

Università degli Studi di Roma “Tor Vergata”

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Kafka Streams: Hands-on Session

A.A. 2017/18

Matteo Nardelli

Laurea Magistrale in
Ingegneria Informatica - II anno

The reference Big Data stack

High-level Interfaces

Data Processing

Data Storage

Resource Management

Support / Integration

Kafka Streams:

- Kafka Streams is a **client library** for processing and analyzing data stored in Kafka
- Supports fault-tolerant local state
- Supports exactly-once processing semantics
- Employs **one-record-at-a-time** processing
- Offers necessary stream processing primitives:
 - high-level Streams DSL
 - low-level Processor API

Read more

- <https://kafka.apache.org/documentation/streams>
- <https://kafka.apache.org/11/documentation/streams/core-concepts>
- <https://kafka.apache.org/11/documentation/streams/developer-guide/dsl-api.html>
- <https://kafka.apache.org/11/documentation/streams/developer-guide/processor-api.html>

Kafka Streams: Main Concepts

Kafka Stream API:

- transforms and enriches data;
- supports per-record stream processing with millisecond latency (no micro-batching);
- supports stateless processing, stateful processing, windowing operations

Write standard Java applications to process data in real time:

- no separate cluster required
- elastic, highly scalable, fault-tolerant
- supports exactly once semantics as of 0.11.0

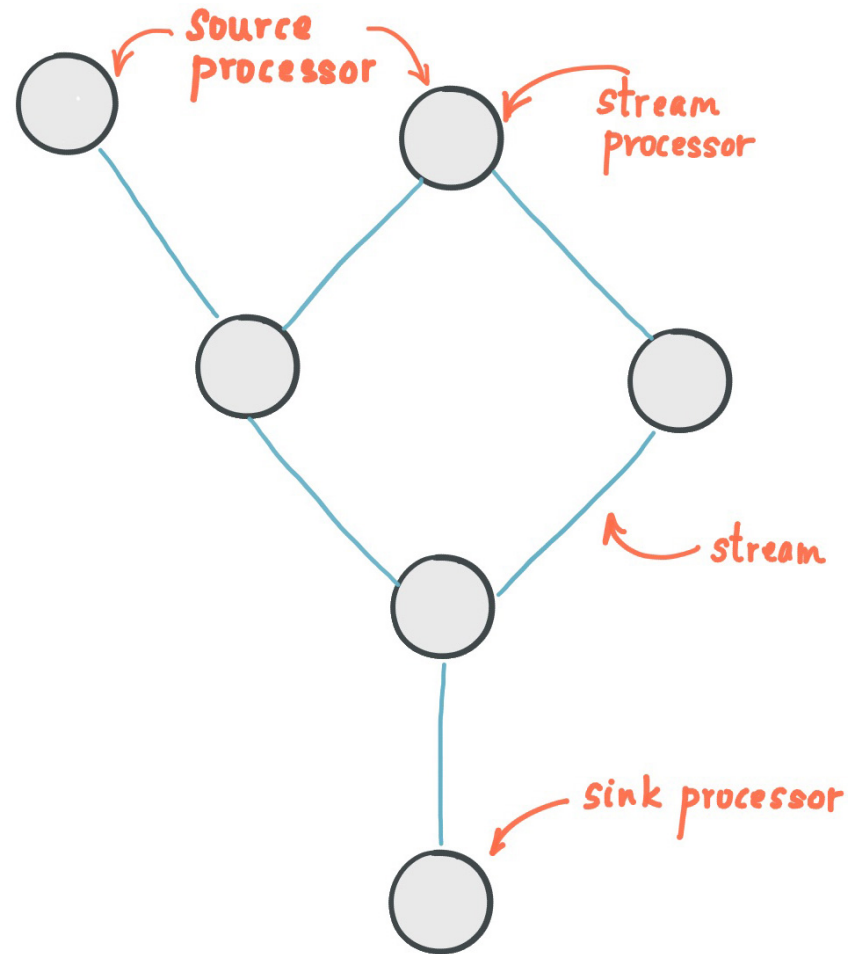
The Kafka Stream API interacts with a Kafka cluster

The application does not run directly on Kafka brokers

Kafka Streams: Topology

- A processor topology is a graph of stream processors (nodes) that are connected by streams (edges).
- **Stream**: unbounded, continuously updating data set. A stream is an ordered, replayable, and fault-tolerant sequence of immutable key-value pairs (data records).
- A **stream processor** is a node in the processor topology:
 - **Source Processor** produces an input stream to its topology **from one or multiple Kafka topics** by consuming records from these topics and forwarding them to its down-stream processors. It has not upstream processors.
 - **Sink Processor** sends any received records from its up-stream processors **to a Kafka topic**. It has no down-stream processors.

