Apache Flink : les concepts du TD1

- **Public :** étudiants ayant suivi la partie Kafka du module
- **À utiliser avec :** le TD1 « De Kafka à Flink » (missions 4 à 9)
- **Version :** Flink 2.2 (API DataStream, Java 21), connecteur Kafka 5.0

Ce support présente les notions de Flink dont le TD a besoin, dans l'ordre où le TD les rencontre. Il ne couvre pas l'état durable, les checkpoints, les garanties de livraison ni les watermarks en détail : ce sont les sujets des séances suivantes.

1. Pourquoi un moteur de traitement de flux ?

1.1 Ce que fait un consumer Kafka écrit à la main

Une application qui consomme un topic Kafka avec la bibliothèque cliente (comme le `ConsoleConsumer` de la mission 3 ou le `CounterWorker` du TP Kafka) suit toujours le même schéma :

```
consumer.subscribe(List.of("Counters"));
while (true) {
    for (ConsumerRecord<String, String> r : consumer.poll(Duration.ofMillis(500)))
{
        traiter(r);          // le métier : quelques lignes
    }
}
```

Le métier tient en quelques lignes. Tout le reste est à la charge du développeur :

Préoccupation	**Ce qu'il faut écrire soi-même**
Répartir le travail	Lancer plusieurs exemplaires de l'application ; le groupe de consommateurs se partage les partitions
Regrouper par clé	S'appuyer sur le partitionnement Kafka… qui ne correspond pas toujours à la clé du calcul
Garder un état	Une `Map` en mémoire, perdue à l'arrêt
Reprendre après une panne	Commiter les offsets, sauvegarder et restaurer l'état (topic compacté), le tout de façon cohérente
Calculer par tranche de temps	Découper les fenêtres, décider quand une fenêtre est terminée, gérer les retards
Écrire le résultat	Un producer, et sa cohérence avec les offsets lus (transactions)

Dès que le traitement dépasse « lire, transformer, écrire un message à la fois », ce code d'infrastructure devient majoritaire et difficile à rendre juste.

1.2 Ce qu'apporte Flink

Apache Flink est un moteur de traitement de flux distribué. Le développeur décrit **quoi** calculer (lire, transformer, regrouper, agréger, écrire) ; Flink décide **comment** l'exécuter sur un cluster :

- il découpe le traitement en tâches et les répartit sur plusieurs machines ;
- il achemine les données entre les tâches, en regroupant les éléments de même clé ;

- il conserve l'état de chaque clé et sait le sauvegarder et le restaurer ;
- il gère le temps : fenêtres, temps de l'événement, retards.

```
  producteurs ──► Kafka (topic d'entrée) ──► job Flink ──► Kafka (topic de sortie),
base, fichiers…
                  stockage durable du flux    calcul         résultats
```

Kafka et Flink sont **complémentaires** : Kafka stocke et transporte le flux (durable, rejouable, partitionné), Flink le calcule. Flink n'a pas de stockage de messages propre : il lit et écrit dans des systèmes externes.

1.3 Flux borné, flux non borné

- Un **flux non borné** n'a pas de fin : les compteurs produisent des mesures tant qu'ils fonctionnent. Un job qui le lit ne se termine jamais de lui-même ; il ne peut produire que des résultats **intermédiaires** ou **par tranche de temps**.
- Un **flux borné** a une fin connue (un fichier, le contenu d'une table). Le traiter, c'est du traitement par lots (*batch*).

Flink traite les deux avec la même API : pour lui, un lot n'est qu'un flux qui se termine. Dans ce TD, tous les flux sont non bornés.

2. Le modèle de programmation

2.1 Un job est un graphe de flux

Un programme Flink ne traite pas les messages lui-même : il **décrit un circuit** que les messages vont parcourir. Ce circuit est fait d'**étapes** (les boîtes ci-dessous) reliées par des **flèches** : chaque flèche indique que les éléments produits par une étape sont envoyés à l'étape suivante, toujours dans le même sens.

```
  [ source ] ──► [ transformation ] ──► [ transformation ] ──► [ sink ]
     Kafka          map, filter…          keyBy + reduce…       print, Kafka…
```

Le circuit n'est pas forcément une simple ligne : une étape peut alimenter plusieurs étapes. Dans le `Job03Agregation`, les cumuls partent à la fois vers un affichage et vers un topic Kafka :

```
  [ source ] ──► [ parsing ] ──► [ map ] ──► [ reduce ] ─┬──► [ print ]
                                                         └──► [ Kafka « point-
totals » ]
```

C'est pourquoi on parle de **graphe** (des boîtes reliées par des flèches) plutôt que de chaîne. Le vocabulaire courant : un **graphe orienté** (les flèches ont un sens), ou *dataflow* (les données « coulent » le long des flèches). Dans ce graphe :

- une **source** produit les éléments (lecture d'un topic Kafka) ;
- des **transformations** (ou opérateurs) les modifient, les filtrent, les regroupent ;
- un **sink** les envoie à l'extérieur (affichage, topic Kafka, base de données).

