KStream

KStream

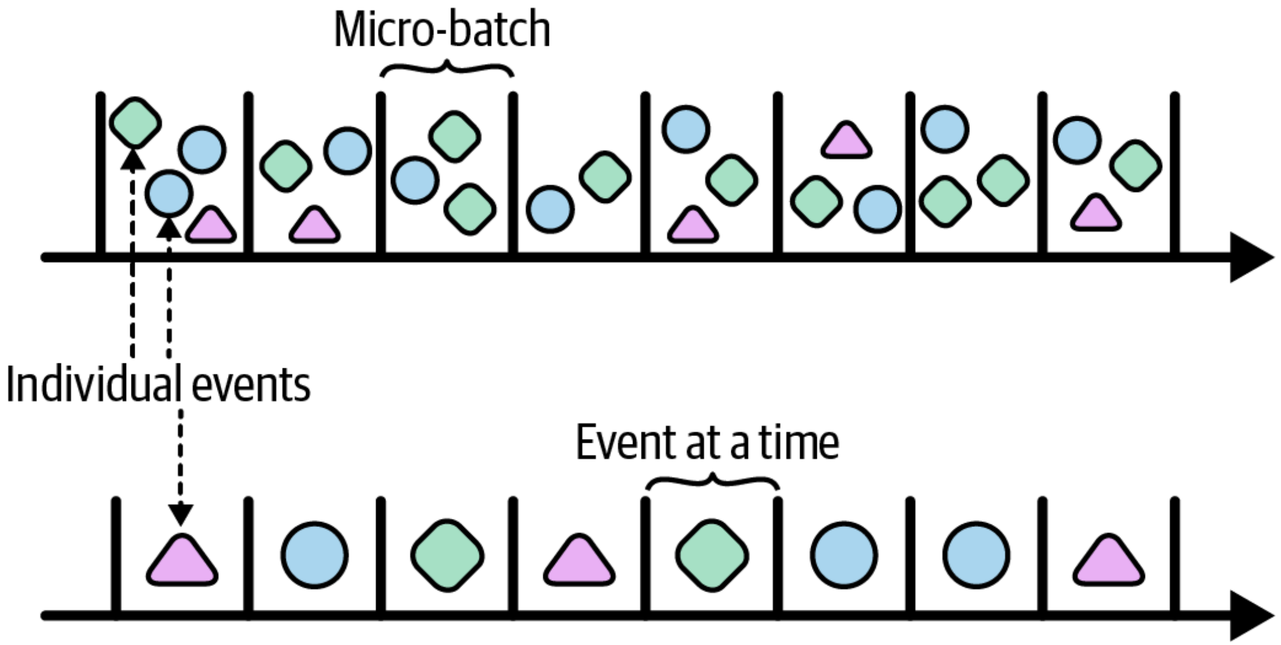
A **KStream** is an abstraction of a **record stream**, where each data record represents a self-contained datum in the unbounded data set.

