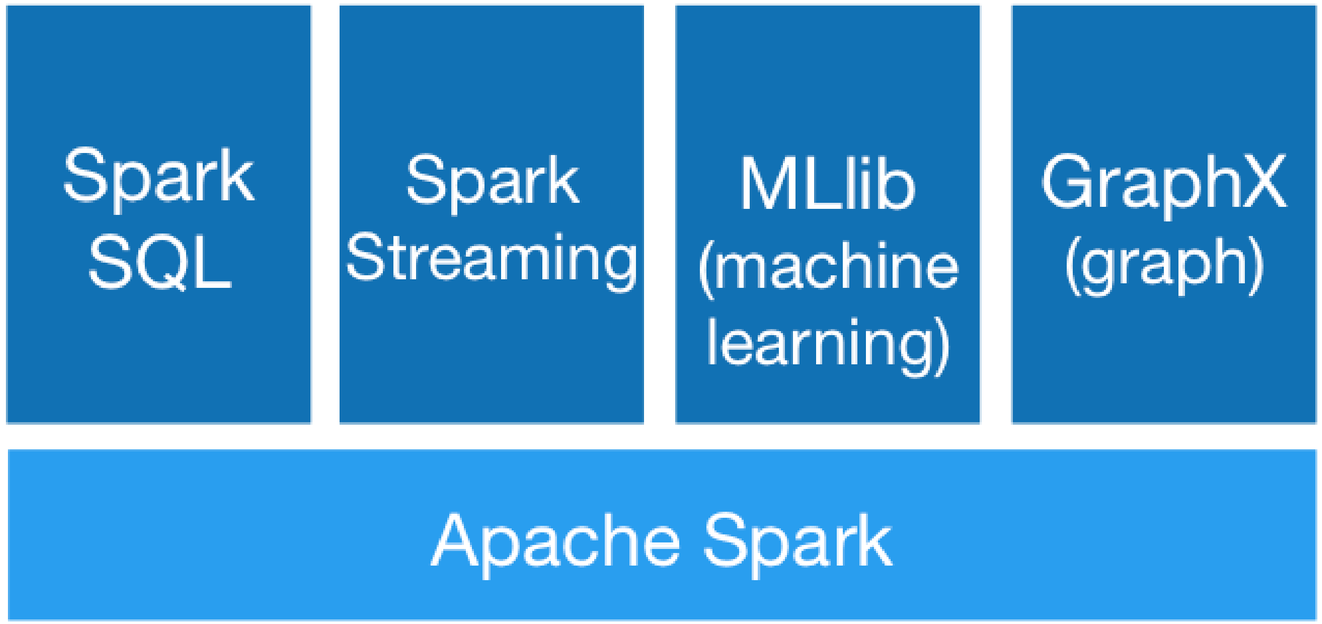
Big Data Analytics

« Pilotage de la performance pour une bonne gouvernance des entreprises »

CHAPITRE 4 – Spark Streaming

Spark Stack



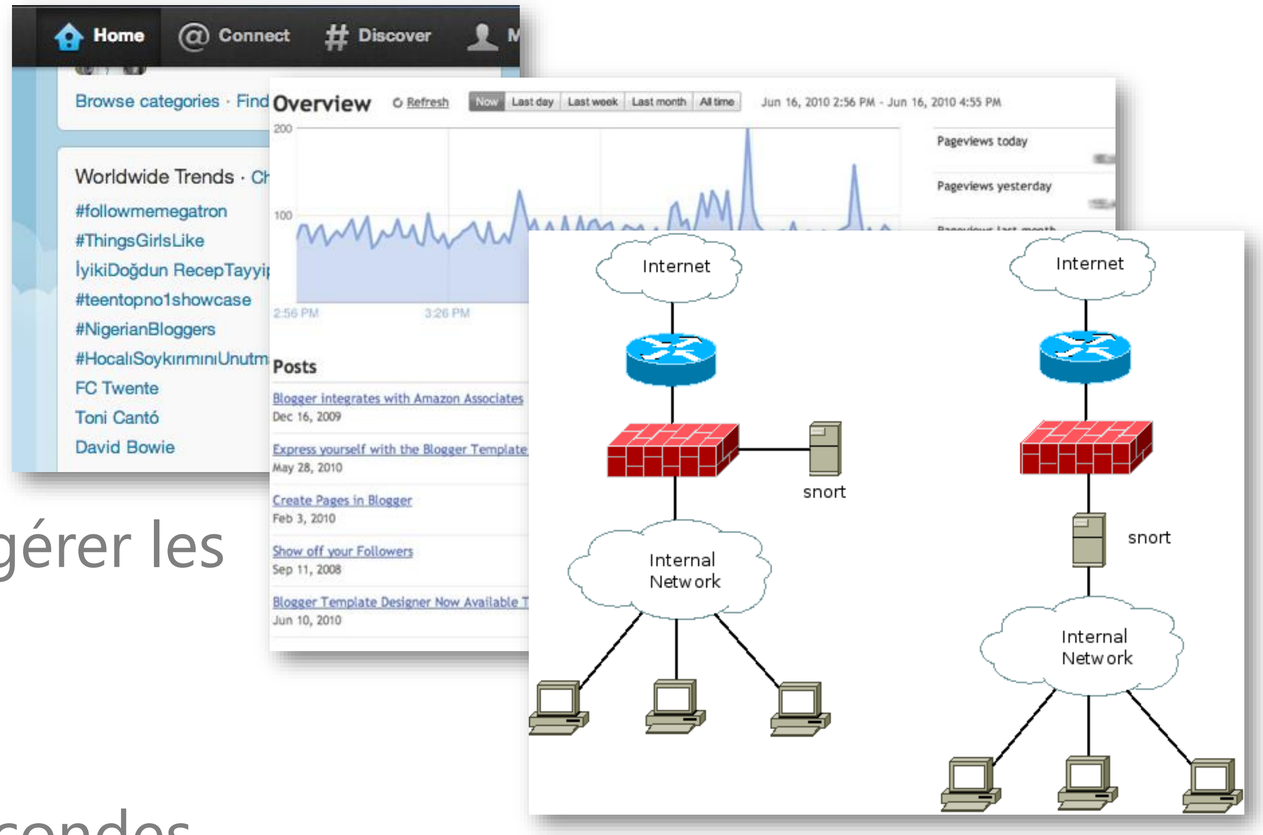
- ***Spark SQL :*** pour le traitement de données (SQL et non structuré)
- ***Spark Streaming :*** traitement de flux de données en direct (live streaming)
- ***MLlib :*** Algorithmes Machine Learning
- ***GraphX :*** Traitement de graphes

Qu'est-ce que Spark Streaming ?