Les arêtes du graphe sont des **DataStream** : un `DataStream<T>` est un flux, potentiellement infini, d'éléments de type T. Chaque transformation prend un ou plusieurs `DataStream` et en produit un nouveau.

2.2 Le squelette d'un job

```
public static void main(String[] args) throws Exception {
    // 1. L'environnement d'exécution : le point d'entrée de tout job
    StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

    // 2. La source
    DataStream<String> lignes = env.fromSource(source,
WatermarkStrategy.noWatermarks(), "ma source");

    // 3. Les transformations
    DataStream<Integer> longueurs = lignes.map(String::length);

    // 4. Le sink
    longueurs.print();

    // 5. Le lancement
    env.execute("mon job");
}
```

`getExecutionEnvironment()` s'adapte au contexte : lancé depuis l'IDE, le programme démarre un **mini-cluster** dans sa propre JVM ; soumis à un cluster (`flink run`), il utilise ce cluster. Le même code sert dans les deux cas. Dans le TD, `Jobs.environment()` ajoute seulement quelques réglages au mini-cluster local (port de l'interface web, nombre de slots).

2.3 Une construction paresseuse

Les étapes 2 à 4 **n'exécutent rien** : elles ajoutent des nœuds au graphe. Aucun message n'est lu lorsque la ligne `map(…)` s'exécute.

C'est l'appel à `env.execute()` qui :

1. fige le graphe décrit jusque-là ;
2. l'envoie au cluster (ou au mini-cluster) ;
3. démarre l'exécution, qui se poursuit jusqu'à l'arrêt du job.

Conséquence : le code de `main` est exécuté **une seule fois**, à la soumission, sur la machine qui soumet le job. Seules les **fonctions** passées aux opérateurs (les lambdas de `map`, `filter`…) sont exécutées sur les TaskManagers, pour chaque élément. Un `System.out.println` placé dans `main` s'affiche donc une fois, chez le client ; placé dans une lambda de `map`, il s'affiche dans les journaux des TaskManagers, une fois par élément.

2.4 Les fonctions utilisateur

Une transformation reçoit une fonction, écrite au choix :

```
// sous forme de lambda ou de référence de méthode
events.filter(e -> e.entries() >= 15);

// sous forme de classe, quand la logique est plus longue ou réutilisée
events.filter(new FiltreAffluence(15));

public static class FiltreAffluence implements FilterFunction<CounterEvent> {
    private final int seuil;
    public FiltreAffluence(int seuil) { this.seuil = seuil; }
    @Override
    public boolean filter(CounterEvent e) { return e.entries() >= seuil; }
}
```

Ces fonctions sont **sérialisées** puis envoyées aux TaskManagers : elles ne doivent référencer que des objets sérialisables (pas de connexion ouverte, pas de client Kafka créé dans `main`).

2.5 Les types des éléments

Flink doit connaître le type des éléments de chaque flux pour les sérialiser quand ils passent d'une machine à l'autre. Il le déduit en général tout seul, y compris pour les `record` Java et les classes simples (POJO).

Il n'y parvient pas toujours avec une **lambda qui produit un type générique**, comme `Tuple2<String, Long>` : à l'exécution, Java a effacé les paramètres de type (*type erasure*). Il faut alors les préciser :

```
events.map(e -> Tuple2.of(e.counterId(), 1L))
      .returns(Types.TUPLE(Types.STRING, Types.LONG));
```

Sans `.returns(…)`, le job échoue dès sa construction avec un message explicite (*« could not be determined automatically… »*).

3. Lire et écrire dans Kafka

3.1 La source Kafka

```
KafkaSource<String> source = KafkaSource.<String>builder()
        .setBootstrapServers("kafka-1:19092")                 // comme
bootstrap.servers
        .setTopics("Counters")
        .setGroupId("td1-job01")                              // comme group.id
        .setStartingOffsets(OffsetsInitializer.latest())      // où commencer à
lire
```

```
        .setValueOnlyDeserializer(new SimpleStringSchema())    // octets de la
valeur → String
        .build();

DataStream<String> messages = env.fromSource(source,
WatermarkStrategy.noWatermarks(), "Kafka : Counters");
```

Réglage	**Équivalent côté consumer Kafka**
`setBootstrapServers`	`bootstrap.servers`
`setGroupId`	`group.id`
`setStartingOffsets(OffsetsInitializer.earliest())`	lire depuis le début des partitions
`setStartingOffsets(OffsetsInitializer.latest())`	lire seulement les nouveaux messages
`setStartingOffsets(OffsetsInitializer.committedOffsets(…))`	reprendre aux offsets commités du groupe
`setValueOnlyDeserializer`	`value.deserializer` (la clé est ignorée)

Le second argument de `fromSource` (la stratégie de watermarks) ne sert que pour le temps de l'événement (§ 7) ; `noWatermarks()` signifie « pas de temps de l'événement ».



