Mémento Kafka Streams & KSQLDB

Table des matières

Introduction

Architecture Kafka Streams

```
Input Topics → Kafka Streams Application → Output Topics
                ↓
          Stream Processing Topology
          (KStream, KTable, GlobalKTable)
```

Configuration de base

Java:

```java
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-stream-app");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

StreamsBuilder builder = new StreamsBuilder();
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

Python (kafka-python-ng):

```python
from kafka import KafkaConsumer, KafkaProducer
import json

# Configuration pour Python (pas de Kafka Streams natif)
consumer = KafkaConsumer(
    'input-topic',
    bootstrap_servers=['localhost:9092'],
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)
```

KStream vs KTable

KStream - Flux d'événements

Concept: Séquence infinie d'événements (insert-only)

Java:

```java
KStream<String, String> orders = builder.stream("orders");

// Transformation
KStream<String, OrderValue> processedOrders = orders
    .filter((key, value) -> value.contains("CONFIRMED"))
    .mapValues(value -> new OrderValue(value))
    .selectKey((key, value) -> value.customerId);
```

Exemple d'usage:

- Logs d'événements
- Transactions financières
- Métriques en temps réel

KTable - État actuel

Concept: Table de données changeantes (upsert/delete)

Java:

```
KTable<String, Long> userCounts = builder.table("user-profiles");


// Agrégation
KTable<String, Long> ordersByCustomer = orders
    .groupByKey()
    .count(Materialized.as("orders-by-customer"));
```

Exemple d'usage:

- Profils utilisateurs
- Inventaire produits
- Configurations

Comparaison KStream vs KTable

Aspect	KStream	KTable
Nature	Flux d'événements	État actuel
Données	Append-only	Mutable (upsert/delete)
Temporalité	Tous les événements	Dernière valeur par clé
Stockage	Pas de stockage local	Stockage local (state store)
Exemple	Commandes passées	Solde compte bancaire

Opérations Kafka Streams

Stateless vs Stateful : Concepts fondamentaux

Opérations Stateless

Définition : Opérations qui traitent chaque enregistrement indépendamment, sans conserver d'état entre les enregistrements.

Caractéristiques :

- Pas de stockage local (state store)

- Traitement record par record
- Parallélisation simple
- Pas de fenêtrage nécessaire
- Performance élevée

Exemples : filter, map, mapValues, flatMap, branch, selectKey

Opérations Stateful

Définition : Opérations qui maintiennent un état local pour traiter les enregistrements, nécessitant un stockage persistant.

Caractéristiques :

- Utilisation de state stores (RocksDB par défaut)
- Maintien d'un état entre les enregistrements
- Possibilité de fenêtrage temporel
- Tolérance aux pannes via sauvegarde d'état
- Consommation mémoire/disque

Exemples : groupBy, count, aggregate, reduce, join

Comparaison Stateless vs Stateful

Aspect	Stateless	Stateful
Stockage	Aucun	State stores (RocksDB)
Performance	Très rapide	Plus lente (I/O disque)
Mémoire	Minimale	Proportionnelle aux données
Parallélisation	Parfaite	Limitée par partitioning
Complexité	Simple	Complexe
Fault tolerance	Simple	Backup/restore automatique
Windowing	Non applicable	Supporté
Use cases	Transformation, filtrage	Agrégation, jointure

Exemple illustratif

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> input = builder.stream("input-topic");

// STATELESS - Pas d'état conservé
KStream<String, String> filtered = input
    .filter((k, v) -> v.length() > 10)        // Stateless
    .mapValues(v -> v.toUpperCase())          // Stateless
    .selectKey((k, v) -> v.substring(0, 3));  // Stateless

// STATEFUL - État conservé dans un state store
KTable<String, Long> wordCounts = input
    .flatMapValues(v -> Arrays.asList(v.split(" ")))  // Stateless
    .groupBy((k, v) -> v)                             // Stateful (regroupement)
    .count();                                         // Stateful (comptage)
```

1. Opérations de transformation (Stateless)

Filter

```
// Filtrer les commandes confirmées
KStream<String, Order> confirmedOrders = orders.filter(
    (key, order) -> order.status.equals("CONFIRMED")
);
```