- Framework pour le traitement de flux à grande échelle
  - Échelles à des centaines de nœuds
  - Peut atteindre des latences à échelle de seconde
  - S'intègre au traitement par lots et interactif de Spark
  - Fournit une API simple pour la mise en œuvre d'un algorithme complexe
  - Peut absorber les flux de données en streaming de Kafka, Flume, ZeroMQ, ...

Motivation

- De nombreuses applications importantes doivent traiter de gros flux de données en temps réel et fournir des résultats en temps quasi réel.
  - Tendances des réseaux sociaux
  - Statistiques du site
  - Systèmes de détection d'intrusion
  - …



- Nécessite de grands clusters pour gérer les charges de travail

- Exiger des latences de quelques secondes

Besoin d’un framework…

… Pour construire de telles applications complexes de traitement de flux

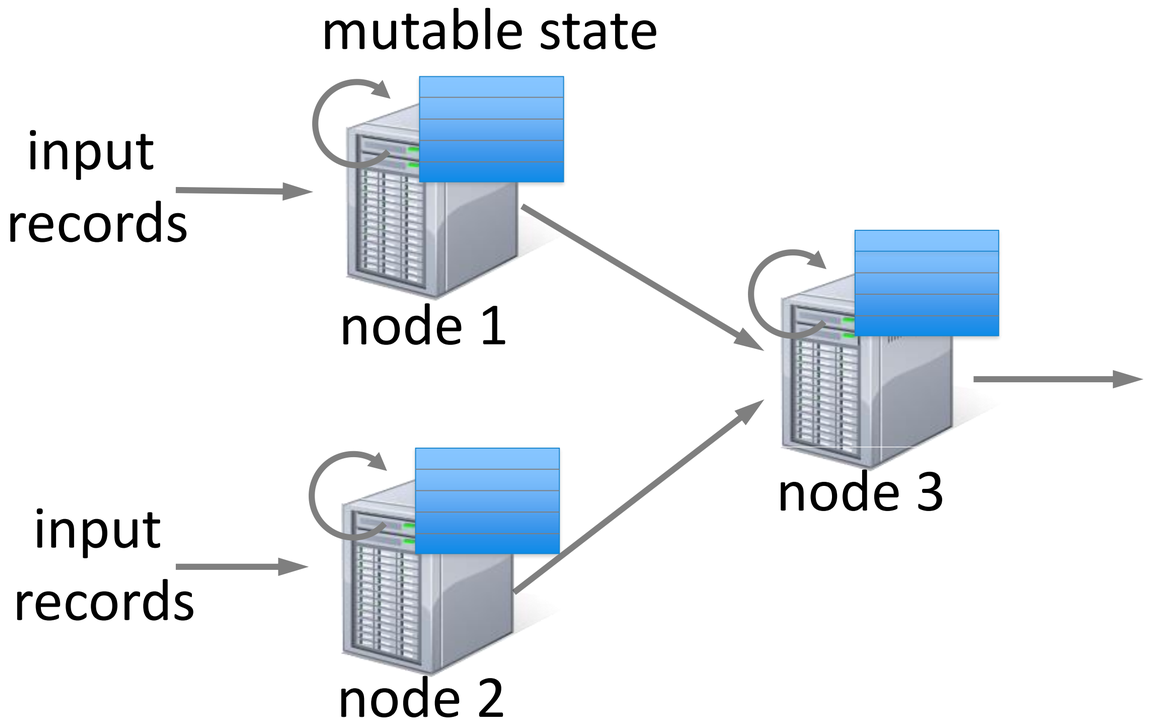
Mais quelles sont les exigences
d'un tel framework?

Exigences

- Evolutif aux grands groupes
- Latences de deuxième échelle
- Modèle de programmation simple
- Intégré au traitement par lots et interactif

Traitement de flux avec état

- Les systèmes de streaming traditionnels ont un modèle de traitement enregistrement par enregistrement basé sur les événements
  - Chaque nœud a l'état mutable
  - Pour chaque enregistrement, mise à jour et envoi de nouveaux enregistrements
- L'état est perdu si le nœud meurt !
- Faire du traitement de flux avec état tolérant aux pannes est un défi



Systèmes de streaming existants

- Storm
  - Enregistrement en boucle si non traité par un nœud
  - Traite chaque enregistrement au moins une fois
  - Peut mettre à jour l'état mutable deux fois !
  - L'état mutable peut être perdu en cas d'échec !

- Trident – Utiliser les transactions pour mettre à jour l'état
  - Traite chaque enregistrement exactement une fois
  - Mise à jour lente des transactions par État

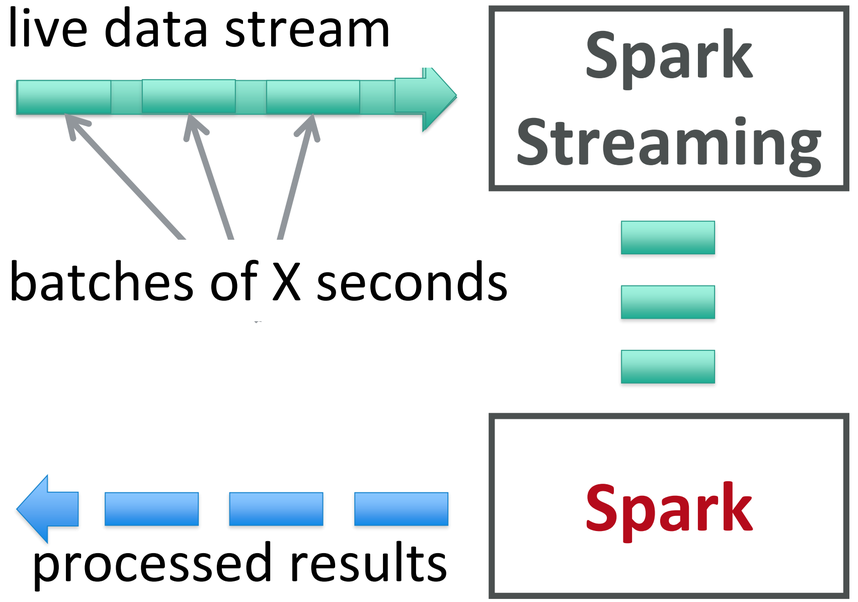
Exigences

- Evolutif aux grands groupes
- Latences de deuxième échelle
- Modèle de programmation simple
- Tolérance aux pannes efficace dans les traitements avec état