La source Kafka de Flink n'est pas un membre de groupe de consommateurs. Elle n'utilise ni `subscribe` ni les rebalances : c'est Flink qui **attribue lui-même les partitions** à ses sous-tâches (§ 6.4). Le `group.id` sert à nommer le client et à commiter les offsets dans Kafka, mais Flink ne les commite qu'au moment de ses *checkpoints* : sans checkpoint, le groupe n'apparaît pas, ou sans offsets, dans `kafka-consumer-groups`. Les positions de lecture qui comptent sont celles que Flink sauvegarde dans ses propres checkpoints.

3.2 Les sinks

`print()` est le sink le plus simple : il écrit chaque élément (son `toString()`) sur la sortie standard **de la machine qui l'exécute**, précédé d'un préfixe optionnel et du numéro de sous-tâche quand le parallélisme dépasse 1 :

```
TOTAL:2> PointTotal[pointCode=P003, events=42, …]
```

Attention : `print` numérote les sous-tâches à partir de **1**, alors que l'interface web les numérote à partir de 0. Sur le cluster, cette sortie se trouve donc dans les journaux des TaskManagers, pas dans le terminal qui a soumis le job.

Le sink Kafka écrit dans un topic. Il faut lui indiquer comment transformer chaque élément en clé et en valeur :

```
KafkaSink<PointTotal> sink = KafkaSink.<PointTotal>builder()
        .setBootstrapServers("kafka-1:19092")
        .setRecordSerializer(KafkaRecordSerializationSchema.<PointTotal>builder()
                .setTopic("point-totals")
                .setKeySerializationSchema(t ->
t.pointCode().getBytes(StandardCharsets.UTF_8))
                .setValueSerializationSchema(t ->
```

```
t.toJson().getBytes(StandardCharsets.UTF_8))
                .build())
        .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
        .build();

totals.sinkTo(sink);
```

Comme pour un producer Kafka, la **clé** détermine la partition de destination : tous les messages de même clé arrivent dans la même partition, dans l'ordre. Les garanties de livraison (`AT_LEAST_ONCE`, `EXACTLY_ONCE`) reposent sur les checkpoints et seront vues plus tard.

4. Transformer élément par élément

4.1 map, filter, flatMap

Opération	Pour chaque élément reçu	Type de sortie	Exemple
`map`	exactement **1** élément émis	peut changer	`String` → `CounterEvent`
`filter`	**0 ou 1** élément émis : l'élément lui-même, s'il passe le test	identique	garder les mesures dont `entries >= 15`
`flatMap`	**0, 1 ou n** éléments émis	peut changer	découper une phrase en mots ; ignorer une ligne illisible

`map` et `filter` sont des cas particuliers de `flatMap`. Celui-ci reçoit un **collecteur** et décide combien de fois l'appeler :

```
public static class ParseOrSkip implements FlatMapFunction<String, CounterEvent> {
    @Override
    public void flatMap(String json, Collector<CounterEvent> out) {
        try {
            out.collect(CounterEvent.parse(json));   // 1 sortie
        } catch (IllegalArgumentException e) {
            System.err.println("Message ignoré : " + e.getMessage());   // 0 sortie
        }
    }
}
```



Une exception levée dans une fonction utilisateur **fait échouer le job** : Flink l'arrête puis le redémarre selon sa stratégie de redémarrage. Si l'erreur vient d'un message invalide, le redémarrage relit le même message et échoue à nouveau. Un `flatMap` qui écarte les éléments invalides évite cette boucle ; en production, on les envoie plutôt vers une sortie dédiée (l'équivalent d'une DLQ).

4.2 Des opérateurs sans état

Ces trois opérateurs calculent leur sortie à partir du **seul élément courant** : ils n'ont rien à mémoriser entre deux éléments. On dit qu'ils sont **sans état** (*stateless*).

Conséquence pour l'exécution : n'importe quelle instance de l'opérateur peut traiter n'importe quel élément. On

peut donc les dupliquer librement et leur envoyer les éléments sans se soucier de leur clé.

4.3 L'ordre des opérations compte

Deux enchaînements peuvent donner le même résultat avec un coût différent. Filtrer **tôt** réduit le nombre d'éléments que les opérateurs suivants doivent traiter, sérialiser et transmettre. C'est la même règle qu'en SQL, où l'on restreint les lignes avant de joindre ou d'agréger.

