TD1 - De Kafka à Flink : premier traitement de flux

- **Durée :** 4 heures
- **Travail :** binôme
- **Environnement :** Docker Compose, Kafka 4.2 (KRaft), Flink 2.2, Java 21, Maven

1. Contexte

Le TP Kafka vous a fait construire, autour du flux des compteurs de passage (topic `Counters`), plusieurs applications consommatrices écrites « à la main » avec l'API cliente Kafka :

- un worker qui calcule le cumul des entrées et sorties de chaque compteur (`CounterWorker`) ;
- un consumer de statistiques par fenêtres de 15 minutes (`StatsConsumer`) ;
- des mécanismes de reprise : commit des offsets, restauration de l'état depuis un topic compacté, DLQ, transactions.

Ces applications fonctionnent, mais une grande partie de leur code ne traite pas le métier : il gère la répartition du travail, l'état en mémoire, les fenêtres, les reprises après panne.

Votre mission consiste à :

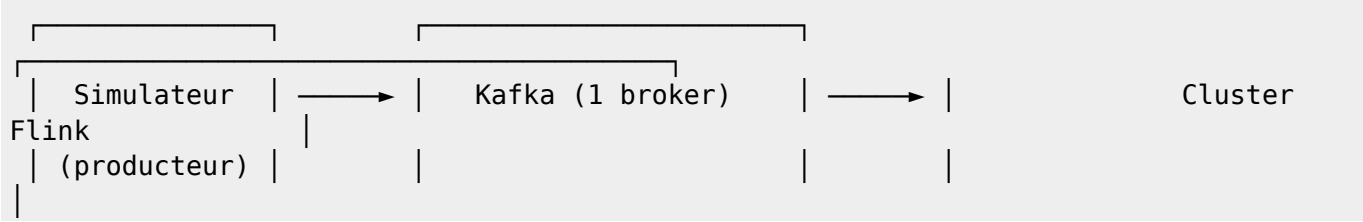
1. réactiver, par la pratique, les notions de Kafka utiles pour la suite ;
2. découvrir **Apache Flink**, un moteur de traitement de flux distribué qui prend en charge ces préoccupations ;
3. lire, exécuter et modifier des jobs Flink simples sur le même flux de compteurs ;
4. observer l'effet de vos modifications sur les résultats et sur l'exécution du job.

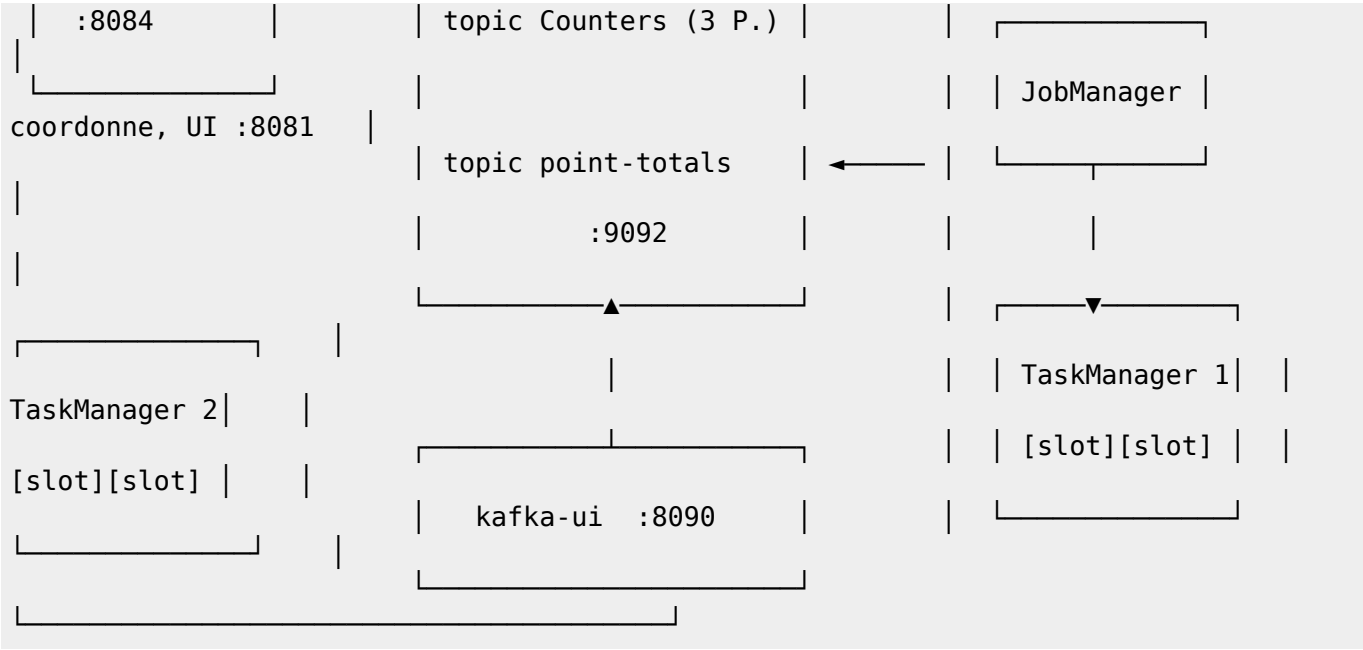
2. Objectifs

À la fin du TD, vous devez être capables de :

- expliquer le rôle de Flink dans une chaîne Kafka → traitement → sortie ;
- lire et modifier un job simple (source → transformations → sink) ;
- distinguer `map`, `filter`, `keyBy` et agrégation ;
- expliquer à grands traits le parallélisme d'un job (sous-tâches, slots, lien avec les partitions Kafka) ;
- expliquer à grands traits ce qu'est une fenêtre, et la différence entre temps de traitement et temps de l'événement ;
- constater et expliquer l'effet d'une modification sur les données ou sur le job.