Traitement de flux discretisé

Exécuter un calcul en continu sous forme d'une série de très petits travaux par lots déterministes

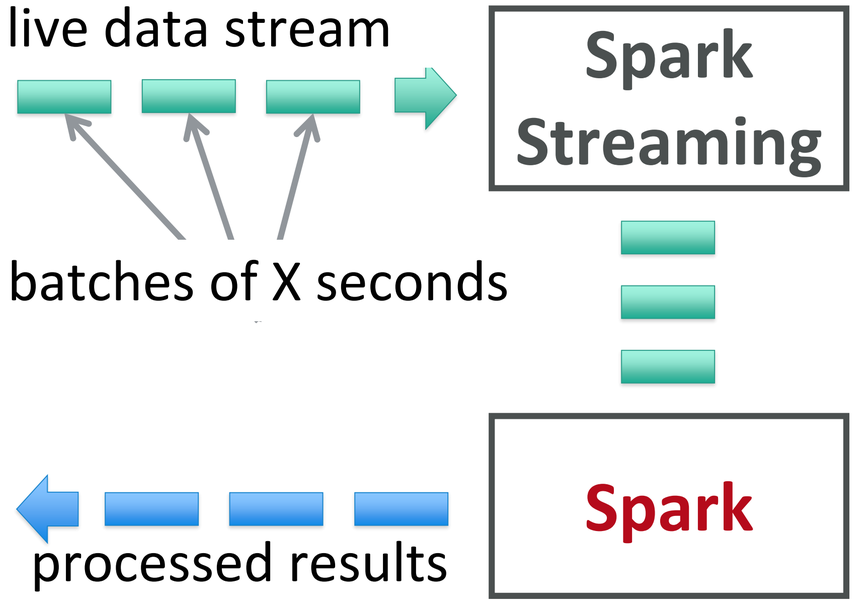
- Couper le flux de streaming en lots de X secondes
- Spark traite chaque lot de données comme des RDD à l'aide d'opérations de RDD.
- Enfin, les résultats traités des opérations RDD sont renvoyés par lots.



Traitement de flux discretisé

Exécuter un calcul en continu sous forme d'une série de très petits travaux par lots déterministes

- Taille des lots aussi basse que ½ seconde, latence ~ 1 seconde
- Possibilité de combiner le traitement par lots et le traitement en continu dans le même système



Exemple 1 - Récupérer les hashtags de Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

tweets DStream

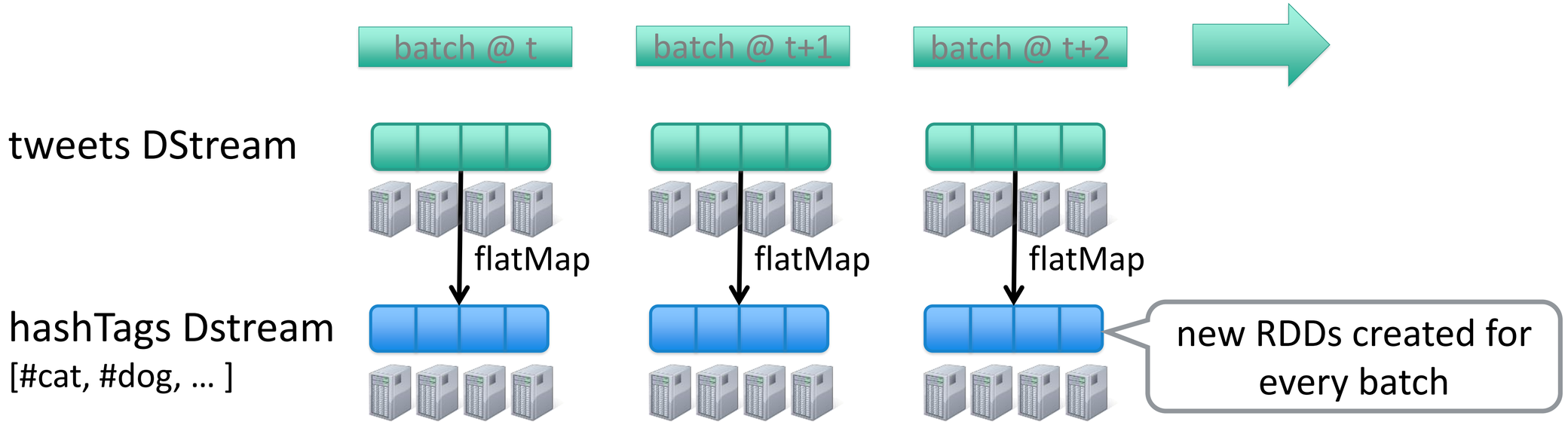
stored in memory as an RDD
(immutable, distributed)

Exemple 1 - Récupérer les hashtags de Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one Dstream to create another DStream

