# Bases de données documentaires et distribuées, http://b3d.bdpedia.fr

## Streaming avec Flink

# Le *Streaming* avec Flink

On a affaire à un **stream**, flux de données de taille (conceptuellement) infinie.

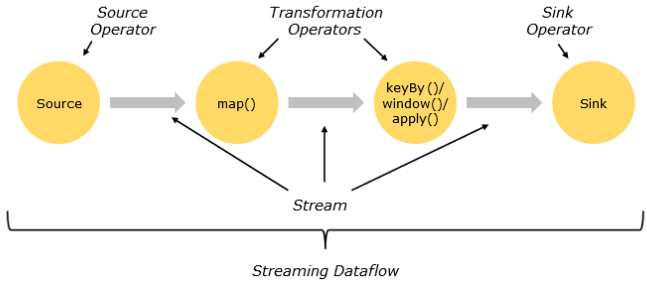
Les éléments de ce flux sont des **item**s dans la terminologie Flink. Un item est de type quelconque, on doit parfois lui attribuer une clé de regroupement.

Et on applique à ce flux et à ces items des **transformations**, en continu

Contrairement à Spark, Flink traite nativement les flux : les items sont pris en compte un par un, instantanémenent (alors que Spark fait du micro-batch, ou pseudo-streaming)

# Les *workflow* de Flink

La source est typiquement un gestionnaire de flux comme Kafka.




La cible est typiquement une base de données.

Les opérateurs (voir la liste dans support de cours) sont à peu près ceux que l'on trouve dans Pig ou Spark.

# Un premier exemple avec l'interpréteur Scala

Au préalable, lancer notre générateur de flux sur le port 9000.

```
// Déclaration du flux de données. NB: senv est l'environnement de streaming
val stream = senv.socketTextStream("localhost", 1234, '\n')
// Application d'un traitement aux fenêtres toutes les 5 secondes
val w = stream.map ({ x => x.toInt + 2 } )
// Voyons ce que cela donne
w.print()
// Workflow défini: on l'exécute
senv.execute("Mon premier traitement de flux ")
```

Notez bien que l'exécution est "paresseuse" : on ne déclenche le *workflow* que dans le *sink* est défini.

# Un workflow = séquence de transformations

S'exprime très simplement en Scala.

```
stream.map ( { x => Tuple1(x.toInt) } )
      .map( {y => (y._1, y._1 * 2) } )
      .print()
```

Astuce : utiliser `:paste` puis `CTRL D` pour entrer des commandes mutli-lignes.

# Traitons des flux d'objets

Plus intéressant : on transforme le flux en entrée en un type donné (comme dans Pig)

```
case class MonDouble(leReel: Float, sonDouble: Double)
val stream = senv.socketTextStream("localhost", 9000, '\n')
val w = stream.map ( { x => Tuple1(x.toFloat) } )
                .map( {y => MonDouble(y._1, y._1 * 2) } )
val fluxFiltre = w.filter ({_.leReel > 1000})
fluxFiltre.print()
senv.execute("Mon premier traitement de flux ")
```

Important : *Streaming* = pipelinage. **Il n'y a pas de matérialisation des résultats intermédiaires**.

# Le fameux compteur de mots

Plus intéressant : on transforme le flux en entrée en un type donné (comme dans Pig)

```
case class CompteurMot(mot: String, compteur: Int)
 val stream = senv.socketTextStream("localhost", 9000, '\n')
 val w = stream.flatMap ({ str => str.split("\\W+") })
        .map({ CompteurMot(_, 1) })
        .keyBy("mot")
        .sum("compteur")
        .print()
senv.execute("Le compteur de mots")
```

# Le même, en utilisant `reduce`

Que signifie `reduce` dans un contexte de flux ? C'est l'accumulation successive des valeurs rencontrées, groupées d'après une clé.

```
case class CompteurMot(mot: String, compteur: Int)
val stream = senv.socketTextStream("localhost", 9000, '\n')
val mots = stream.flatMap ({ str => str.split("\\W+") }).map({ CompteurMot(_
val compte = mots.reduce( (acc, occ)  => {CompteurMot (acc.mot, acc.compteur
senv.execute("Le compteur de mots")
```