3. Architecture utilisée





Composant	Rôle	Adresse
Simulateur	Produit les mesures des compteurs dans `Counters` (le même que dans le TP Kafka)	http://localhost:8084
Kafka	1 broker KRaft : topics `Counters` (entrée) et `point-totals` (sortie de la mission 7)	`localhost:9092` (hôte), `kafka-1:19092` (réseau Docker)
kafka-ui	Exploration des topics et des groupes	http://localhost:8090
JobManager Flink	Reçoit les jobs, planifie leur exécution, interface web	http://localhost:8081
TaskManagers Flink (×2)	Exécutent les traitements, 2 slots chacun	—
Flink local (IDE)	Mini-cluster lancé dans votre IDE, avec sa propre interface web	http://localhost:8082



Ce qui change par rapport au TP Kafka : le cluster Kafka est réduit à **un seul broker** (facteur de réplication 1) pour laisser de la mémoire à Flink. La réplication et le failover ont été étudiés dans le TP Kafka ; ils ne sont pas l'objet de ce TD.

Ce qui ne change pas : le simulateur, le format des messages, la clé (`COUNTER-xxxxx-n`), le topic `Counters` à **3 partitions**, kafka-ui et les scripts `live.sh`, `kafka.sh` et `poison.sh`.

4. Préparation de l'environnement

4.1 Prérequis

- Docker et Docker Compose (prévoir environ 5 Go de mémoire pour Docker) ;
- un JDK 21 et Maven 3.9 (ceux du TP Kafka) ;
- un IDE Java (IntelliJ IDEA recommandé) ;
- sous Windows : exécuter les scripts dans **Git Bash**.

```
docker --version
```

```
docker compose version
java -version
mvn -v
```

4.2 Contenu du dossier du TD

```
td1/
├── docker-compose.yml        Kafka, kafka-ui, simulateur, cluster Flink
├── counter-simulator/        le simulateur du TP Kafka
├── flink-jobs/               projet Maven des jobs Flink du TD
│   └── src/main/java/fr/tp/flink/
│       ├── common/           CounterEvent (le même record que dans le TP Kafka),
outils
│       ├── Job01Lecture.java
│       ├── Job02Transformations.java
│       ├── Job03Agregation.java
│       ├── Job04Parallelisme.java
│       └── Job05Fenetres.java
└── scripts/
    ├── live.sh               pilote le flux du simulateur
    ├── kafka.sh              outils Kafka en ligne de commande
    ├── poison.sh             injecte un message invalide
    └── flink.sh              soumet / liste / arrête les jobs sur le cluster Flink
```

4.3 Démarrage

```
docker compose up -d --build
docker compose ps                    # kafka-init doit être « exited (0) », les
autres « running »
docker compose logs kafka-init       # création des topics Counters et point-totals
```

Démarrer le flux : 5 points de comptage × 2 sous-points = 10 compteurs, un message par compteur et par seconde, horloge simulée avançant de 60 s à chaque tick :

```
./scripts/live.sh start 5 2 1000 60
./scripts/live.sh status
```

Vérifier dans kafka-ui (http://localhost:8090) : *Topics* → *Counters* → *Messages*.

Format d'un message (clé `COUNTER-00001-1`) :

```
{"counterId":"COUNTER-00001","pointCode":"P001","pointName":"Point
1","location":"Site 1",
 "subPointNumber":1,"timestamp":"2026-11-15T07:00","entries":5,"exits":3,
 "quarterHourEntries":40,"quarterHourExits":32}
```



L'horloge simulée avance d'une minute par seconde réelle : **une heure simulée défile en une minute**. Le profil par défaut (PEAK) produit des pointes d'affluence entre 8 h et 10 h et entre 17 h et 19 h (heures simulées).

Partie 1 - Retour sur Kafka



Cette partie ne reprend pas tout le TP Kafka. Elle réactive uniquement les trois notions sur lesquelles Flink va s'appuyer : **clé → partition**, **groupe de consommateurs et offsets**, **état d'une application consommatrice**.

Mission 1 - Clé, partition et ordre (15 min)

5. Observer le topic

```
./scripts/kafka.sh kafka-topics --describe --topic Counters
```

Relever :

Élément	Valeur
Nombre de partitions	
Facteur de réplication	
Leader de chaque partition	

Répondre :

1. Pourquoi un facteur de réplication de 1 est-il acceptable ici, et pas en production ?
2. Une partition est-elle lue par plusieurs consommateurs d'un même groupe en même temps ?