5. Regrouper par clé et agréger

5.1 keyBy : un partitionnement logique

Pour calculer un cumul **par point de comptage**, il faut que tous les éléments d'un même point soient traités au même endroit. C'est le rôle de keyBy :

```
KeyedStream<CounterEvent, String> parPoint = events.keyBy(CounterEvent::pointCode);
```

keyBy prend une fonction qui extrait la **clé** de chaque élément. Flink calcule un hachage de cette clé et envoie l'élément à l'instance de l'opérateur suivant chargée de ce hachage. Garanties :

- tous les éléments de **même clé** vont à la **même instance**, toujours la même tant que le parallélisme ne change pas ;
- une instance peut gérer **plusieurs clés** ;
- l'ordre des éléments d'une même clé, venant d'une même instance amont, est conservé.

keyBy ne transforme pas les éléments : il produit un `KeyedStream`, sur lequel deviennent disponibles les opérations **par clé** (agrégations, fenêtres, état).



Comparaison avec Kafka. Le partitionnement Kafka et keyBy reposent sur la même idée (même clé → même destination), mais à deux niveaux différents :

- la clé Kafka est choisie **une fois, par le producteur**, et fixe la partition de stockage ;
- la clé du keyBy est choisie **par chaque traitement**, selon ce qu'il calcule, et peut être différente de la clé Kafka.

Dans le flux des compteurs, la clé Kafka est le capteur (`COUNTER-00001-1`) : les deux portes d'un même point peuvent être dans deux partitions différentes. Un consumer Kafka ne voit donc pas forcément tout un point ; un `keyBy(pointCode)` rétablit ce regroupement, sans toucher au topic.

5.2 Les agrégations continues

Sur un `KeyedStream`, une agrégation combine chaque nouvel élément avec le **résultat courant de sa clé** :

```
// reduce : (cumul courant, nouvel élément) → nouveau cumul, du même type
```

```
DataStream<PointTotal> totals = events
        .map(PointTotal::of)            // chaque mesure → un total « unitaire »
        .keyBy(PointTotal::pointCode)
        .reduce(PointTotal::plus);      // a.plus(b) : additionne les deux totaux
```

Agrégation	**Rôle**
`reduce(f)`	combine deux éléments du même type en un seul, avec une fonction fournie
`sum(champ)`	somme d'un champ (position dans un `Tuple`, ou nom d'attribut d'un POJO)
`min(champ)`, `max(champ)`	minimum, maximum d'un champ
`minBy(champ)`, `maxBy(champ)`	l'élément entier qui porte le minimum, le maximum

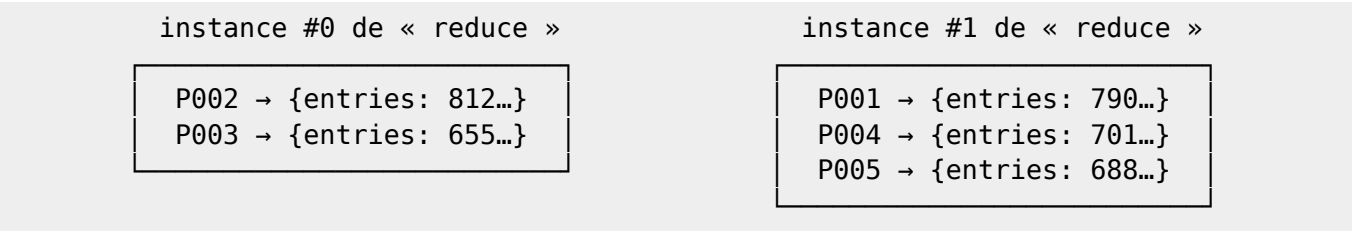
Sur un flux non borné, il n'y a pas de « dernier élément », donc pas de résultat final : ces agrégations **émettent le nouveau résultat à chaque élément reçu**. La sortie est une suite de cumuls de plus en plus grands :

```
P001 : 9   → P001 : 17   → P001 : 30   → …
```

Pour obtenir un résultat par tranche de temps plutôt qu'un cumul sans fin, on utilise une fenêtre (§ 7).

5.3 L'état par clé

Entre deux éléments, l'opérateur `reduce` doit se souvenir du cumul de chaque clé. Ce cumul est un **état par clé** (*keyed state*), conservé par Flink dans l'instance qui gère la clé :



C'est l'équivalent de la `Map<String, …>` qu'un consumer Kafka tiendrait à la main, avec deux différences : Flink la répartit automatiquement entre les instances selon la clé, et il sait la **sauvegarder** (checkpoint) pour la restaurer après une panne ou un redémarrage.