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

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

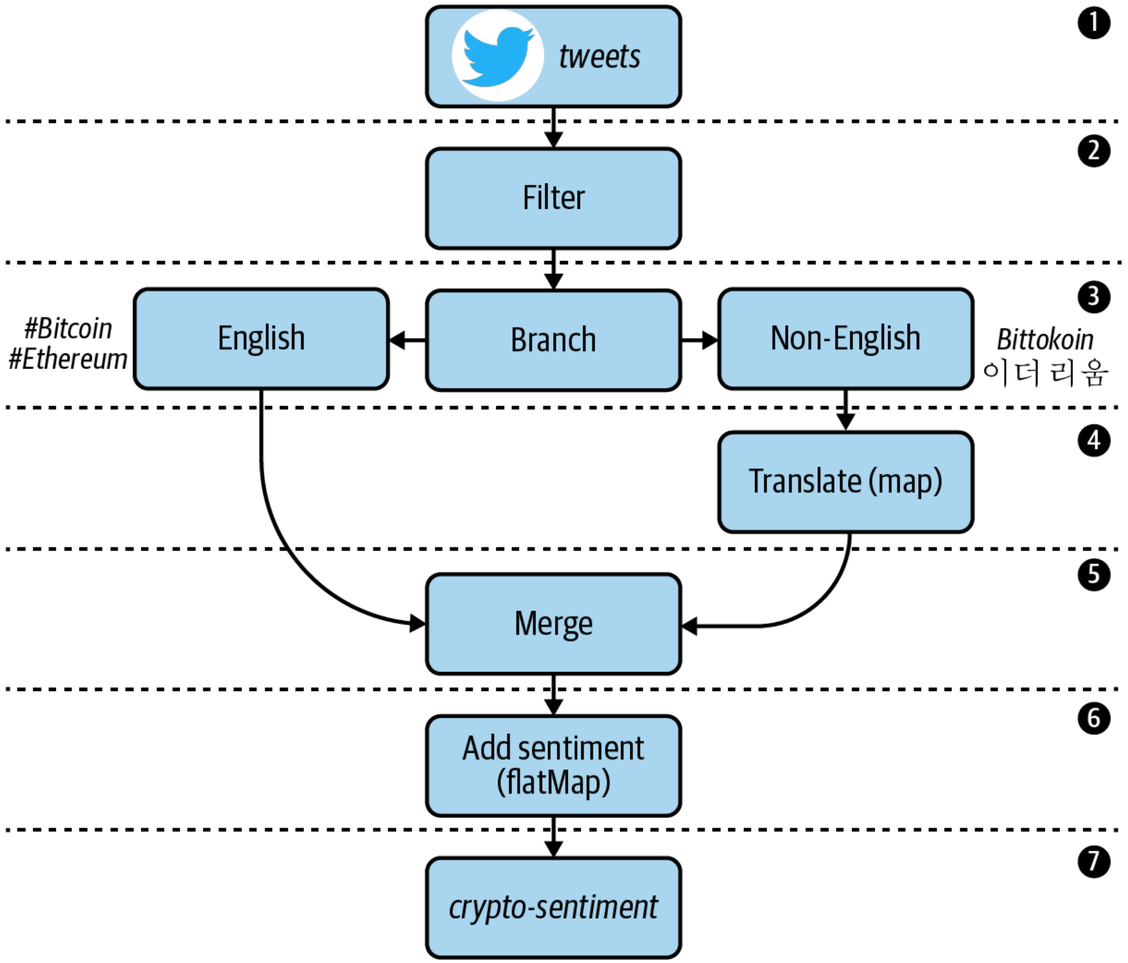
Stateless Processing:
This processing requires no memory of previously seen events.
Each event is consumed, processed, and subsequently forgotten.

Kafka Streams' processing model



Micro-batching involves grouping records into small batches and emitting them to downstream processors at a fixed interval; **event-at-a-time** processing allows each event to be processed at soon as it comes in, instead of waiting for a batch to materialize.
Kafka Streams focuses solely on streaming use cases(this is called a **Kappa architecture**)

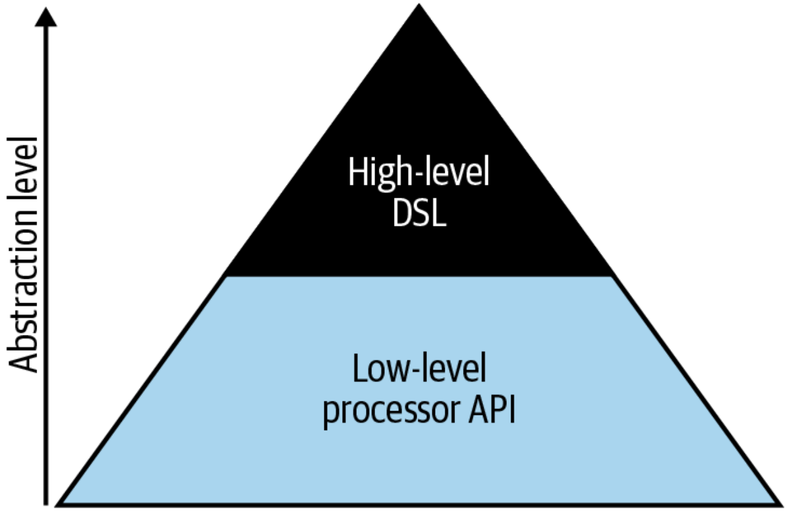
Stateless Kstream - Example



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).