6. Clé et partition

Afficher, pour 30 messages, la partition, l'offset et la clé (sans la valeur) :

```
./scripts/kafka.sh kafka-console-consumer --topic Counters --max-messages 30 \
  --formatter-property print.partition=true \
  --formatter-property print.offset=true \
  --formatter-property print.key=true \
  --formatter-property print.value=false
```

Relancer la commande une seconde fois, puis compléter :

Partition	Clés observées
0	
1	
2	

Répondre :

1. Une même clé apparaît-elle toujours dans la même partition ? Quel mécanisme du producteur le garantit ?
2. Les 10 compteurs sont-ils répartis équitablement entre les 3 partitions ? Pourquoi ?
3. Kafka garantit-il l'ordre des messages d'un compteur donné ? Et l'ordre entre deux compteurs de partitions différentes ?



Retenez ce principe « même clé → même partition → même consommateur » : Flink propose le même mécanisme **à l'intérieur d'un traitement**, avec l'opération keyBy (mission 7).

Mission 2 - Groupes de consommateurs et offsets (15 min)

7. Un groupe, des offsets

Consommer 20 messages dans un groupe nommé `rappel`, puis décrire le groupe :

```
./scripts/kafka.sh kafka-console-consumer --topic Counters --group rappel --max-
messages 20 \
  --formatter-property print.partition=true --formatter-property print.offset=true
./scripts/kafka.sh kafka-consumer-groups --describe --group rappel
```

Attendre une dizaine de secondes et décrire à nouveau le groupe.

Répondre :

1. Que représentent `CURRENT-OFFSET`, `LOG-END-OFFSET` et `LAG` ?
2. Pourquoi le lag augmente-t-il alors que personne ne consomme ?
3. Où Kafka stocke-t-il les offsets commités d'un groupe ?

8. Deux membres dans le même groupe

Dans **deux terminaux**, lancer simultanément :

```
./scripts/kafka.sh kafka-console-consumer --topic Counters --group rappel --max-
messages 60 \
  --formatter-property print.partition=true --formatter-property print.value=false
--formatter-property print.key=true
```

Répondre :

1. Quelles partitions chaque terminal reçoit-il ?
2. Combien de membres de ce groupe peuvent travailler en parallèle, au maximum ? Que ferait un 4e membre ?

9. Repositionner un groupe

```
./scripts/kafka.sh kafka-consumer-groups --group rappel --reset-offsets --to-
earliest --topic Counters --dry-run
./scripts/kafka.sh kafka-consumer-groups --group rappel --reset-offsets --to-
earliest --topic Counters --execute
./scripts/kafka.sh kafka-console-consumer --topic Counters --group rappel --max-
messages 5 \
  --formatter-property print.partition=true --formatter-property print.offset=true
```

Répondre :

1. D'où repart le groupe ? Quel intérêt en production (rappel : correction d'un bug dans le TP Kafka) ?
2. Quelle différence entre `auto.offset.reset` et un offset commité ? Quand `auto.offset.reset` est-il utilisé ?

Mission 3 - Ce que vous avez codé à la main (10 min)

10. Relire le CounterWorker

Extrait simplifié de `CounterWorker.processBatch()` (TP Kafka, étape 3) :

```
for (ConsumerRecord<String, String> r : records) {
    nextOffsets.put(new TopicPartition(r.topic(), r.partition()), new
OffsetAndMetadata(r.offset() + 1));
    CounterEvent e;
    try {
        e = CounterEvent.parse(r.value());
    } catch (IllegalArgumentException ex) {
        continue;                                          // message illisible
ignoré
    }
    CounterTotals t = totals.computeIfAbsent(r.key(), k -> new CounterTotals());
```

```
// état en mémoire
    t.add(e, r.partition(), r.offset(), workerId);
    touched.put(r.key(), t);
}
flushTotals(touched);                                    // publication dans
counter-totals
consumer.commitSync(nextOffsets);                        // puis commit des
offsets
```

Rappelez-vous aussi : la méthode `restore()` rechargeait l'état depuis le topic compacté `counter-totals` à chaque rebalance, et le `StatsConsumer` calculait lui-même le début de chaque fenêtre de 15 minutes (`WindowStats.windowStart()`).

Compléter le tableau :

Préoccupation	**Comment le TP Kafka l'a traitée**	**Problèmes rencontrés**
Répartir le travail entre plusieurs instances		
Conserver un état (cumuls)		
Reprendre après une panne sans perte ni doublon		
Regrouper les messages par fenêtre de temps		
Gérer un message invalide		



Toutes ces préoccupations reviennent dans presque toutes les applications de traitement de flux. Un **moteur de traitement de flux** comme Flink les prend en charge : vous décrivez le traitement (lire, transformer, regrouper, agréger, écrire), et le moteur s'occupe de le distribuer, de gérer l'état et de reprendre après une panne. Ce TD aborde la description du traitement et sa distribution. L'état et la tolérance aux pannes feront l'objet d'une séance dédiée.