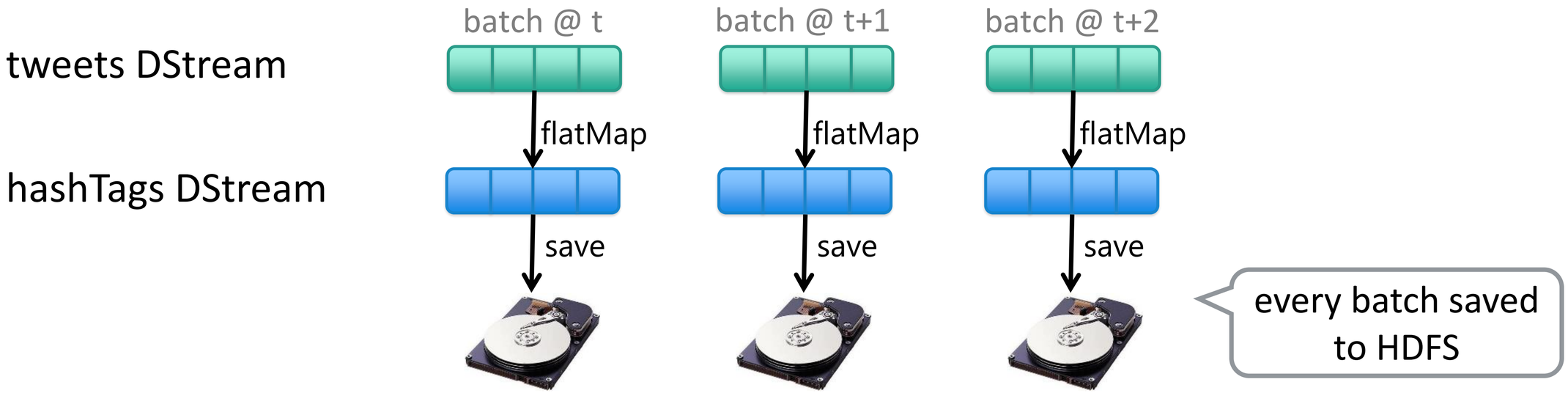


esprit

Exemple 1 - Récupérer les hashtags de Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage



esprit

Exemple Java

Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

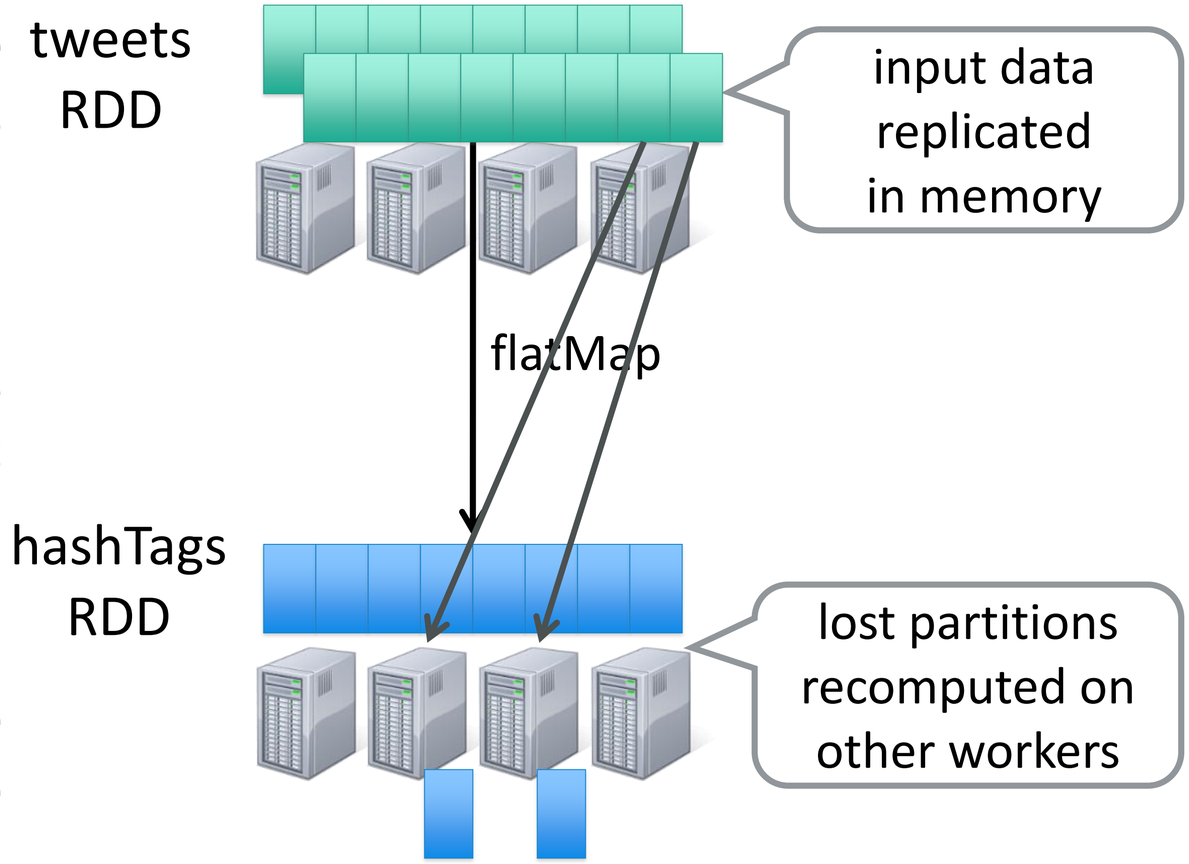
Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })
hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

Tolérance aux fautes

- Les RDD mémorisent la séquence d'opérations créée à partir des données d'entrée tolérantes aux pannes
- Les lots de données d'entrée sont répliqués dans la mémoire de plusieurs nœuds « *Worker* », donc tolérants aux pannes
- Les données perdues en raison d'une défaillance de l'opérateur, peuvent être recalculées à partir des données d'entrée