The two APIs you can choose from are:

- The high-level DSL
- The low-level Processor API

DSL Vs Processor API

```
class DslExample {

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

    KStream<Void, String> stream = builder.stream("users"); ❷

    stream.foreach( ❸
        (key, value) -> {
          System.out.println("(DSL) Hello, " + value);
        });

    // omitted for brevity
    Properties config = ...; ❹

    KafkaStreams streams = new KafkaStreams(builder.build(), config); ❺
    streams.start();

    // close Kafka Streams when the JVM shuts down (e.g., SIGTERM)
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❻
  }
}
```

```
public class SayHelloProcessor implements Processor<Void, String, Void, Void> { ❶
  @Override
  public void init(ProcessorContext<Void, Void> context) {} ❷

  @Override
  public void process(Record<Void, String> record) { ❸
    System.out.println("(Processor API) Hello, " + record.value());
  }

  @Override
  public void close() {} ❹
}
```

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 */
    );
```

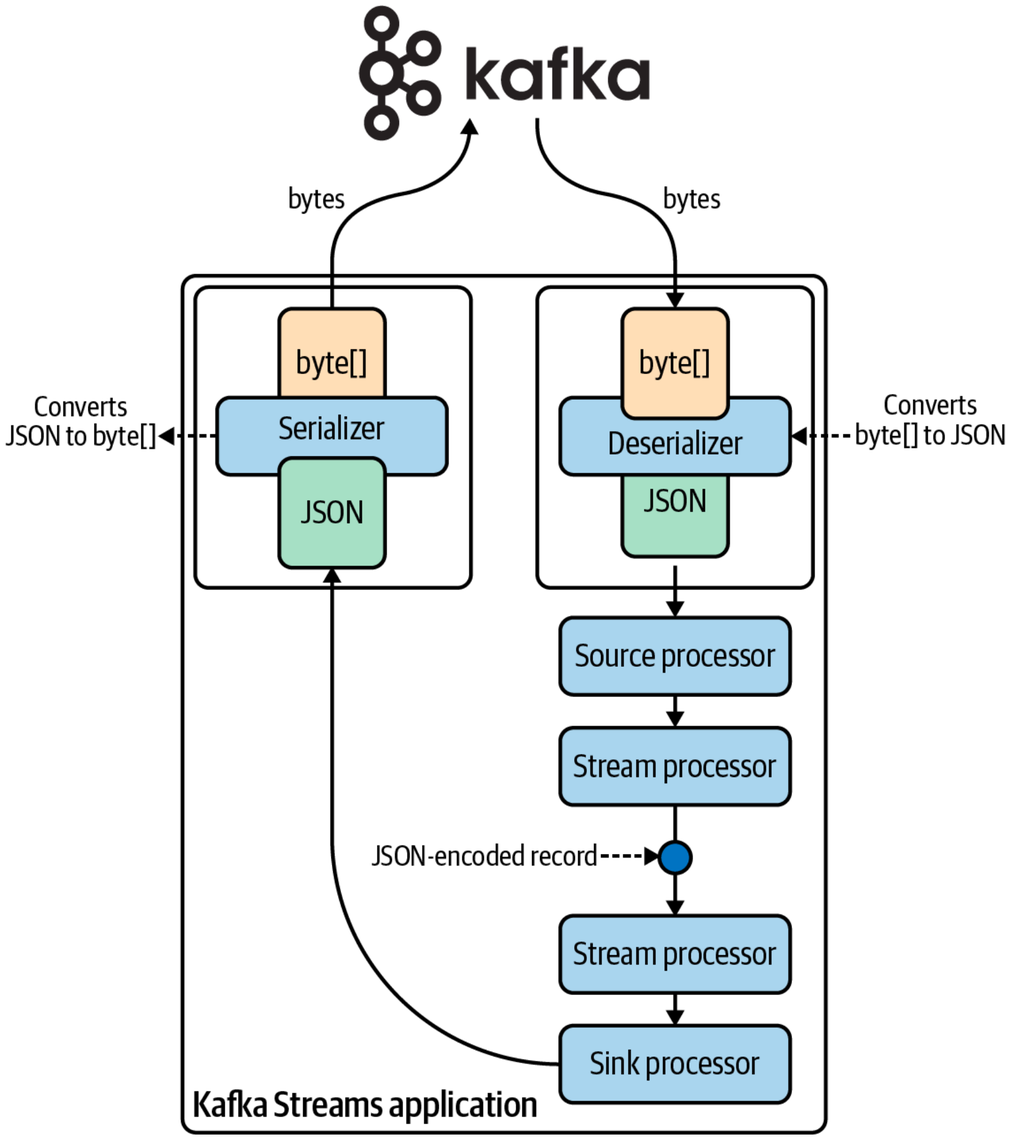
KStream

KStream interface leverages two generics:

- one for specifying the type of keys (K) in our Kafka topic and
- the other for specifying the type of values (V).

Kafka Streams, by default, represents data flowing through our application as byte arrays.

Serialization/Deserialization



Data type	Serdes class
byte[]	`Serdes.ByteArray()`, `Serdes.Bytes()`
ByteBuffer	`Serdes.ByteBuffer()`
Double	`Serdes.Double()`
Integer	`Serdes.Integer()`
Long	`Serdes.Long()`
String	`Serdes.String()`
UUID	`Serdes.UUID()`
Void	`Serdes.Void()`

Custom Serdes