Map/MapValues

```
// Transformer les valeurs
KStream<String, String> upperCase = stream.mapValues(
    value -> value.toUpperCase()
);

// Changer clé et valeur
KStream<String, Integer> mapped = stream.map(
    (key, value) -> KeyValue.pair(value.customerId, value.amount)
);
```

FlatMap

```
// Exploder un événement en plusieurs
KStream<String, String> words = sentences.flatMapValues(
    sentence -> Arrays.asList(sentence.toLowerCase().split(" "))
);
```

Branch

```
// Diviser un stream en plusieurs branches
KStream<String, Order>[] branches = orders.branch(
    (key, order) -> order.amount > 1000,  // Grosses commandes
    (key, order) -> order.amount > 100,   // Commandes moyennes
    (key, order) -> true                  // Petites commandes
);
```

2. Opérations d’agrégation (Stateful)

GroupBy + Count

```
KTable<String, Long> orderCountByStatus = orders
    .groupBy((key, order) -> order.status)
    .count(Materialized.as("order-count-by-status"));
```

Aggregate

```
KTable<String, Double> totalAmountByCustomer = orders
    .groupByKey()
    .aggregate(
        () -> 0.0,  // Initializer
        (key, order, aggregate) -> aggregate + order.amount,  // Aggregator
        Materialized.as("total-by-customer")
    );
```

Windowed Aggregations

```
// Fenêtre glissante de 5 minutes
KTable<Windowed<String>, Long> windowedCounts = orders
    .groupByKey()
    .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
    .count();

// Fenêtre par session
KTable<Windowed<String>, Long> sessionCounts = orders
    .groupByKey()
    .windowedBy(SessionWindows.with(Duration.ofMinutes(30)))
    .count();
```

3. Opérations de jointure

Stream-Stream Join

```
// Join temporel (dans une fenêtre)
KStream<String, OrderEnriched> enrichedOrders = orders
    .join(payments,
        (order, payment) -> new OrderEnriched(order, payment),
        JoinWindows.of(Duration.ofMinutes(5))
    );
```

Stream-Table Join

```
// Enrichissement avec données de référence
KStream<String, OrderWithCustomer> enriched = orders
    .leftJoin(customersTable,
        (order, customer) -> new OrderWithCustomer(order, customer)
    );
```

Table-Table Join

```
// Join entre deux tables
KTable<String, CustomerOrderSummary> joined = customersTable
    .join(orderSummaryTable,
```

```
        (customer, orderSummary) -> new CustomerOrderSummary(customer, orderSummary)
    );
```

4. Cas d'usage par opération

Opération	Cas d'usage	Exemple
filter	Filtrage conditionnel	Commandes > 100€
map	Transformation 1:1	Format de données
flatMap	Transformation 1:N	Tokenisation texte
groupBy	Regroupement	Par client, par région
count	Comptage	Nb commandes/client
aggregate	Calcul métrique	Somme, moyenne, min/max
join	Enrichissement	Commande + données client
branch	Routage conditionnel	Traitement différencié

KSQLDB - Types de requêtes

1. Création de Streams et Tables

Différences fondamentales STREAM vs TABLE dans KSQLDB

Aspect	STREAM	TABLE
Nature des données	Flux d'événements (append-only)	État actuel (upsert/delete)
Clé primaire	Optionnelle	Obligatoire
Topic Kafka sous-jacent	Logs non-compactés	Topics compactés (log.cleanup.policy=compact)
Historique	Garde tous les événements	Garde seulement la dernière valeur
Requêtes Pull	❌ Non supportées	✅ Supportées
Use case	Événements, logs, transactions	Données de référence, profils, configurations

Configuration des Topics pour STREAM vs TABLE

Topics pour STREAMS

```
# Topic standard pour STREAM (log-based)
kafka-topics --create \
  --topic orders-stream \
  --partitions 6 \
  --replication-factor 3 \
  --config cleanup.policy=delete \
  --config retention.ms=604800000  # 7 jours
```

Topics pour TABLES

```
# Topic compacté pour TABLE (état actuel)
kafka-topics --create \
  --topic customers-table \
  --partitions 6 \
  --replication-factor 3 \
  --config cleanup.policy=compact \
  --config min.cleanable.dirty.ratio=0.1 \
  --config segment.ms=86400000  # 1 jour
```