Partie 2 - Découvrir Flink

Mission 4 - L'architecture de Flink (15 min)

11. Vocabulaire

Terme	**Définition**
Job	Le programme de traitement soumis au cluster ; il tourne en continu tant qu'on ne l'arrête pas
Opérateur	Une étape du traitement : source, `map`, `filter`, agrégation, sink…
DataStream	Un flux (potentiellement infini) d'éléments d'un même type, qui relie les opérateurs
Source / Sink	Opérateur d'entrée (ici : Kafka) / de sortie (ici : affichage ou Kafka)
Parallélisme	Nombre d'instances d'un opérateur qui s'exécutent en même temps
Sous-tâche (*subtask*)	Une instance d'un opérateur ; un opérateur de parallélisme 3 a 3 sous-tâches
JobManager	Reçoit les jobs, découpe le graphe en tâches, les place sur les TaskManagers, supervise
TaskManager	Processus (JVM) qui exécute les sous-tâches
Slot	Une « place » d'exécution dans un TaskManager ; il borne le parallélisme possible

12. Observer le cluster

Ouvrir l'interface web du cluster : http://localhost:8081.

Relever :

Élément	Valeur
Version de Flink	
Nombre de TaskManagers	
Nombre total de slots	
Slots disponibles	

Repérer dans `docker-compose.yml` la variable `FLINK_PROPERTIES` du JobManager et des TaskManagers.

Répondre :

1. Quel paramètre fixe le nombre de slots par TaskManager ? Et le parallélisme par défaut d'un job ?
2. Dans le TP Kafka, le `CounterWorker` était une application Java autonome, lancée en plusieurs exemplaires. Ici, qui lance les exemplaires du traitement ?
3. Comparer : une application consommatrice Kafka est un **programme qui utilise une bibliothèque cliente** ; un job Flink est un **programme confié à un cluster qui l'exécute**. Qu'est-ce que cela change pour le développeur ?

Mission 5 - Premier job : lire le flux (30 min)

13. Lire le code

Ouvrir `flink-jobs` dans l'IDE (projet Maven), puis `Job01Lecture.java` :

```
StreamExecutionEnvironment env = Jobs.environment();

KafkaSource<String> source = KafkaSource.<String>builder()
        .setBootstrapServers(params.bootstrap())
        .setTopics("Counters")
        .setGroupId("td1-job01")
        .setStartingOffsets(OffsetsInitializer.latest())
        .setValueOnlyDeserializer(new SimpleStringSchema())
        .build();

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

DataStream<CounterEvent> events = messages
        .map(CounterEvent::parse)
        .name("Parsing JSON");

events.print("EVT");

env.execute("Job01 - lecture du topic Counters");
```

```
 Kafka « Counters » ──► [ source ] ──► [ map : JSON → CounterEvent ] ──► [ sink :
print ]
```

Répondre :

1. Identifier dans le code la source, la transformation et le sink.
2. Quel est le type des éléments du flux avant et après le `map` ?
3. Que se passerait-il si l'on supprimait la dernière ligne (`env.execute(…)`) ? Pourquoi dit-on que la construction d'un job est « paresseuse » ?
4. Quel réglage du consommateur Kafka correspond à `OffsetsInitializer.latest()` ?

14. Lancer un job

Chaque job peut être lancé de deux façons.

14.1 Depuis l'IDE (mode local)

Le job s'exécute dans un **mini-cluster** Flink démarré dans la JVM de l'IDE. C'est le mode le plus rapide pour mettre au point un traitement.

- IntelliJ : exécuter `main` de `Job01Lecture`. À la première exécution, modifier la configuration de lancement : *Modify options → Add dependencies with "provided" scope to classpath*. Les arguments du job se saisissent dans *Program arguments*.
- Sans IDE, depuis le dossier `flink-jobs` :

```
mvn -q compile exec:exec -Djob=Job01Lecture
mvn -q compile exec:exec -Djob=Job02Transformations -Djob.args="--seuil 18"   #
avec des arguments
```

Les lignes EVT> s'affichent dans la console de l'IDE. L'interface web du mini-cluster est sur http://localhost:8082. Arrêter le job en arrêtant le programme.



Pourquoi le scope `provided` ? Les bibliothèques de Flink sont **fournies par le cluster** : on ne les embarque pas dans le jar du job. Pour exécuter le job hors du cluster, il faut donc les ajouter au classpath. Voir les commentaires du `pom.xml`.

14.2 Sur le cluster Flink

Compiler le jar (`flink-jobs/target/flink-jobs.jar`), puis le soumettre :

```
cd flink-jobs && mvn -q package && cd ..     # ou, sans Maven local : docker
compose run --rm build
./scripts/flink.sh run Job01Lecture
```

```
./scripts/flink.sh list
./scripts/flink.sh logs                          # sorties des TaskManagers (Ctrl+C
pour quitter)
```



Cycle de modification sur le cluster : modifier le code → recompiler le jar → arrêter l'ancien job (`./scripts/flink.sh cancel-all`) → soumettre le nouveau. Un job déjà soumis ne voit pas vos modifications.

Le cluster n'a que **4 slots** : pensez à arrêter les jobs dont vous n'avez plus besoin.