```
public class TweetDeserializer implements Deserializer<Tweet> {
  private Gson gson =
      new GsonBuilder()
          .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE) ❶
          .create();

  @Override
  public Tweet deserialize(String topic, byte[] bytes) { ❷
    if (bytes == null) return null; ❸
    return gson.fromJson(
      new String(bytes, StandardCharsets.UTF_8), Tweet.class); ❹
  }
}
```

```
class TweetSerializer implements Serializer<Tweet> {
  private Gson gson = new Gson();

  @Override
  public byte[] serialize(String topic, Tweet tweet) {
    if (tweet == null) return null; ❶
    return gson.toJson(tweet).getBytes(StandardCharsets.UTF_8); ❷
  }
}
```

```
Gson gson = new Gson();
byte[] bytes = ...; ❶
Type type = ...; ❷
gson.fromJson(new String(bytes), type); ❸
```

The basic method for deserializing byte arrays with ***Gson.***

```
public class Tweet {
    private Long createdAt;
    private Long id;
    private String lang;
    private Boolean retweet;
    private String text;
    // getters and setters omitted for brevity

    }
```

Using Custom serdes

```
KStream<byte[], byte[]> stream = builder.stream("tweets");
stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream"));
```

to:

```
KStream<byte[], Tweet> stream = ❶
  builder.stream(
    "tweets",
    Consumed.with(Serdes.ByteArray(), new TweetSerdes())); ❷