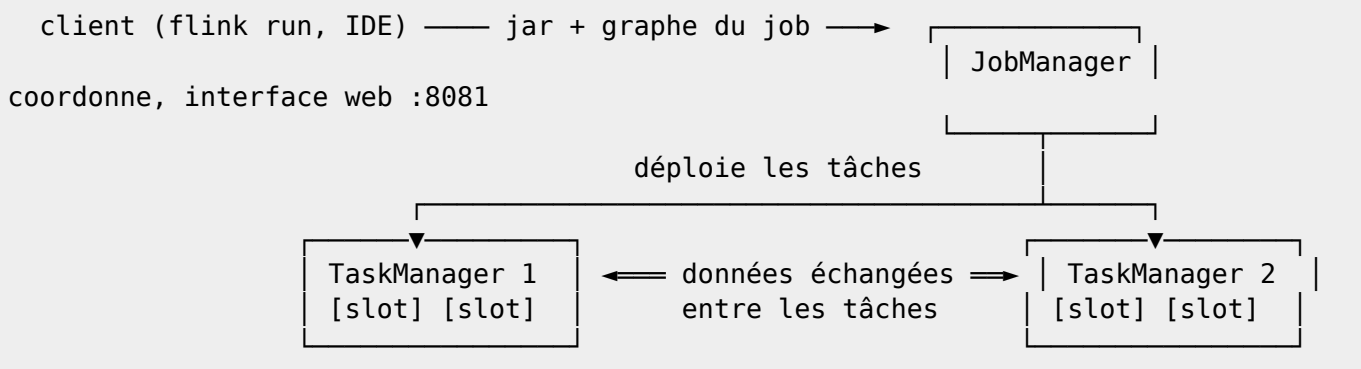
Dans le TD, les checkpoints ne sont **pas activés**. L'état n'existe donc qu'en mémoire : arrêter le job, c'est perdre les cumuls ; le resoumettre, c'est repartir de zéro. Les checkpoints et la reprise sur panne sont l'objet de la séance « état et tolérance aux pannes ».

5.4 Choisir sa clé

- La clé doit correspondre à **ce que l'on calcule** : cumul par point → clé = point ; par site → clé = site.
- Le nombre de clés distinctes borne le parallélisme **utile** : avec 5 clés, au plus 5 instances ont du travail.
- Une clé beaucoup plus fréquente que les autres surcharge l'instance qui la reçoit (*data skew*, clé « chaude »).
- Une clé constante (tous les éléments ont la même) concentre tout le travail sur une seule instance : c'est parfois nécessaire (total global), jamais gratuit.

6. L'exécution distribuée

6.1 Les composants d'un cluster

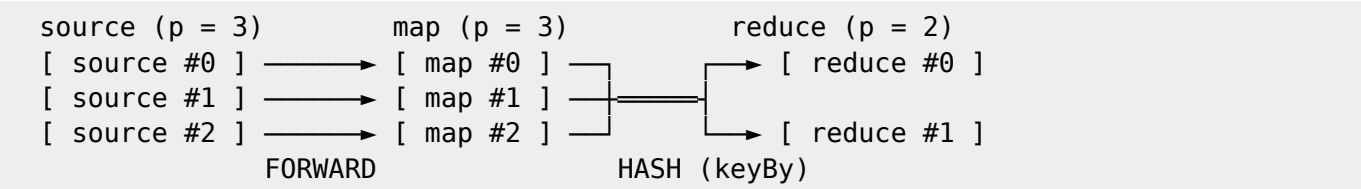


Composant	Rôle
Client	Construit le graphe (exécute `main` jusqu'à `execute()`) et le soumet avec le jar
JobManager	Reçoit les jobs, transforme le graphe en tâches, les place dans des slots, surveille l'exécution, redémarre en cas d'échec ; sert l'interface web
TaskManager	Processus Java (JVM) qui exécute les tâches et échange les données avec les autres TaskManagers
Slot	Une « place » d'exécution dans un TaskManager. Le nombre de slots se règle par TaskManager (`taskmanager.numberOfTaskSlots`)

Un job Flink n'est pas un programme qu'on lance en plusieurs exemplaires : c'est un **programme confié au cluster**, qui décide où et combien de fois exécuter chaque opérateur.

6.2 Parallélisme et sous-tâches

Chaque opérateur s'exécute en un ou plusieurs exemplaires en parallèle. Le **parallélisme** est leur nombre ; chaque exemplaire est une **sous-tâche** (*subtask*), numérotée à partir de 0.