Création détaillée de STREAMS

STREAM basique

```
-- Stream depuis un topic Kafka
CREATE STREAM orders_stream (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    product_id VARCHAR,
    amount DOUBLE,
    order_date BIGINT
) WITH (
    KAFKA_TOPIC='orders',
    VALUE_FORMAT='JSON',
    TIMESTAMP='order_date'
);
```

STREAM avec configuration avancée

```
-- Stream avec toutes les options de configuration
CREATE STREAM advanced_orders_stream (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    product_id VARCHAR,
    amount DOUBLE,
    order_date BIGINT,
    metadata STRUCT<
        source VARCHAR,
        version INTEGER,
        tags ARRAY<VARCHAR>
    >
) WITH (
    KAFKA_TOPIC='orders-advanced',
    VALUE_FORMAT='AVRO',
    VALUE_SCHEMA_ID=123,                    -- ID du schéma Avro
    TIMESTAMP='order_date',
    TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss',
    PARTITIONS=12,                          -- Nombre de partitions
    REPLICAS=3,                             -- Facteur de réplication
    RETENTION_MS=86400000,                  -- Rétention 1 jour
    CLEANUP_POLICY='delete'
);
```

STREAM depuis un pattern de topics

```
-- Stream qui lit depuis plusieurs topics matching un pattern
CREATE STREAM multi_region_orders (
    order_id VARCHAR KEY,
    region VARCHAR,
    amount DOUBLE
) WITH (
    KAFKA_TOPIC='orders-.*',  -- Pattern regex
    VALUE_FORMAT='JSON'
);
```

Création détaillée de TABLES

TABLE basique

```
-- Table depuis un topic compacté
CREATE TABLE customers_table (
    customer_id VARCHAR PRIMARY KEY,
    name VARCHAR,
    email VARCHAR,
    region VARCHAR
) WITH (
    KAFKA_TOPIC='customers',
    VALUE_FORMAT='JSON'
);
```

TABLE avec configuration complète

```
-- Table avec toutes les options
CREATE TABLE advanced_customers_table (
    customer_id VARCHAR PRIMARY KEY,
    profile STRUCT<
        name VARCHAR,
        email VARCHAR,
        phone VARCHAR,
        address STRUCT<
            street VARCHAR,
            city VARCHAR,
            country VARCHAR
        >
    >,
    preferences MAP<VARCHAR, VARCHAR>,
    created_at BIGINT,
    updated_at BIGINT
) WITH (
    KAFKA_TOPIC='customers-advanced',
    VALUE_FORMAT='AVRO',
    VALUE_SCHEMA_ID=456,
```

```
    PARTITIONS=6,
    REPLICAS=3,
    CLEANUP_POLICY='compact',               -- Obligatoire pour TABLE
    MIN_CLEANABLE_DIRTY_RATIO='0.1',       -- Seuil de compaction
    SEGMENT_MS=3600000                      -- Segments d'1 heure
);
```

TABLE depuis une requête (Materialized View)

```
-- Table calculée (matérialisée automatiquement)
CREATE TABLE customer_summary AS
SELECT customer_id,
       COUNT(*) as total_orders,
       SUM(amount) as total_spent,
       AVG(amount) as avg_order_value,
       MAX(order_date) as last_order_date
FROM orders_stream
GROUP BY customer_id
EMIT CHANGES;
```

Formats de sérialisation supportés

Format	Description	Configuration	Use Case
JSON	Format texte lisible	`VALUE_FORMAT='JSON'`	Développement, debug
AVRO	Binaire avec schéma	`VALUE_FORMAT='AVRO'`	Production, performance
PROTOBUF	Protocol Buffers	`VALUE_FORMAT='PROTOBUF'`	Interopérabilité
DELIMITED	CSV-like	`VALUE_FORMAT='DELIMITED'`	Données simples
KAFKA	Format natif Kafka	`VALUE_FORMAT='KAFKA'`	Types primitifs