Concepts clés

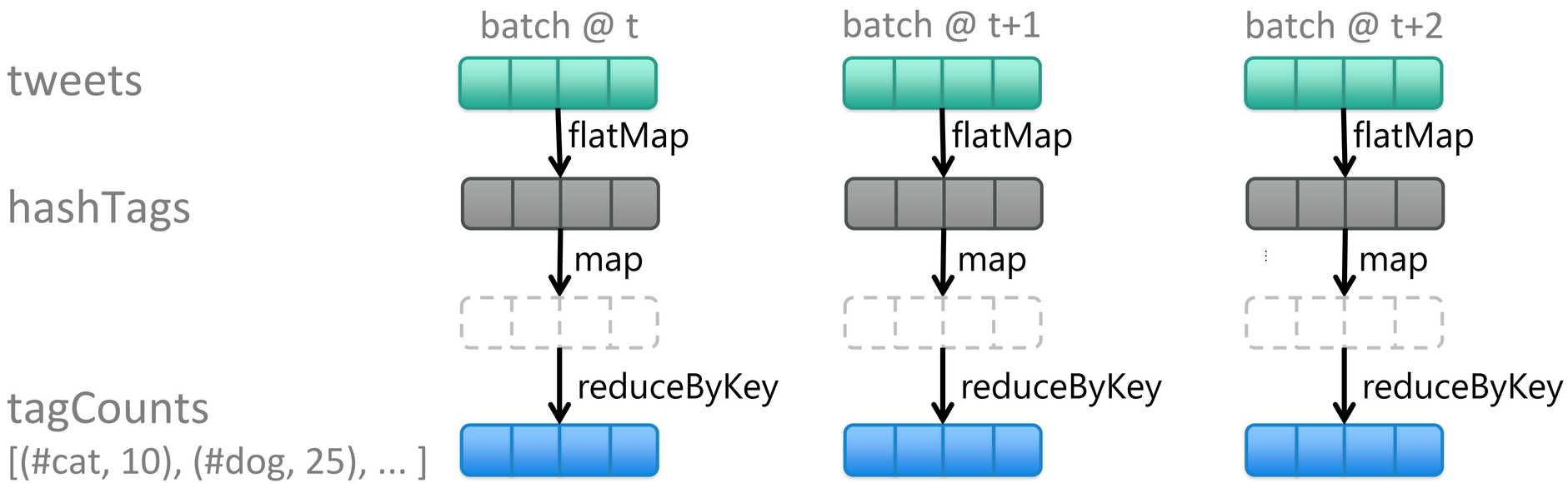
- **DStream** – séquence de RDD représentant un flux de données
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

- **Transformations** – modifier les données d'un DStream à un autre
  - Opérations standard RDD – map, countByValue, reduce, join, …
  - Opérations avec état – window, countByValueAndWindow, …

- **Output Operations – envoyer des données à une entité externe**
  - saveAsHadoopFiles – enregistre dans HDFS
  - foreach – tout faire avec chaque lot de résultats

Exemple 2 - Compter les hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```