Dans l'interface web, ouvrir le job en cours (*Running Jobs*).

Relever :

Élément	Valeur
Nombre de « boîtes » dans le graphe du job	
Noms des opérateurs dans chaque boîte	
Parallélisme	
Valeurs des colonnes *Records Received* et *Records Sent*	

Répondre :

1. Le code décrit trois opérateurs, mais combien de boîtes le graphe affiche-t-il ? (Indice : Flink **chaîne** dans une même tâche les opérateurs consécutifs de même parallélisme, quand aucune redistribution n'est nécessaire.)
2. À quoi correspond la valeur *Records Received* de la boîte ? Vérifier qu'elle progresse au rythme du simulateur (10 messages par seconde).
3. Pourquoi *Records Sent* reste-t-il à 0 ? (Indice : la sortie du job est un simple affichage, pas un envoi vers une autre boîte ni vers un système externe.)
4. Où les lignes EVT> s'affichent-elles quand le job tourne sur le cluster ? Pourquoi pas dans votre terminal ?

15. Observer et modifier

Pour chaque expérience, noter ce que vous observez **et** l'explication.

#	Action	Question
a	Arrêter le flux (`./scripts/live.sh stop`), attendre 20 s, le relancer	L'état du job change-t-il ? Qu'est-ce que cela dit d'un flux « infini » ?
b	Décommenter `messages.print("BRUT")`, recompiler, resoumettre	Combien de lignes par message ? Les deux sinks reçoivent-ils les mêmes éléments ? Que montre maintenant le graphe ?
c	Remplacer `latest()` par `earliest()`, recompiler, resoumettre	Que lit-on ? Combien de temps pour « rattraper » le flux ?
d	Job en cours : `./scripts/kafka.sh kafka-consumer-groups –list`	Le groupe `td1-job01` apparaît-il ? Comparez avec le groupe `rappel` de la mission 2.
e	Injecter un message invalide : `./scripts/poison.sh`	Quel est l'état du job dans l'interface ? Lire l'onglet *Exceptions*. Le job redémarre-t-il ?



Pour l'expérience **d** : la source Kafka de Flink n'utilise pas le mécanisme des groupes de Kafka pour se répartir les partitions. C'est Flink qui assigne lui-même les partitions à ses sous-tâches, et il ne commite les offsets dans Kafka qu'au moment des *checkpoints*, qui ne sont pas activés ici

(séance « état et tolérance aux pannes »).

Pour l'expérience **e** : comparez avec le `StatsConsumer` du TP Kafka (étape 4.4), qui redémarrait et replantait indéfiniment sur le même message.

Mission 6 - Transformer le flux : map, filter, flatMap (35 min)

16. Lire le code

Ouvrir `Job02Transformations.java` :

```
// flatMap : 1 entrée → 0, 1 ou n sorties (ici : un message invalide est ignoré)
DataStream<CounterEvent> events = messages
        .flatMap(new ParseOrSkip())
        .name("Parsing (invalides ignorés)");

// map : 1 entrée → exactement 1 sortie
DataStream<Passage> passages = events
        .map(e -> new Passage(e.pointCode(), e.counterId(), e.timestamp(),
                e.entries(), e.exits(), e.entries() - e.exits()))
        .name("Projection + solde");

// filter : 1 entrée → 0 ou 1 sortie, du même type
DataStream<Passage> affluence = passages
        .filter(p -> p.entries() >= seuil)
        .name("Affluence >= " + seuil);

affluence.print("AFFLUENCE");
```

```
 Kafka ──► [ flatMap : parsing ] ──► [ map : Passage + solde ] ──► [ filter :
entries >= seuil ] ──► print
```

Compléter :

Opération	Nombre de sorties par entrée	Le type peut-il changer ?	Utilisation dans le job
`map`			
`filter`			
`flatMap`			

17. Exécuter

Arrêter le job précédent (`./scripts/flink.sh cancel-all`), puis :

```
./scripts/flink.sh run Job02Transformations --seuil 15
./scripts/flink.sh logs
```

Répondre :

1. À quels moments (heure **simulée** des messages) les lignes AFFLUENCE apparaissent-elles ? Pourquoi ?
2. Injecter un message invalide (`./scripts/poison.sh`). Le job survit-il ? Où le message ignoré est-il signalé ?
3. Le flux d'entrée compte 600 messages par minute (10 compteurs × 60). Combien de lignes AFFLUENCE obtient-on, approximativement, par minute ? Que mesure l'écart ?

18. Modifier

#	Modification	À observer / expliquer
a	Resoumettre avec `–seuil 10` puis `–seuil 20`	Effet sur le volume de sortie. Pourquoi le seuil est-il un **paramètre** du job plutôt qu'une constante dans le code ?
b	Passer le simulateur en profil constant : `./scripts/live.sh start 5 2 1000 60 CONSTANT` (5 entrées, 4 sorties par message), puis revenir à PEAK	Que devient la sortie avec `–seuil 15` ? Le job a-t-il changé ?
c	Ajouter le champ `location` (site) au record `Passage` et l'afficher	Quelles lignes du code faut-il modifier ?
d	Ne garder que les messages du point `P003` en affluence, en plaçant le `filter` **avant** le `map` (sur `CounterEvent`)	Le résultat est-il le même ? Quel ordre est le plus économe, et pourquoi ?