Exemples par format

```
-- JSON avec schéma inline
CREATE STREAM orders_json (
    order_id VARCHAR KEY,
    data VARCHAR
```

```
) WITH (
    KAFKA_TOPIC='orders-json',
    VALUE_FORMAT='JSON'
);

-- AVRO avec Schema Registry
CREATE STREAM orders_avro (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    amount DOUBLE
) WITH (
    KAFKA_TOPIC='orders-avro',
    VALUE_FORMAT='AVRO',
    VALUE_SCHEMA_FULL_NAME='com.example.Order'
);

-- DELIMITED (CSV-like)
CREATE STREAM orders_csv (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    amount DOUBLE
) WITH (
    KAFKA_TOPIC='orders-csv',
    VALUE_FORMAT='DELIMITED',
    VALUE_DELIMITER=','
);
```

Configuration spécifique des topics

Configuration optimale pour STREAM

```
CREATE STREAM high_throughput_events (
    event_id VARCHAR KEY,
    payload VARCHAR,
    timestamp BIGINT
) WITH (
    KAFKA_TOPIC='events',
```

```
    VALUE_FORMAT='JSON',
    PARTITIONS=24,                    -- Beaucoup de partitions pour parallélisme
    REPLICAS=3,
    RETENTION_MS=259200000,          -- 3 jours
    CLEANUP_POLICY='delete',         -- Suppression après rétention
    SEGMENT_MS=3600000,              -- Segments d'1h pour rotation rapide
    COMPRESSION_TYPE='lz4'           -- Compression pour économiser l'espace
);
```

Configuration optimale pour TABLE

```
CREATE TABLE reference_data (
    id VARCHAR PRIMARY KEY,
    data STRUCT<
        value VARCHAR,
        metadata MAP<VARCHAR, VARCHAR>
    >,
    version INTEGER
) WITH (
    KAFKA_TOPIC='reference',
    VALUE_FORMAT='AVRO',
    PARTITIONS=6,                        -- Moins de partitions (données de référenc
e)
    REPLICAS=3,
    CLEANUP_POLICY='compact',            -- Compaction obligatoire
    MIN_CLEANABLE_DIRTY_RATIO='0.01',    -- Compaction agressive
    SEGMENT_MS=86400000,                 -- Segments journaliers
    COMPRESSION_TYPE='snappy',           -- Compression pour stockage long terme
    DELETE_RETENTION_MS=86400000         -- Garde les tombstones 1 jour
);
```

Types de données supportés dans KSQLDB

Types primitifs

```
CREATE STREAM typed_example (
    id VARCHAR KEY,
```

```
    name VARCHAR,
    age INTEGER,
    salary DOUBLE,
    is_active BOOLEAN,
    raw_data BYTES,
    event_time BIGINT
) WITH (...);
```

Types complexes

```
CREATE STREAM complex_types (
    id VARCHAR KEY,

    -- STRUCT (objet imbriqué)
    address STRUCT<
        street VARCHAR,
        city VARCHAR,
        zipcode INTEGER
    >,

    -- ARRAY (liste)
    tags ARRAY<VARCHAR>,

    -- MAP (dictionnaire)
    attributes MAP<VARCHAR, VARCHAR>,

    -- Combinaisons complexes
    orders ARRAY<STRUCT<
        order_id VARCHAR,
        items ARRAY<STRUCT<
            product VARCHAR,
            quantity INTEGER
        >>
    >>
) WITH (...);
```

Requêtes spécifiques STREAM vs TABLE

Requêtes sur STREAM (Push queries uniquement)

```
-- ✅ Push query - streaming continu
SELECT order_id, amount, customer_id
FROM orders_stream
WHERE amount > 1000
EMIT CHANGES;

-- ✅ Agrégation temporelle
SELECT customer_id,
       COUNT(*) as order_count,
       SUM(amount) as total
FROM orders_stream
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY customer_id
EMIT CHANGES;

-- ❌ Pull query - IMPOSSIBLE sur STREAM
-- SELECT * FROM orders_stream WHERE order_id = '123';  -- ERREUR
```

Requêtes sur TABLE (Push ET Pull queries)

```
-- ✅ Push query - changements en continu
SELECT customer_id, name, region
FROM customers_table
EMIT CHANGES;

-- ✅ Pull query - point-in-time
SELECT * FROM customers_table WHERE customer_id = 'CUST123';

-- ✅ Pull query avec conditions
SELECT name, region
FROM customers_table
WHERE region = 'EU' AND name LIKE 'John%';
```