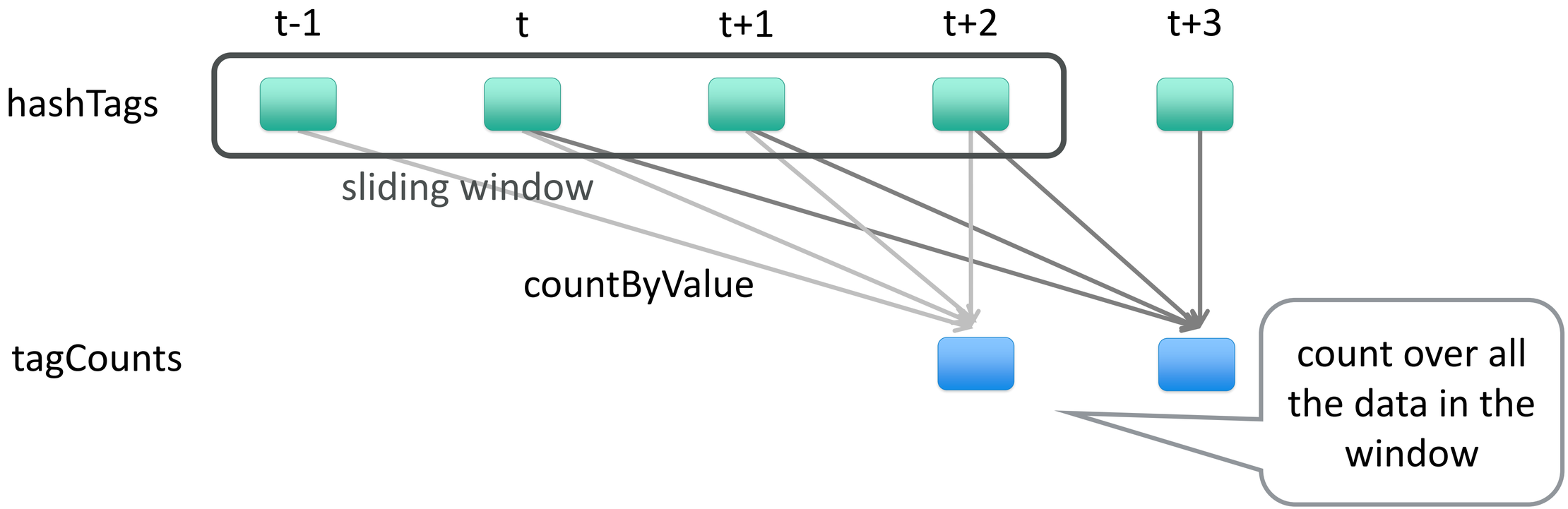


Exemple 3 - Comptez les hashtags sur les 10 dernières minutes

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

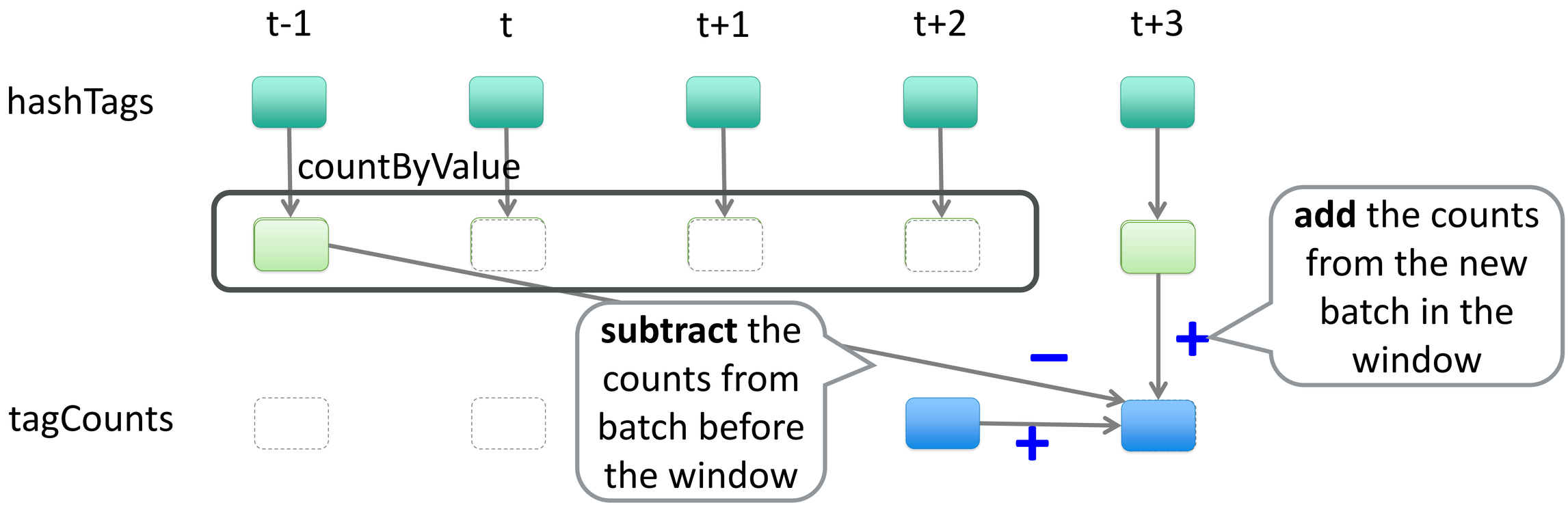
Exemple 3 - Comptez les hashtags sur les 10 dernières minutes

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



CountByValue basé sur Smart window

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



esprit

Reduce base sur Smart window

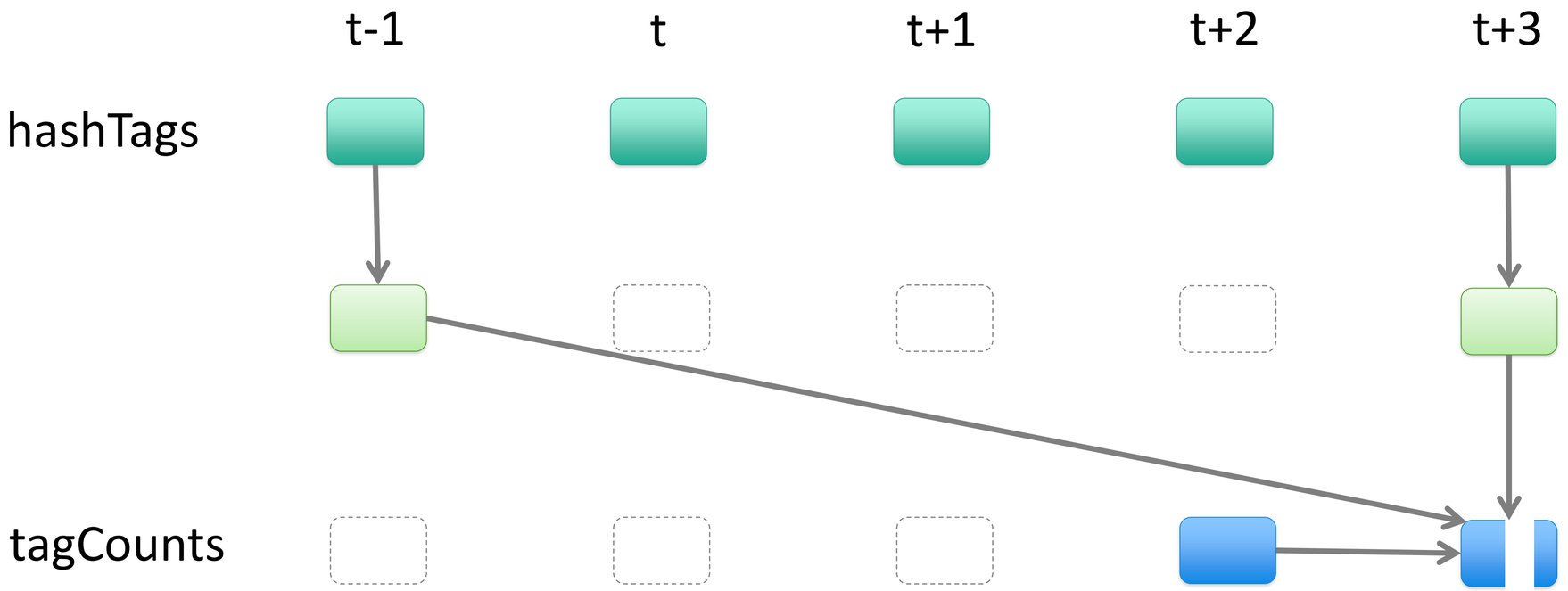
- Technique permettant de calculer progressivement le nombre généralisé à de nombreuses opérations de Reduce
  - Besoin d'une fonction pour *“inverse reduce”* (*“subtract”* pour calculer)

- Implémenter le calcul :

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```

Traitement avec état à tolérance de pannes

Toutes les données intermédiaires sont des RDD, donc peuvent être recalculées en cas de perte



Traitement avec état à tolérance de pannes

- Les données à état ne sont pas perdues même si un nœud « Worker » meurt
  - Ne change pas la valeur de votre résultat

- Exactement une seule sémantique pour toutes les transformation
  - Pas de double calcul !