Question de synthèse :

1. Pour produire sa sortie, l'un de ces trois opérateurs a-t-il besoin de se souvenir des messages précédents ? Qu'est-ce que cela implique pour leur exécution en parallèle ?

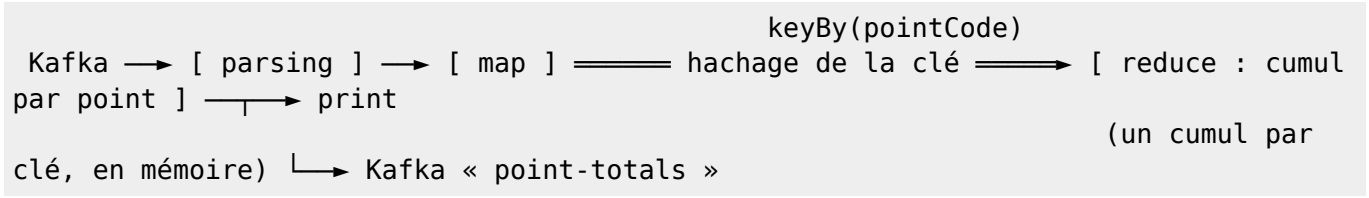
Mission 7 - Regrouper et agréger : keyBy et reduce (35 min)

19. Lire le code

Ouvrir `Job03Agregation.java` : il calcule le cumul des entrées et des sorties **par point de comptage** et le publie dans le topic `point-totals`.

```
DataStream<PointTotal> totals = events
        .map(PointTotal::of)              // chaque événement → un total « unitaire
» (1 message)
        .keyBy(PointTotal::pointCode)     // regroupement logique par point
        .reduce(PointTotal::plus)         // cumul courant de la clé + nouvel
élément → nouveau cumul
        .name("Cumul par point");

totals.print("TOTAL");
totals.sinkTo(sink);                      // KafkaSink : topic point-totals, clé =
pointCode
```



Répondre :

1. Pourquoi `reduce` n'est-il disponible qu'après un `keyBy` ?
2. `reduce` émet-il un seul résultat final, ou un résultat à chaque message reçu ? Pourquoi, sur un flux infini, ne peut-il pas y avoir de « résultat final » ?
3. Où l'opérateur `reduce` conserve-t-il le cumul de chaque point entre deux messages ?

20. Exécuter

```
./scripts/flink.sh cancel-all
./scripts/flink.sh run Job03Agregation
./scripts/flink.sh logs
```

Dans kafka-ui, ouvrir *Topics → point-totals → Messages*.

Relever :

Élément	**Valeur**
Nombre de boîtes dans le graphe du job	
Type de liaison affiché entre les boîtes (`FORWARD`, `HASH`…)	
Records Sent de la première boîte, *Records Received* de la seconde : que transite par la liaison ?	
Partition de `point-totals` contenant les cumuls de `P001`	

Répondre :

1. Pourquoi le `keyBy` coupe-t-il la chaîne d'opérateurs (une boîte de plus qu'en mission 5) ?
2. Pourquoi toutes les valeurs d'un même point arrivent-elles dans la même partition de `point-totals` ?
3. Comparer avec le `CounterWorker` :

	`CounterWorker` (TP Kafka)	**`Job03Agregation` (Flink)**
Lignes de code du traitement	≈ 600	
Qui regroupe les messages par clé ?		
Où est conservé l'état (les cumuls) ?		
Comment répartir la charge sur plusieurs machines ?		

21. Modifier et observer

#	**Action**	**À observer / expliquer**
a	Arrêter puis resoumettre `Job03Agregation`	Les cumuls reprennent-ils là où ils en étaient ? Pourquoi ?

#	Action	À observer / expliquer
b	Resoumettre avec `--start earliest`	Les cumuls sont-ils justes ? Est-ce une bonne solution si le topic contient un an de données ?
c	Cumuler **par site** au lieu de par point. Relancer le flux avec 10 points (`./scripts/live.sh start 10 2 1000 60`) pour avoir 2 sites	Quelles lignes modifier ? Combien de clés distinctes ?
d	Ajouter un second traitement : le nombre de messages reçus **par compteur**, avec l'agrégation prédéfinie `sum` (code ci-dessous)	Pourquoi `.returns(…)` est-il nécessaire ?
e	Question : comment calculer le total **de tous les points** ? Quelle clé utiliser ?	Combien de sous-tâches feraient alors tout le travail ? Conséquence ?

Code pour l'expérience **d** :

```
events.map(e -> Tuple2.of(e.counterId(), 1L))
      .returns(Types.TUPLE(Types.STRING, Types.LONG))   // type des tuples (perdu
par la lambda)
      .keyBy(t -> t.f0)
      .sum(1)                                            // somme du champ n°1 du
tuple
      .print("NB");
```



L'expérience **a** montre la limite du job tel qu'il est écrit : l'état (les cumuls) n'existe qu'en mémoire et disparaît avec le job. Le `CounterWorker` le restaurait à la main depuis `counter-totals`. Flink sait sauvegarder cet état et les positions de lecture Kafka de façon cohérente (*checkpoints*) : c'est l'objet de la séance « état et tolérance aux pannes ».