Le parallélisme se règle à plusieurs niveaux ; le plus précis l'emporte :

Niveau	Comment
Opérateur	`.setParallelism(n)` sur l'opérateur, dans le code
Job	`env.setParallelism(n)` dans le code, ou `flink run -p n` à la soumission
Cluster	`parallelism.default` dans la configuration (1 dans le TD)

6.3 Les slots bornent le parallélisme

Par défaut, Flink place dans un **même slot** une sous-tâche de chaque opérateur du job (*slot sharing*) : un slot peut accueillir toute une « tranche » du traitement. Un job a donc besoin d'autant de slots que **le plus grand parallélisme** de ses opérateurs.

S'il n'y a pas assez de slots libres, le job ne peut pas démarrer : il reste en attente de ressources, puis échoue. Les slots occupés par un job ne sont pas disponibles pour les autres : sur un petit cluster, il faut arrêter les jobs dont on n'a plus besoin.

6.4 Partitions Kafka et sous-tâches de la source

Flink attribue les **partitions** du topic aux sous-tâches de la source. Une partition est lue par **une seule** sous-tâche ; une sous-tâche peut lire plusieurs partitions.

Partitions	Sous-tâches source	Répartition
3	1	la sous-tâche lit les 3 partitions
3	3	une partition chacune
3	4	une sous-tâche ne reçoit **rien**

C'est la même règle que pour un groupe de consommateurs Kafka : **le nombre de partitions borne le parallélisme utile de la lecture**. Pour lire plus vite, il faut à la fois plus de sous-tâches et plus de partitions.

6.5 Les liaisons entre opérateurs

La manière dont les éléments passent d'un opérateur au suivant dépend de l'opération :

Liaison	Quand	Effet
FORWARD	même parallélisme, aucune redistribution demandée	la sous-tâche *i* envoie à la sous-tâche *i*, localement
HASH	après un keyBy	chaque élément va à la sous-tâche chargée du hachage de sa clé ; les données traversent le réseau
REBALANCE	parallélismes différents sans keyBy	distribution à tour de rôle, pour équilibrer la charge

Une liaison HASH ou REBALANCE est une **redistribution** (*shuffle*) : les éléments sont sérialisés et peuvent changer de machine. C'est l'étape la plus coûteuse d'un job.

6.6 Le chaînage des opérateurs

Quand deux opérateurs consécutifs sont reliés par une liaison FORWARD (même parallélisme, pas de redistribution), Flink les **chaîne** : il les exécute dans la même tâche, dans le même thread, en se passant les éléments par simple appel de méthode, sans sérialisation ni file d'attente.

```
code       :  source ⟶ map ⟶ filter ⟶ keyBy ⟶ reduce ⟶ print
exécution  :  [ source → map → filter ] ═ HASH ═▶ [ reduce → print ]
                       tâche 1                          tâche 2
```

L'interface web affiche une **boîte par tâche**, avec le nom des opérateurs chaînés à l'intérieur. Le chaînage

améliore nettement les performances ; on peut le désactiver (`env.disableOperatorChaining()`) pour observer chaque opérateur séparément, au prix d'échanges supplémentaires entre tâches.

6.7 Lire l'interface web

Élément	**Ce qu'il montre**
Overview	TaskManagers, slots totaux et disponibles, jobs en cours
Graphe du job	Une boîte par tâche, la liaison entre les boîtes, le parallélisme
Records Received / Sent	Éléments **échangés entre tâches** ; ce qui entre ou sort vers un système externe (Kafka, affichage) n'y figure pas forcément
SubTasks	Le détail par sous-tâche : où elle s'exécute, combien d'éléments elle a traités
Exceptions	Les erreurs qui ont fait échouer le job ou l'ont fait redémarrer

7. Le temps et les fenêtres

7.1 Pourquoi des fenêtres ?

Une agrégation continue (§ 5.2) produit un cumul qui grossit indéfiniment. Pour des indicateurs utiles, on veut des résultats **par tranche de temps** : « entrées du point P001 entre 08:00 et 08:15 ».

Une **fenêtre** regroupe les éléments d'une même clé sur un intervalle de temps et produit un résultat **quand l'intervalle se termine**.

```
events.keyBy(CounterEvent::pointCode)                                    // 1. par
clé
      .window(TumblingProcessingTimeWindows.of(Duration.ofSeconds(10)))   // 2.
découpage du temps
      .process(new Summarize(...));                                       // 3.
calcul sur chaque fenêtre
```

7.2 Les formes de fenêtres

```
  fixes (tumbling)     |----A----|----B----|----C----|        chaque élément dans
une seule fenêtre
  glissantes (sliding) |----A----|
                            |----B----|                      taille et pas
différents : chevauchement
                                 |----C----|
  de session           |-A-|        |---B---|    |-C-|        fermées après une
période d'inactivité
```

Le TD n'utilise que des **fenêtres fixes** (*tumbling*) : de taille constante, sans chevauchement, alignées sur l'horloge (une fenêtre de 15 minutes commence à 00, 15, 30 ou 45). Les deux autres formes seront vues plus tard.

