KStream

KStream

A **KStream** is an abstraction of a **record stream**, where each data record represents a self-contained datum in the unbounded data set.

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

stream processing application were to sum the values per user, it would return 4 for alice.

Processor topologies

Steps:

Specify one or more input streams that are read from Kafka topics.
Compose transformations on these streams.
Write the resulting output streams back to Kafka topics, or expose the processing results of your application directly to other applications through Kafka Streams Interactive Queries (e.g., via a REST API).

Reading from Kafka

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.KStream;

StreamsBuilder builder = new StreamsBuilder();

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

Transform a stream

Stateless and Stateful

Stateless transformations

Stateless transformations do not require state for processing and they do not require a state store associated with the stream processor.

- KStream → KStream[]

```
KStream<String, Long> stream = ...;
KStream<String, Long>[] branches = stream.branch(
    (key, value) -> key.startsWith("A"), /* first predicate  */
    (key, value) -> key.startsWith("B"), /* second predicate */
    (key, value) -> true                 /* third predicate  */
  );

// KStream branches[0] contains all records whose keys start with "A"
// KStream branches[1] contains all records whose keys start with "B"
// KStream branches[2] contains all other records

// Java 7 example: cf. `filter` for how to create `Predicate` instances
```

Predicates are evaluated in order. A record is placed to one and only one output stream on the first match

Filter

Evaluates a boolean function for each element and retains those for which the function returns true

```
KStream<String, Long> stream = ...;

// A filter that selects (keeps) only positive numbers
// Java 8+ example, using lambda expressions
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);

// Java 7 example
KStream<String, Long> onlyPositives = stream.filter(
    new Predicate<String, Long>() {
      @Override
      public boolean test(String key, Long value) {
        return value > 0;
      }
    });
```

Inverse Filter

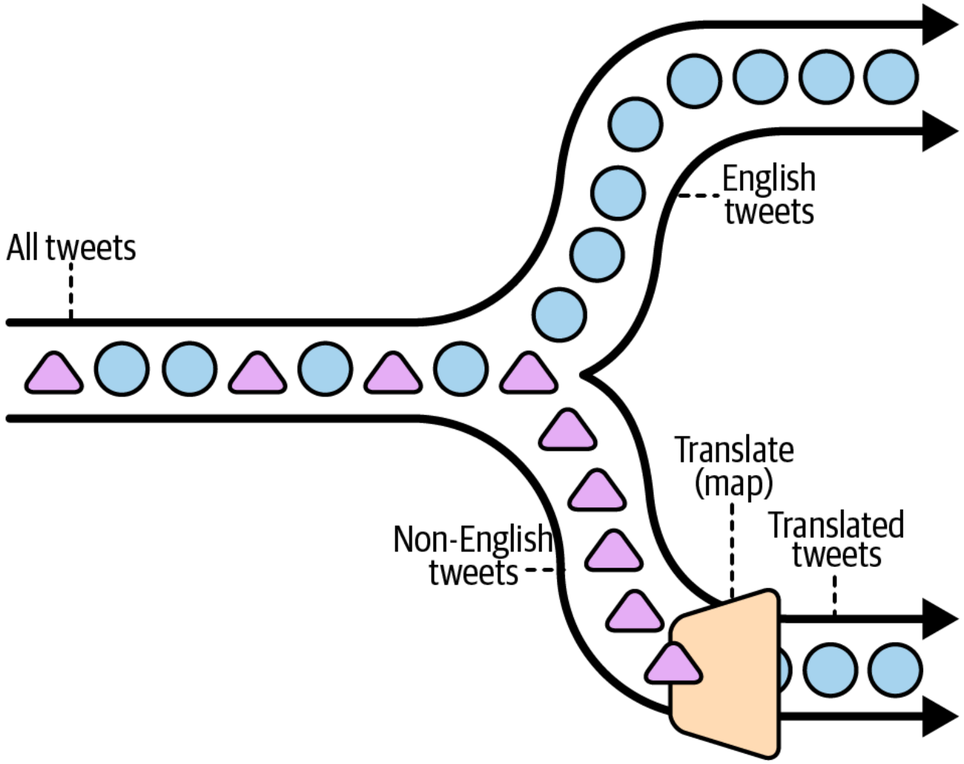
Evaluates a boolean function for each element and drops those for which the function returns true.