Mission 8 - Le parallélisme (20 min)

22. Qui traite quoi ?

`Job04Parallelisme` suit le chemin de chaque compteur : partition Kafka → sous-tâche de la source → `keyBy(pointCode)` → sous-tâche qui traite le point. Il n'affiche une ligne que lorsqu'il découvre un nouveau chemin.

```
 Partitions Kafka        Source (p-source)          keyBy(pointCode)
Routage (p-agg)
   P0 ──────────────►  [ source #0 ] ─┐                                ┌──► [ #0
]  P002, P003…
   P1 ──────────────►  [ source #1 ] ─┼═══ hachage de pointCode ═══┼
   P2 ──────────────►  [ source #2 ] ─┘                                └──► [ #1
]  P001, P004…
```

Pour chaque configuration : arrêter le job précédent, soumettre, lire les lignes ROUTE (`./scripts/flink.sh logs`) et le graphe dans l'interface web (cliquer sur une boîte, onglet *SubTasks*).

```
./scripts/flink.sh run Job04Parallelisme                          # tout à 1
./scripts/flink.sh run -p 2 Job04Parallelisme                     # parallélisme
par défaut du job = 2
./scripts/flink.sh run Job04Parallelisme --p-source 3 --p-agg 2
./scripts/flink.sh run Job04Parallelisme --p-source 4 --p-agg 2
./scripts/flink.sh run -p 5 Job04Parallelisme
```

Compléter :

Configuration	**Sous-tâches source**	**Sous-tâches source qui reçoivent des messages**	**Slots occupés**	**Points traités par chaque sous-tâche « Routage »**
défaut (1)				
`-p 2`				
`–p-source 3 –p-agg 2`				
`–p-source 4 –p-agg 2`				
`-p 5`				

Répondre :

1. Avec `–p-source 4`, pourquoi une sous-tâche source ne reçoit-elle rien ? Quel lien avec la règle du consumer group (mission 2) ?
2. Un même point est-il toujours traité par la même sous-tâche après le `keyBy` ? Pourquoi est-ce indispensable pour calculer un cumul juste ?
3. Avec 5 points et 2 sous-tâches, la charge est-elle équilibrée ? Que se passerait-il si un point produisait 10 fois plus de messages que les autres ?
4. Que se passe-t-il avec `-p 5` ? Pourquoi ?
5. Pour traiter plus vite le topic `Counters`, suffit-il d'augmenter le parallélisme du job ? Quelle autre limite faut-il lever ? (Rappel TP Kafka, étape 3.7 : effet d'un changement du nombre de partitions sur la répartition des clés.)

Expérience complémentaire : resoumettre avec `–chainage false` et comparer le graphe. Que coûte la séparation des opérateurs en tâches distinctes ?

Mission 9 - Aperçu des fenêtres (20 min)

23. Pourquoi des fenêtres ?

Le cumul de la mission 7 grossit indéfiniment. Pour suivre l'affluence, on veut plutôt 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 **à la fermeture** de cet intervalle.

```
events.keyBy(CounterEvent::pointCode)
      .window(TumblingProcessingTimeWindows.of(Duration.ofSeconds(10)))   //
fenêtres fixes de 10 s
```

```
      .process(new Summarize(...))                                    // résumé
de chaque fenêtre
```

Mais de quel temps parle-t-on ?

Horloge	**Définition**	**Dans notre flux**
Temps de **traitement**	L'heure de la machine qui exécute l'opérateur	L'heure réelle
Temps de l'**événement**	L'heure à laquelle l'événement s'est produit, portée par le message	Le champ `timestamp` (horloge simulée)

24. Fenêtres en temps de traitement

```
./scripts/flink.sh cancel-all
./scripts/live.sh start 5 2 1000 60
./scripts/flink.sh run Job05Fenetres                        # fenêtres de 10 s, temps
de traitement
./scripts/flink.sh logs
```

Répondre :

1. Combien de messages par point et par fenêtre attendez-vous (2 compteurs par point, 1 message par seconde) ? Vérifier.
2. Relancer le flux deux fois plus vite (`./scripts/live.sh start 5 2 500 60`) : que devient le nombre de messages par fenêtre ?

25. Fenêtres en temps de l'événement

```
./scripts/flink.sh run Job05Fenetres --mode evenement     # fenêtres de 15 min de
temps simulé
```

Dans ce mode, le job indique à Flink comment lire l'horodatage de chaque événement, et jusqu'à quel retard l'attendre (`WatermarkStrategy` dans `Job05Fenetres.java`). La fenêtre `[08:00, 08:15[` se ferme quand le flux montre que le temps des événements a dépassé 08:15.

Répondre :

1. Les fenêtres sont-elles alignées sur l'heure réelle ou sur l'heure simulée ? Combien de messages par fenêtre et par point ?
2. Faire varier la vitesse du flux (`500` puis `1000` ms) : le contenu des fenêtres change-t-il ? Et leur fréquence d'affichage ?
3. Comparer avec le `StatsConsumer` du TP Kafka (fenêtres de 15 min, temps de l'événement) : qui calcule le début et la fin des fenêtres ? qui décide qu'une fenêtre est terminée ?