7.3 La fonction de fenêtre

Après `window(…)`, on indique le calcul à faire sur chaque fenêtre :

- `process(ProcessWindowFunction)` : reçoit **tous** les éléments de la fenêtre à sa fermeture, ainsi que ses bornes ; la plus souple, mais Flink doit garder tous les éléments en mémoire ;
- `reduce` / `aggregate` : combinent les éléments **au fil de l'eau**, Flink ne garde qu'un résultat partiel par fenêtre ; plus économes.

```
public static class Summarize extends ProcessWindowFunction<CounterEvent,
WindowSummary, String, TimeWindow> {
    @Override
    public void process(String pointCode, Context ctx, Iterable<CounterEvent>
elements,
                        Collector<WindowSummary> out) {
        // appelée une fois par clé et par fenêtre ; ctx.window() donne le début et
la fin
    }
}
```

7.4 Temps de traitement et temps de l'événement

Toute la question est de savoir **de quelle horloge** on parle :

Horloge	**Définition**	**Dans le flux des compteurs**
Temps de **traitement** (*processing time*)	L'heure de la machine qui exécute l'opérateur, au moment où l'élément passe	l'heure réelle
Temps de l'**événement** (*event time*)	L'heure à laquelle l'événement s'est produit, inscrite dans le message	le champ `timestamp` (horloge simulée)

En **temps de traitement**, une fenêtre contient « ce qui est arrivé pendant ces 10 secondes ». C'est simple et immédiat, mais le résultat dépend de la vitesse du flux, des ralentissements et des redémarrages : relire l'historique ne redonne pas les mêmes fenêtres.

En **temps de l'événement**, une fenêtre contient « ce qui s'est produit entre 08:00 et 08:15 », quel que soit le moment où les messages sont lus. Le résultat est **reproductible** : relire le topic redonne les mêmes fenêtres. C'est l'horloge qui a un sens métier, au prix d'une difficulté : savoir quand une fenêtre est terminée.

7.5 Savoir quand une fenêtre est terminée : les watermarks

En temps de traitement, la fenêtre `[10:00:00, 10:00:10[` se ferme quand l'horloge de la machine atteint 10:00:10.

En temps de l'événement, il n'y a pas d'horloge à consulter : le temps avance **avec les messages**. Et ceux-ci peuvent arriver dans le désordre (plusieurs partitions, plusieurs producteurs, retards réseau). Flink a besoin de deux informations, fournies par une `WatermarkStrategy` :

```
WatermarkStrategy<CounterEvent> strategie = WatermarkStrategy
        .<CounterEvent>forBoundedOutOfOrderness(Duration.ofMinutes(1))   // retard
toléré
```

```
        .withTimestampAssigner((e, tsKafka) -> e.eventTimeMillis());       // où
lire l'heure de l'événement

events.assignTimestampsAndWatermarks(strategie)
      .keyBy(CounterEvent::pointCode)
      .window(TumblingEventTimeWindows.of(Duration.ofMinutes(15)))
      .process(new Summarize(...));
```

1. **Où lire l'heure de chaque événement** : ici, le champ `timestamp` du message.
2. **Où en est le temps du flux** : un *watermark* de valeur *t* signifie « on ne s'attend plus à recevoir d'événement antérieur à *t* ». Avec `forBoundedOutOfOrderness(1 min)`, le watermark suit l'heure d'événement la plus élevée vue jusqu'ici, moins une minute.

Une fenêtre en temps de l'événement se ferme quand le watermark dépasse sa fin : la fenêtre `[08:00, 08:15[` produit son résultat lorsqu'un message daté d'au moins 08:16 environ est passé (08:15 + 1 minute de tolérance).



Une sous-tâche qui ne reçoit aucun message (par exemple une source sans partition) ne fait jamais avancer son watermark, et bloque alors les fenêtres en aval. `withIdleness(…)` indique à Flink de l'ignorer après un certain temps d'inactivité.

Le fonctionnement détaillé des watermarks, le traitement des messages arrivés après la fermeture de leur fenêtre et les fenêtres glissantes et de session sont l'objet de la séance « traitements, fenêtres et agrégations ».

8. Développer et exécuter un job

8.1 Deux modes d'exécution

Mode	Comment	Usage
Local (mini-cluster)	Exécuter `main` depuis l'IDE ou avec Maven : JobManager et TaskManager démarrent dans la même JVM	Mise au point rapide : points d'arrêt, affichage dans la console de l'IDE
Cluster	Construire le jar, puis le soumettre (`flink run`, ou `./scripts/flink.sh run` dans le TD)	Exécution distribuée, comme en production

Sur le cluster, un job soumis s'exécute avec **le jar tel qu'il était à la soumission**. Pour prendre en compte une modification : recompiler, arrêter l'ancien job, soumettre le nouveau.