Kafka Streams: Topology



PROCESSOR TOPOLOGY

Kafka Streams: State

Kafka Streams provides so-called state stores:

- Data stores can be used to store and query data
- Every task in Kafka Streams embeds one or more state stores that can be accessed via APIs to store and query data required for processing
- These state stores can either be a persistent key-value store, an in-memory hashmap, or another convenient data structure
- Kafka Streams offers fault-tolerance and automatic recovery for local state stores

Kafka Streams: KStreams and KTables

- **KStream**: an abstraction of a **record** stream, where each data record represents a self-contained datum in the unbounded data set. It contains data from a single partition.
- **KTable**: an abstraction of a **changelog** stream (i.e., evolving facts), where each value represents an update of the key value; if the key does not exist, it is created. It contains data from a single partition.
- **GlobalKTable**: like a KTable, but populated with data from all partitions of the topic.

Reference stream:

```
("alice", 1) --> ("alice", 3)
```

Sum the values per user:

- with KStream, it would return 4 for alice.
- with KTable, it would return 3 for alice, because the second data record would be considered an update of the previous record.

Streams DSL (Domain Specific Language)

- A KStream represents a partitioned record stream.
- The local KStream instance of every application instance will be populated with data from only a subset of the partitions of the input topic.
- Collectively, across all application instances, all input topic partitions are read and processed.

```
KStream<String, Long> wordCounts = builder.stream(  
    "kafka-topic",           /* input topic */  
    Consumed.with(Serdes.String(), /* key serde */  
                  Serdes.Long()    /* value serde */  
);
```

SerDes:

- specifies how to serialize/deserialize the key and value data store in a Kafka topic

Stateless Transformations

- `branch()`: Branch (or split) a KStream based on the supplied predicates into one or more KStream instances
- `filter()`: Evaluates a boolean function for each element and retains those for which the function returns true. `filterNot()` drops data for which the function returns true.
- `flatMap()`: Takes one record and produces zero, one, or more records. You can modify the record keys and values, including their types.
- `foreach()`: Terminal operation. Performs a stateless action on each record.
- `groupByKey()`: Groups the records by the existing key
- `groupBy()`: Groups the records by a new key, which may be of a different key type. When grouping a table, you may also specify a new value and value type
- `map()`: Takes one record and produces one record. You can modify the record key and value, including their types.

Stateless Transformations

Table To Stream:

- `(Ktable).toStream()`: Get the changelog stream of this table

Writing back to Kafka:

- `to()`: it sends data to a Kafka topic (the data key determines the topic partition). It requires to explicitly provide *serdes* when the key and/or value types of the KStream do not match the configured default SerDes. To specify the SerDes explicitly, we can use the Produced class.

Stateful Transformations

Stateful transformations include: Aggregating, Joining, Windowing, and Custom transformation

Aggregating data

After records are grouped by key via `groupByKey` or `groupBy`, they can be aggregated via an operation such as `reduce`.

Aggregations are key-based operations, i.e., they always operate over records of the same key.

- `aggregate()`: Aggregates the values of records by the grouped key. Aggregating is a generalization of `reduce` and allows, e.g., the aggregate value to have a different type than the input values
- `count()`: counts the number of records by the grouped key
- `reduce()`: Combines the values of records by the grouped key

Stateful Transformations

Windowing

Windowing lets you control how to group records that have the same key for stateful operations such as aggregations or joins into so-called windows. Windows are tracked per record key.

- Tumbling window (window size = slide interval)
`TimeWindows.of(windowSizeMs);`
- Sliding and hopping time window:
`TimeWindows.of(windowSizeMs).advanceBy(advanceMs);`
- Session window, that is created after an inactivity gap:
`SessionWindows.with(TimeUnit.MINUTES.toMillis(5));`