26. Rejouer l'historique

Soumettre les deux variantes en relisant tout le topic :

```
./scripts/flink.sh cancel-all
./scripts/flink.sh run Job05Fenetres --start earliest
./scripts/flink.sh run Job05Fenetres --start earliest --mode evenement
```

Répondre :

1. En temps de traitement, que contient la première fenêtre ? Ce résultat a-t-il un sens métier ?
2. En temps de l'événement, les fenêtres reconstruites sont-elles identiques à celles calculées en direct ?
3. Conclure : quelle horloge choisir pour des statistiques d'affluence ? Dans quel cas le temps de traitement suffit-il ?



Les *watermarks* (comment Flink mesure l'avancement du temps des événements), les messages en retard et les autres types de fenêtres (glissantes, de session) sont l'objet de la séance « traitements, fenêtres et agrégations ».

Synthèse (15 min)

27. Questions de synthèse

1. Dessiner la chaîne complète du `Job03Agregation` : simulateur, topic, opérateurs Flink (avec leur parallélisme), topic de sortie. Indiquer où les données sont redistribuées.
2. Expliquer en trois phrases le rôle de Flink entre deux topics Kafka.
3. Classer `map`, `filter`, `flatMap`, `keyBy`, `reduce`, `sum`, `window` : lesquels transforment chaque élément indépendamment ? lequel redistribue les données ? lesquels conservent un état ?
4. Quel est le lien entre le nombre de partitions d'un topic, le parallélisme de la source et le nombre de slots ?
5. Citer deux limites du `Job03Agregation` tel qu'il est écrit, qu'un job de production devrait lever.
6. Reprendre le tableau de la mission 3 : pour chaque préoccupation, indiquer ce que Flink prend en charge, et ce qui reste à voir dans les prochaines séances.

Commandes utiles

```
# Environnement
docker compose up -d --build
docker compose ps
docker compose down                       # arrêt (ajouter -v pour effacer aussi les
données Kafka)

# Flux du simulateur
./scripts/live.sh start 5 2 1000 60      # points, sous-points, période (ms), pas
de l'horloge simulée (s) [, PEAK|RANDOM|CONSTANT]
./scripts/live.sh status
./scripts/live.sh stop

# Kafka
```

```
./scripts/kafka.sh kafka-topics --describe --topic Counters
./scripts/kafka.sh kafka-console-consumer --topic point-totals --from-beginning --
max-messages 10 --formatter-property print.key=true
./scripts/kafka.sh kafka-consumer-groups --list
./scripts/kafka.sh kafka-consumer-groups --describe --group rappel
./scripts/poison.sh                      # message invalide dans Counters

# Compilation des jobs
cd flink-jobs && mvn -q package && cd ..       # ou : docker compose run --rm build

# Exécution locale (IDE ou Maven), interface web http://localhost:8082
mvn -q compile exec:exec -Djob=Job03Agregation -Djob.args="--start earliest"

# Cluster Flink, interface web http://localhost:8081
./scripts/flink.sh run Job03Agregation [--start earliest]
./scripts/flink.sh run -p 2 Job04Parallelisme --p-source 3
./scripts/flink.sh list
./scripts/flink.sh cancel <jobId>
./scripts/flink.sh cancel-all
./scripts/flink.sh logs
```

Paramètre de job	Jobs	Effet
`–start latest` ou `–start earliest`	tous	Lire seulement les nouveaux messages (défaut) ou tout le topic
`–seuil N`	Job02	Seuil d'affluence (défaut 15)
`–p-source N`, `–p-agg N`	Job04	Parallélisme de la source / de l'opérateur après le keyBy
`–chainage false`	Job04	Désactive le chaînage des opérateurs
`–mode traitement` ou `–mode evenement`	Job05	Horloge des fenêtres (défaut : traitement)
`–taille N`	Job05	Taille des fenêtres : secondes (traitement, défaut 10) ou minutes (événement, défaut 15)
`–bootstrap hôte:port`	tous	Adresse de Kafka (défaut : `localhost:9092` depuis l'IDE, `kafka-1:19092` sur le cluster)



En cas de problème

- `NoResourceAvailableException` : plus de slots libres → `./scripts/flink.sh cancel-all`.
- Rien ne s'affiche : le flux est-il démarré (`./scripts/live.sh status`) ? Le job lit-il depuis `latest` alors que le flux est arrêté ?
- En local, `ClassNotFoundException` sur une classe Flink : l'option *Add dependencies with "provided" scope to classpath* n'est pas cochée.
- En local, `BindException` sur le port 8082 : un autre job local tourne déjà.
- Sous Windows, une commande `docker compose exec` reçoit un chemin du type `C:/Program Files/Git/opt/…` : utiliser les scripts fournis (ils désactivent la conversion des chemins de Git Bash).

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

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc5/fm4/td1?rev=1794751236

Last update: **2026/11/15 15:00**