8.2 Les dépendances « provided »

Le cluster fournit déjà les bibliothèques de Flink. Le jar d'un job ne contient donc que son propre code et les bibliothèques que le cluster n'a pas (ici, le connecteur Kafka). Dans le `pom.xml`, les dépendances de Flink sont déclarées avec le scope `provided` : utilisées à la compilation, mais pas embarquées dans le jar.

Pour exécuter le job **hors du cluster** (depuis l'IDE), il faut les remettre dans le classpath : c'est le réglage *Add dependencies with “provided” scope to classpath* d'IntelliJ.

8.3 Paramétrer un job

Un job reçoit ses arguments comme tout programme Java (`String[] args`). Passer les valeurs qui varient (seuil, topic, point de départ dans Kafka, parallélisme…) en **paramètres** plutôt qu'en constantes permet de soumettre le même jar avec des réglages différents, sans recompiler. Dans le TD, la classe `Params` lit les arguments de la forme `–nom valeur`.

9. Du consumer Kafka au job Flink

Préoccupation	**Consumer Kafka écrit à la main**	**Job Flink**
Lire le topic	`subscribe` + boucle de `poll`	`KafkaSource` + `fromSource`
Répartir les partitions	Le groupe de consommateurs, avec ses rebalances	Flink attribue les partitions aux sous-tâches de la source
Exécuter en parallèle	Lancer N exemplaires de l'application	Régler le parallélisme ; le cluster place les sous-tâches
Regrouper par clé	Dépend de la clé Kafka choisie par le producteur	`keyBy` sur n'importe quel champ
Garder un état	`Map` en mémoire	État par clé géré par Flink
Sauvegarder l'état et la position de lecture	Commit des offsets + topic compacté, à synchroniser soi-même	Checkpoints (séance suivante)
Fenêtres de temps	Calcul des bornes, détection de la fin, à la main	`window(…)` + watermarks
Message invalide	`try/catch`, DLQ à la main	`flatMap` qui écarte, sortie dédiée
Écrire le résultat	Producer, transactions	`KafkaSink`, garanties de livraison

Le développeur garde la responsabilité du **métier** : quelles données lire, quelle clé choisir, quels calculs faire, quelle horloge utiliser. Flink prend en charge la **mécanique** de l'exécution distribuée.

10. Aide-mémoire de l'API

```
// Environnement et lancement
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(2);                          // parallélisme par défaut du job
env.disableOperatorChaining();                  // désactiver le chaînage (pour
observer)
env.execute("nom du job");                      // construit le graphe et le lance

// Source
DataStream<String> s = env.fromSource(kafkaSource,
WatermarkStrategy.noWatermarks(), "nom");

// Transformations sans état
s.map(x -> ...)                                 // 1 → 1
 .filter(x -> ...)                              // 1 → 0 ou 1
 .flatMap((x, out) -> ...)                      // 1 → 0..n (avec une classe si le
type est générique)
 .returns(Types....)                            // préciser le type produit par une
lambda
 .name("nom affiché")                           // nom de l'opérateur dans
```

```
l'interface web
 .setParallelism(3);                              // parallélisme de cet opérateur

// Par clé
s.keyBy(x -> cle)                                 // redistribution par hachage de la
clé
 .reduce((a, b) -> ...)                           // ou .sum(champ), .min(champ),
.max(champ)…

// Fenêtres
s.keyBy(x -> cle)
 .window(TumblingProcessingTimeWindows.of(Duration.ofSeconds(10)))   // temps de
traitement
 .process(new MaProcessWindowFunction());

s.assignTimestampsAndWatermarks(strategie)
 .keyBy(x -> cle)
 .window(TumblingEventTimeWindows.of(Duration.ofMinutes(15)))        // temps de
l'événement
 .process(new MaProcessWindowFunction());

// Sinks
s.print("PREFIXE");
s.sinkTo(kafkaSink);
```

11. Pour aller plus loin

Les notions laissées de côté dans ce TD, traitées dans les séances suivantes :

- **État et tolérance aux pannes :** checkpoints, *state backends*, savepoints, reprise après panne, garanties *exactly-once* de bout en bout avec Kafka ;
- **Traitements, fenêtres et agrégations :** watermarks en détail, événements en retard, fenêtres glissantes et de session, `aggregate`, `ProcessFunction` et minuteries (*timers*), sorties annexes (*side outputs*).

Documentation officielle : Apache Flink 2.2 (rubriques *Concepts* et *Application Development* → *DataStream API*).

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc5/fm4/chap1?rev=1794851666

Last update: **2026/11/16 18:54**