Surveillance et debugging

Commandes utiles

```
-- Lister tous les streams et tables
SHOW STREAMS;
SHOW TABLES;

-- Décrire la structure
DESCRIBE orders_stream;
DESCRIBE EXTENDED customers_table;  -- Infos détaillées

-- Voir les topics sous-jacents
SHOW TOPICS;

-- Monitoring des requêtes
SHOW QUERIES;
EXPLAIN <query_id>;

-- Voir les données (échantillon)
PRINT 'orders' FROM BEGINNING LIMIT 10;
```

Métriques importantes

```
# Vérifier la configuration du topic
kafka-topics --describe --topic orders --bootstrap-server localhost:9092

# Surveiller le lag
kafka-consumer-groups --bootstrap-server localhost:9092 --group _confluent-ksql-defaul
t_query_CTAS_* --describe
```

Bonnes pratiques configuration

Type	Recommandation	Explication
STREAM	`cleanup.policy=delete`	Garde l'historique complet
STREAM	`retention.ms` court	Évite l'accumulation excessive

Type	Recommandation	Explication
STREAM	Nombreuses partitions	Parallélisme pour high throughput
TABLE	`cleanup.policy=compact`	Garde seulement l'état actuel
TABLE	`min.cleanable.dirty.ratio` faible	Compaction fréquente
TABLE	Moins de partitions	Données de référence, moins de volume
Les deux	`replicas=3` minimum	Haute disponibilité
Les deux	Compression activée	Économie stockage/bande passante

CREATE STREAM

```
-- Stream depuis un topic Kafka
CREATE STREAM orders_stream (
    order_id VARCHAR KEY,
    customer_id VARCHAR,
    product_id VARCHAR,
    amount DOUBLE,
    order_date BIGINT
) WITH (
    KAFKA_TOPIC='orders',
    VALUE_FORMAT='JSON',
    TIMESTAMP='order_date'
);
```

CREATE TABLE

```
-- Table depuis un topic compacté
CREATE TABLE customers_table (
    customer_id VARCHAR PRIMARY KEY,
    name VARCHAR,
    email VARCHAR,
    region VARCHAR
) WITH (
    KAFKA_TOPIC='customers',
    VALUE_FORMAT='JSON'
);
```

2. Requêtes de sélection

SELECT basique

```
-- Sélection simple
SELECT order_id, customer_id, amount
FROM orders_stream
WHERE amount > 100
EMIT CHANGES;
```

SELECT avec fenêtrage

```
-- Agrégation par fenêtre temporelle
SELECT customer_id,
       COUNT(*) as order_count,
       SUM(amount) as total_amount
FROM orders_stream
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY customer_id
EMIT CHANGES;
```

3. Jointures KSQLDB

Stream-Table Join

```
-- Enrichir commandes avec données client
SELECT o.order_id,
       o.amount,
       c.name as customer_name,
       c.region
FROM orders_stream o
LEFT JOIN customers_table c ON o.customer_id = c.customer_id
EMIT CHANGES;
```

Stream-Stream Join

```sql
-- Corréler commandes et paiements
SELECT o.order_id,
       o.amount as order_amount,
       p.amount as payment_amount
FROM orders_stream o
INNER JOIN payments_stream p
    WITHIN 1 HOUR
    ON o.order_id = p.order_id
EMIT CHANGES;
```

4. Agrégations avancées

GROUP BY avec HAVING

```sql
-- Clients avec plus de 5 commandes par jour
SELECT customer_id,
       COUNT(*) as daily_orders,
       SUM(amount) as daily_total
FROM orders_stream
WINDOW TUMBLING (SIZE 1 DAY)
GROUP BY customer_id
HAVING COUNT(*) > 5
EMIT CHANGES;
```

Agrégations multiples

```sql
-- Statistiques complètes par produit
SELECT product_id,
       COUNT(*) as order_count,
       SUM(amount) as total_revenue,
       AVG(amount) as avg_order_value,
       MIN(amount) as min_order,
       MAX(amount) as max_order
FROM orders_stream
WINDOW TUMBLING (SIZE 1 HOUR)
```

```
GROUP BY product_id
EMIT CHANGES;
```