Autres opérations

- Maintenir un état arbitraire, tracer les sessions
  - Maintenir le sentiment par utilisateur en tant qu'état et le mettre à jour avec ses tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

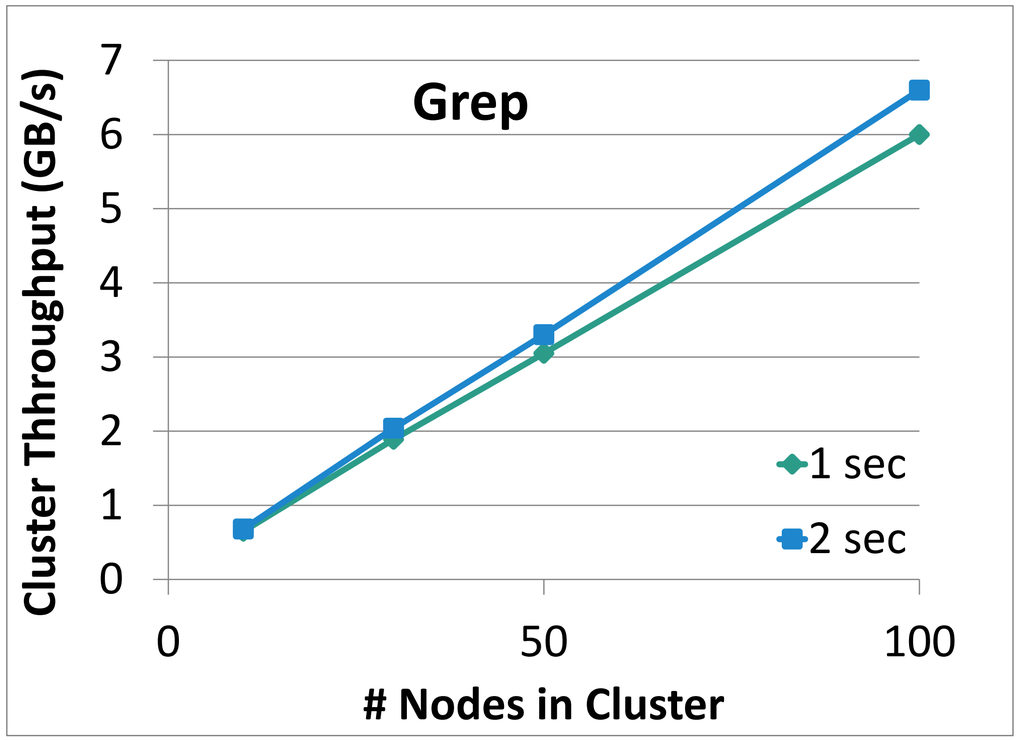
- Effectuer des calculs Spark RDD arbitraires dans DStream
  - Joindre les tweets entrants avec un fichier spam pour filtrer les mauvais tweets

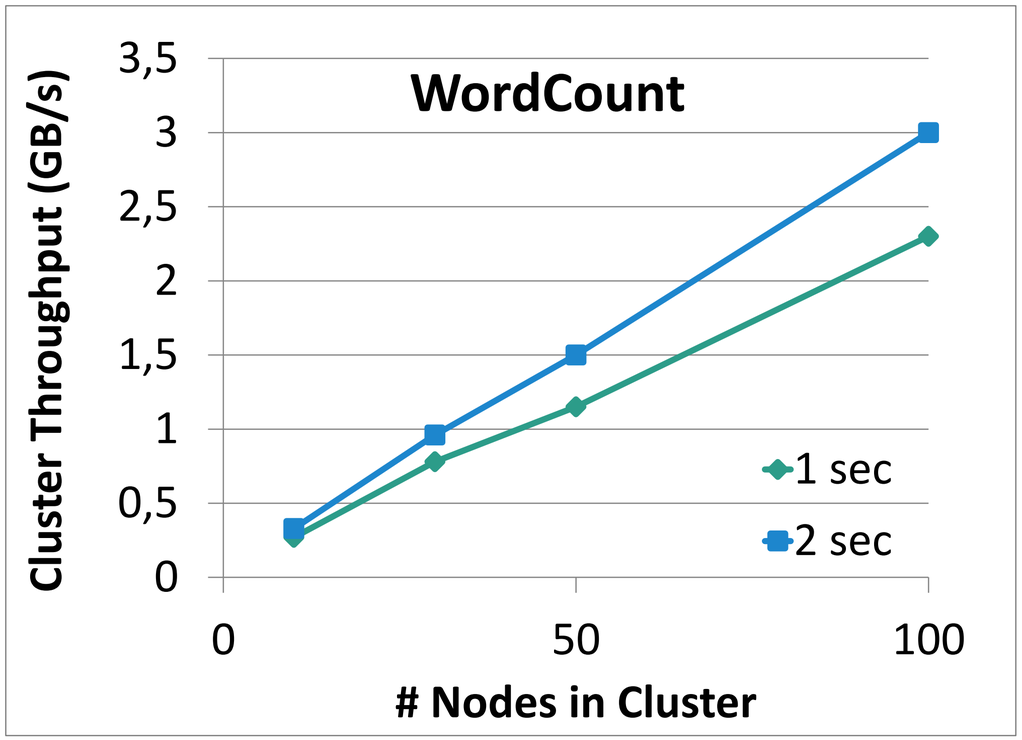
```
tweets.transform(tweetsRDD => {
        tweetsRDD.join(spamHDFSFile).filter(...)
})
```

Performances

Peut traiter **6 Go / seconde** (**60M d'enregistrements / seconde**) de données sur 100 nœuds à une latence **inférieure à 1 seconde**