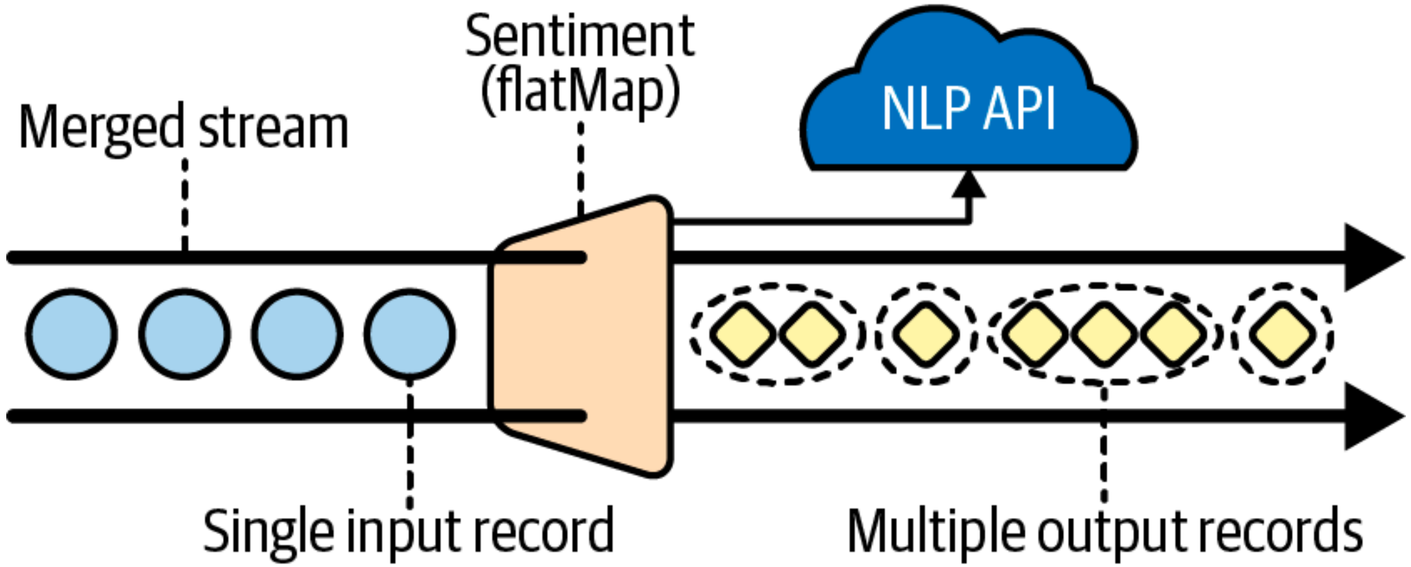
```
KStream<String, Long> stream = ...;

// An inverse filter that discards any negative numbers or zero
// Java 8+ example, using lambda expressions
KStream<String, Long> onlyPositives = stream.filterNot((key, value) -> value <= 0);

// Java 7 example
KStream<String, Long> onlyPositives = stream.filterNot(
    new Predicate<String, Long>() {
      @Override
      public boolean test(String key, Long value) {
        return value <= 0;
      }
    });
```

Map vs FlatMap





Map

Takes one record and produces one record. You can modify the record key and value, including their types
Marks the stream for data re-partitioning

```
KStream<byte[], String> stream = ...;

// Java 8+ example, using lambda expressions
// Note how we change the key and the key type (similar to `selectKey`)
// as well as the value and the value type.
KStream<String, Integer> transformed = stream.map(
    (key, value) -> KeyValue.pair(value.toLowerCase(), value.length()));

// Java 7 example
KStream<String, Integer> transformed = stream.map(
    new KeyValueMapper<byte[], String, KeyValue<String, Integer>>() {
      @Override
      public KeyValue<String, Integer> apply(byte[] key, String value) {
        return new KeyValue<>(value.toLowerCase(), value.length());
      }
    });
```

FlatMap

Takes one record and produces zero, one, or more records. You can modify the record keys and values, including their types

```
KStream<Long, String> stream = ...;
KStream<String, Integer> transformed = stream.flatMap(
     // Here, we generate two output records for each input record.
     // We also change the key and value types.
     // Example: (345L, "Hello") -> ("HELLO", 1000), ("hello", 9000)
    (key, value) -> {
      List<KeyValue<String, Integer>> result = new LinkedList<>();
      result.add(KeyValue.pair(value.toUpperCase(), 1000));
      result.add(KeyValue.pair(value.toLowerCase(), 9000));
      return result;
    }
  );

// Java 7 example: cf. `map` for how to create `KeyValueMapper` instances
```

Applying a grouping or a join after flatMap will result in re-partitioning of the records. If possible use flatMapValues instead, which will not cause data re-partitioning.

Peek

Performs a stateless action on each record, and returns an unchanged stream

```
KStream<byte[], String> stream = ...;

// Java 8+ example, using lambda expressions
KStream<byte[], String> unmodifiedStream = stream.peek(
    (key, value) -> System.out.println("key=" + key + ", value=" + value));

// Java 7 example
KStream<byte[], String> unmodifiedStream = stream.peek(
    new ForeachAction<byte[], String>() {
      @Override
      public void apply(byte[] key, String value) {
        System.out.println("key=" + key + ", value=" + value);
      }
    });
```

Lab: DSL - Transform a stream of events – 60 Minutes