5. Types de fenêtrage KSQLDB

Type de fenêtre	Syntaxe	Usage
TUMBLING	`WINDOW TUMBLING (SIZE 1 HOUR)`	Fenêtres fixes non-chevauchantes
HOPPING	`WINDOW HOPPING (SIZE 1 HOUR, ADVANCE BY 15 MINUTES)`	Fenêtres fixes chevauchantes
SESSION	`WINDOW SESSION (60 SECONDS)`	Fenêtres basées sur l'activité

6. Materialized Views

```
-- Créer une vue matérialisée
CREATE TABLE customer_order_summary AS
SELECT customer_id,
       COUNT(*) as total_orders,
       SUM(amount) as lifetime_value,
       AVG(amount) as avg_order_value
FROM orders_stream
GROUP BY customer_id
EMIT CHANGES;
```

7. Types de requêtes KSQLDB

Type	Description	Exemple
Push Query	Streaming continu	`SELECT * FROM stream EMIT CHANGES`
Pull Query	Point-in-time	`SELECT * FROM table WHERE key='123'`
Persistent Query	CREATE STREAM/TABLE AS	Sauvegarde résultat
Transient Query	Temporaire	Test et développement

Comparaison des approches

Tableau comparatif complet

Critère	Kafka Consumer Library	Kafka Streams	KSQLDB
Complexité d'implémentation	Haute	Moyenne	Faible
Flexibilité	Maximale	Haute	Limitée
Courbe d'apprentissage	Élevée	Moyenne	Faible
Gestion d'état	Manuelle	Automatique	Automatique
Exactement-une-fois	Manuelle	Automatique	Automatique
Rebalancing	Manuel	Automatique	Automatique
Tolérance aux pannes	Manuelle	Automatique	Automatique
Scaling	Manuel	Automatique	Automatique
Jointures	Code custom	API native	SQL natif
Agrégations	Code custom	API native	SQL natif
Windowing	Code custom	API native	SQL natif
Interface	Code Java/Python	DSL Java	SQL
Performance	Optimale	Très bonne	Bonne
Cas d'usage	Logique métier complexe	Stream processing	Analytics, prototypage

Quand utiliser chaque approche

Approche	Utiliser quand	Ne pas utiliser quand
Consumer Library	• Logique très spécifique • Performance critique • Intégration complexe	• Besoins standards • Équipe sans expertise Kafka
Kafka Streams	• Stream processing • Transformations complexes • État local nécessaire	• Requêtes SQL simples • Prototypage rapide
KSQLDB	• Analytics temps réel • Prototypage • Équipe orientée SQL	• Logique métier complexe • Performance extrême

Conversion Consumer Library → Kafka Streams

Exemple: Compteur de mots

Version Consumer Library (Java)

```
public class WordCountConsumer {
    private Map<String, Long> wordCounts = new ConcurrentHashMap<>();

    public void runConsumer() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "word-count-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.cla
ss);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.c
lass);

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("input-topic"));

        // Producer pour les résultats
        Properties prodProps = new Properties();
        prodProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        prodProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.cla
ss);
        prodProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, LongSerializer.cla
ss);

        KafkaProducer<String, Long> producer = new KafkaProducer<>(prodProps);

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMil
lis(100));

                for (ConsumerRecord<String, String> record : records) {
                    // Processing manuel
                    String[] words = record.value().toLowerCase().split(" ");
```

```
                    for (String word : words) {
                        // Mise à jour du compteur local
                        wordCounts.compute(word, (k, v) -> (v == null) ? 1L : v + 1L);

                        // Envoi du résultat
                        producer.send(new ProducerRecord<>(
                            "word-counts", word, wordCounts.get(word)
                        ));
                    }
                }

                // Commit manuel des offsets
                consumer.commitSync();
            }
        } finally {
            consumer.close();
            producer.close();
        }
    }
}
```

Version Kafka Streams (Java)

```
public class WordCountStreams {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "word-count-streams");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getCla
ss());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getC
lass());

        StreamsBuilder builder = new StreamsBuilder();

        // Pipeline de traitement déclaratif
```

```
        KStream<String, String> textLines = builder.stream("input-topic");

        KTable<String, Long> wordCounts = textLines
            // Transformation en mots
            .flatMapValues(line -> Arrays.asList(line.toLowerCase().split(" ")))
            // Regroupement par mot
            .groupBy((key, word) -> word)
            // Comptage avec état persistant
            .count(Materialized.as("word-counts-store"));

        // Export vers topic de sortie
        wordCounts.toStream().to("word-counts");

        // Démarrage de l'application
        KafkaStreams streams = new KafkaStreams(builder.build(), props);

        // Graceful shutdown
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));

        streams.start();
    }
}
```