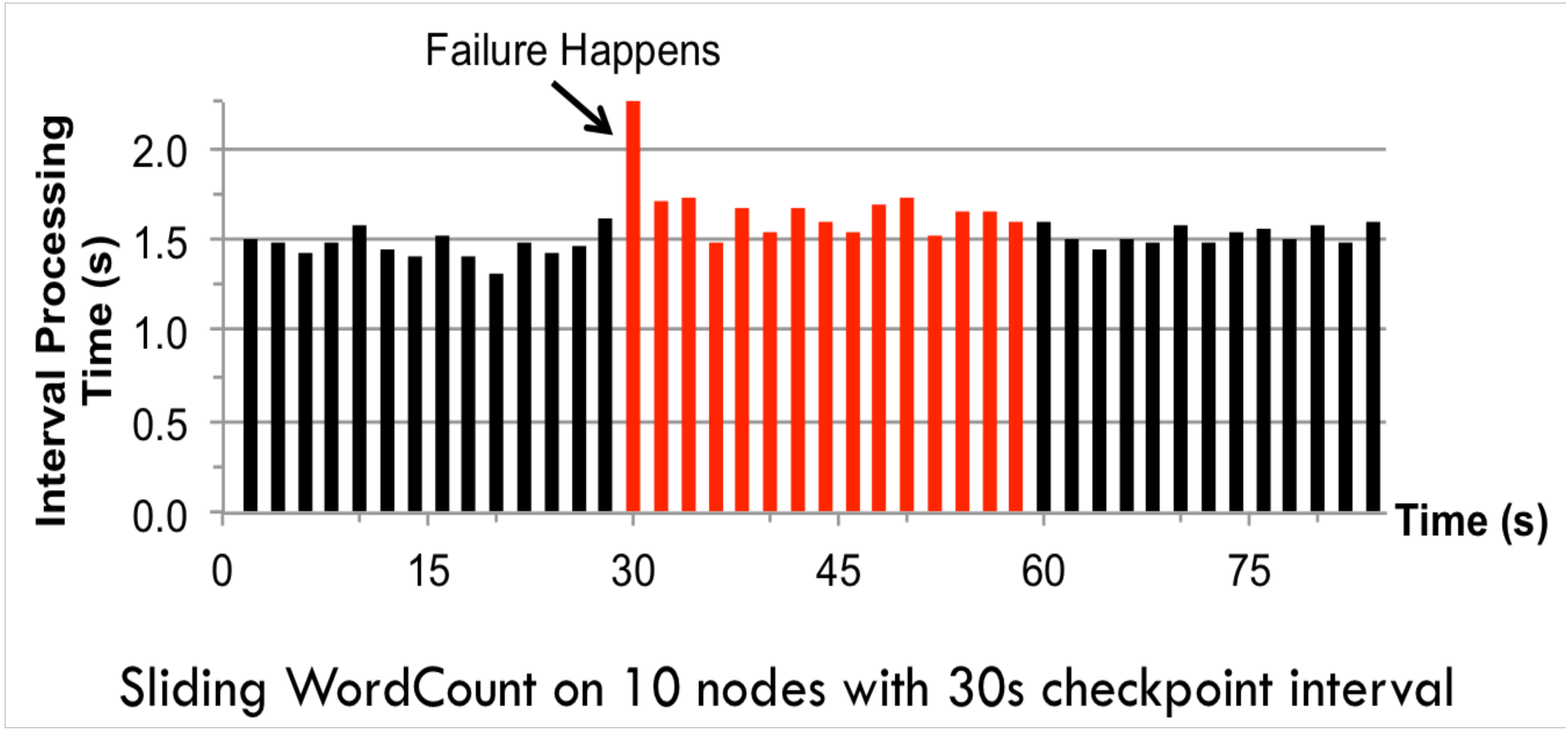
- Testé avec 100 flux de données sur 100 instances EC2 avec 4 cœurs chacun





Récupération Rapide des erreurs

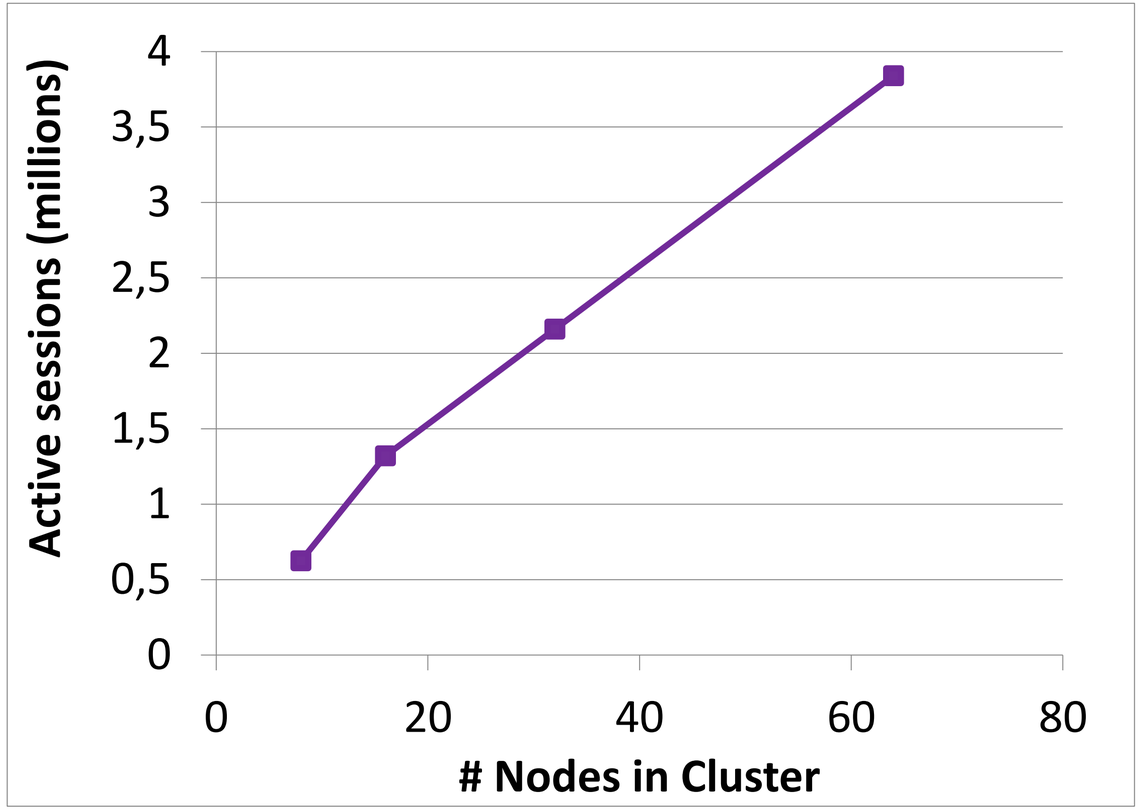
Récupère des défauts / retards en 1 seconde



esprit

Applications réelles : Conviva

- Surveillance en temps réel des métadonnées vidéo
  - Atteint 1-2 secondes de latence
  - Des millions de sessions vidéo traitées
  - Échelle linéaire avec la taille du cluster



esprit

Applications réelles : Mobile Millennium Project

- Estimation du temps de transit du trafic en utilisant l'apprentissage automatique en ligne sur des observations GPS
  - Simulations par chaînes de Markov sur les observations GPS,
  - Très gourmand en ressources processeur, nécessite des dizaines de machines pour un calcul utile,
  - Échelle linéaire avec la taille du cluster.