Avantages de la conversion

Aspect	Consumer Library	Kafka Streams
Lignes de code	~80 lignes	~25 lignes
Gestion d'état	HashMap en mémoire (volatile)	State store persistant
Fault tolerance	Perte des données en cas de crash	Restauration automatique
Scaling	Partition manuelle	Rebalancing automatique
Exactly-once	Complexe à implémenter	Configuration simple
Testing	Mock consumer/producer	TopologyTestDriver

Pattern de conversion typique

1. **Consumer poll loop → Stream processing pipeline**
2. **Manual offset management → Automatic commit**
3. **In-memory state → Persistent state stores**
4. **Manual error handling → Built-in retry/DLQ**
5. **Custom serialization → Serde configuration**

Version KSQLDB (SQL)

```
-- 1. Créer le stream d'entrée
CREATE STREAM text_stream (
    line VARCHAR
) WITH (
    KAFKA_TOPIC='input-topic',
    VALUE_FORMAT='DELIMITED'
);

-- 2. Exploder les lignes en mots (équivalent flatMapValues)
CREATE STREAM words_stream AS
SELECT EXPLODE(SPLIT(LCASE(line), ' ')) AS word
FROM text_stream
EMIT CHANGES;

-- 3. Compter les mots (équivalent groupBy + count)
CREATE TABLE word_counts AS
SELECT word,
       COUNT(*) as count
FROM words_stream
GROUP BY word
EMIT CHANGES;

-- Alternative avec fenêtrage (comptage par heure)
CREATE TABLE word_counts_hourly AS
SELECT word,
       COUNT(*) as count
FROM words_stream
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY word
```

```
EMIT CHANGES;

-- Requête pour visualiser les résultats
SELECT * FROM word_counts EMIT CHANGES;
```

Comparaison des 3 approches

Aspect	Consumer Library	Kafka Streams	KSQLDB
Code required	~80 lignes Java	~25 lignes Java	~15 lignes SQL
Compétences	Java + Kafka API	Java + Streams DSL	SQL
Flexibilité	Maximale	Haute	Limitée au SQL
Déploiement	JAR custom	JAR Streams app	Requêtes SQL
État	HashMap volatile	State store persistant	State store persistant
Scaling	Manuel	Automatique	Automatique
Monitoring	Custom	Streams metrics	KSQLDB metrics

Conversions progressives : un même cas d'usage, 3 implémentations

Détection de fraude temps réel

1. Consumer Library (Java) - Maximum de contrôle

```
public class FraudDetectionConsumer {
    private Map<String, UserActivity> userSessions = new ConcurrentHashMap<>();
    private static final double FRAUD_THRESHOLD = 1000.0;

    public void detectFraud() {
        // Configuration consumer/producer
        // Poll loop avec logique métier complexe
        for (ConsumerRecord<String, Transaction> record : records) {
            Transaction txn = record.value();

            // Logique de détection custom
```

```
            UserActivity activity = userSessions.computeIfAbsent(
                txn.userId, k -> new UserActivity()
            );

            activity.addTransaction(txn);

            if (activity.getTotalAmount() > FRAUD_THRESHOLD) {
                // Alerte fraude
                fraudProducer.send(new ProducerRecord<>("fraud-alerts",
                    txn.userId, new FraudAlert(txn, activity)));
            }
        }
    }
}
```

2. Kafka Streams (Java) - Équilibre puissance/simplicité

```
public class FraudDetectionStreams {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, Transaction> transactions = builder.stream("transactions");

        // Détection avec fenêtrage
        KTable<String, Double> userAmounts = transactions
            .groupByKey()
            .windowedBy(TimeWindows.of(Duration.ofMinutes(10)))
            .aggregate(
                () -> 0.0,
                (key, txn, aggregate) -> aggregate + txn.amount,
                Materialized.as("user-amounts")
            );

        // Détection de fraude
        KStream<String, FraudAlert> fraudAlerts = userAmounts
            .toStream()
            .filter((windowed, amount) -> amount > FRAUD_THRESHOLD)
```

```
            .map((windowed, amount) -> KeyValue.pair(
                windowed.key(),
                new FraudAlert(windowed.key(), amount)
            ));

        fraudAlerts.to("fraud-alerts");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();
    }
}
```

3. KSQLDB (SQL) - Simplicité maximale

```
-- Stream des transactions
CREATE STREAM transactions_stream (
    user_id VARCHAR KEY,
    amount DOUBLE,
    merchant VARCHAR,
    timestamp BIGINT
) WITH (
    KAFKA_TOPIC='transactions',
    VALUE_FORMAT='JSON',
    TIMESTAMP='timestamp'
);

-- Agrégation par utilisateur (fenêtre de 10 minutes)
CREATE TABLE user_spending AS
SELECT user_id,
       SUM(amount) as total_amount,
       COUNT(*) as transaction_count
FROM transactions_stream
WINDOW TUMBLING (SIZE 10 MINUTES)
GROUP BY user_id
HAVING SUM(amount) > 1000.0  -- Seuil de fraude
EMIT CHANGES;
```

```
-- Stream d'alertes de fraude
CREATE STREAM fraud_alerts AS
SELECT user_id,
       total_amount,
       transaction_count,
       'FRAUD_DETECTED' as alert_type
FROM user_spending
EMIT CHANGES;

-- Export vers topic d'alertes
CREATE STREAM fraud_alerts_output
WITH (KAFKA_TOPIC='fraud-alerts', VALUE_FORMAT='JSON') AS
SELECT * FROM fraud_alerts
EMIT CHANGES;
```

Exemple Python (simulation Kafka Streams)

```
from kafka import KafkaConsumer, KafkaProducer
from collections import defaultdict
import json

class PythonStreamProcessor:
    def __init__(self):
        self.word_counts = defaultdict(int)
        self.consumer = KafkaConsumer(
            'input-topic',
            bootstrap_servers=['localhost:9092'],
            value_deserializer=lambda x: x.decode('utf-8')
        )
        self.producer = KafkaProducer(
            bootstrap_servers=['localhost:9092'],
            value_serializer=lambda x: json.dumps(x).encode('utf-8')
        )

    def process_stream(self):
        for message in self.consumer:
            # Traitement similaire à Kafka Streams
```

```
            words = message.value.lower().split()

            for word in words:
                self.word_counts[word] += 1

                # Publication du résultat
                self.producer.send(
                    'word-counts',
                    key=word.encode('utf-8'),
                    value={'word': word, 'count': self.word_counts[word]}
                )

# Simulation d'opérations stateful en Python
class StatefulProcessor:
    def __init__(self):
        self.state_stores = {}  # Simulation des state stores

    def aggregate(self, key, value, aggregator_func):
        """Simulation d'une agrégation stateful"""
        current_state = self.state_stores.get(key, None)
        new_state = aggregator_func(key, value, current_state)
        self.state_stores[key] = new_state
        return new_state

    def windowed_count(self, key, window_size_ms):
        """Simulation de comptage avec fenêtre"""
        import time
        current_time = int(time.time() * 1000)
        window_key = f"{key}_{current_time // window_size_ms}"

        count = self.state_stores.get(window_key, 0) + 1
        self.state_stores[window_key] = count
        return count
```

Ressources et bonnes pratiques

Configuration de production

```
// Optimisations pour la production
props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);
props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, StreamsConfig.EXACTLY_ONCE_V2);
props.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG, StreamsConfig.OPTIMIZE);
props.put(StreamsConfig.STATE_DIR_CONFIG, "/var/kafka-streams");
```

Monitoring

```
// Métriques importantes à surveiller
- kafka.streams:type=stream-metrics,client-id=*
- kafka.streams:type=stream-thread-metrics,thread-id=*
- kafka.streams:type=task-metrics,thread-id=*,task-id=*
```

Tests

```
// Testing avec TopologyTestDriver
TopologyTestDriver testDriver = new TopologyTestDriver(topology, props);
TestInputTopic<String, String> inputTopic = testDriver.createInputTopic("input");
TestOutputTopic<String, Long> outputTopic = testDriver.createOutputTopic("output");

inputTopic.pipeInput("hello world");
assertEquals(Long.valueOf(1L), outputTopic.readValue());
```