stream.print(Printed.<byte[], Tweet>toSysOut().withLabel("tweets-stream"));
```

Transform a stream

Stateless

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

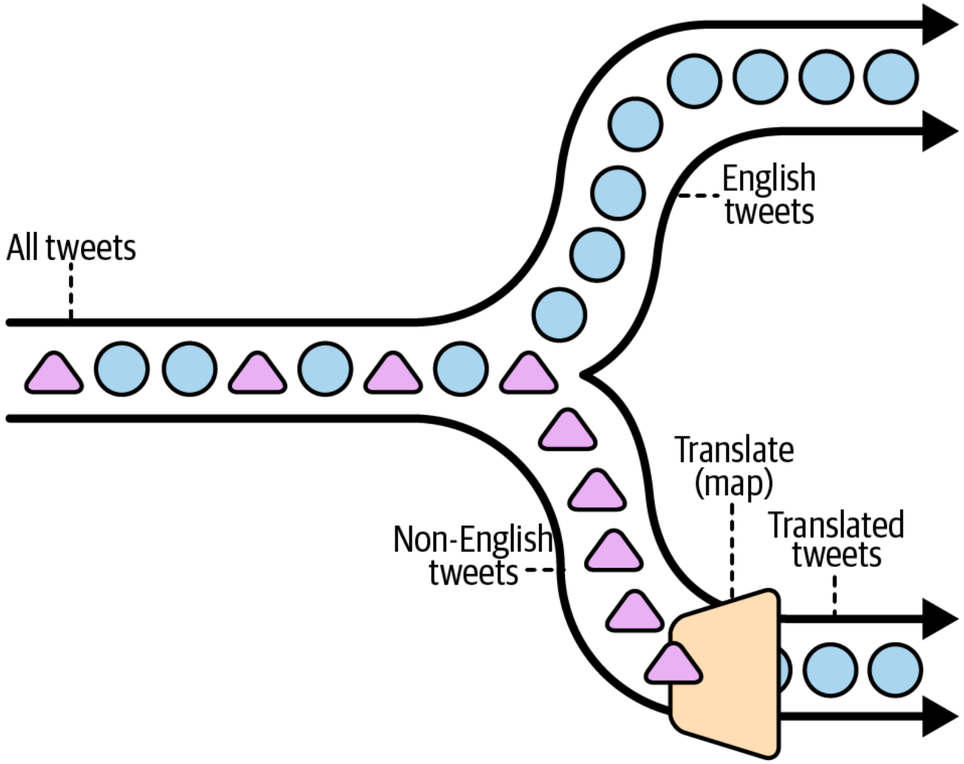
Evaluates a boolean function for each element and drops those for which the function returns true.

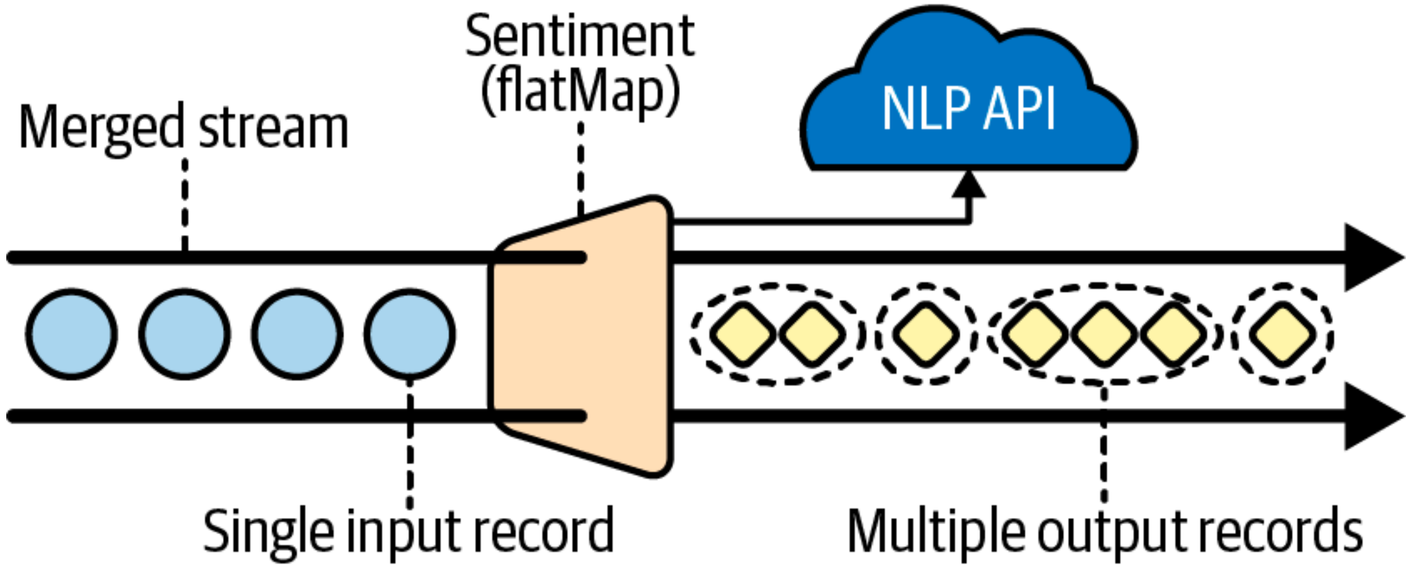
```
KStream<String, Long> stream = ...;

// 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

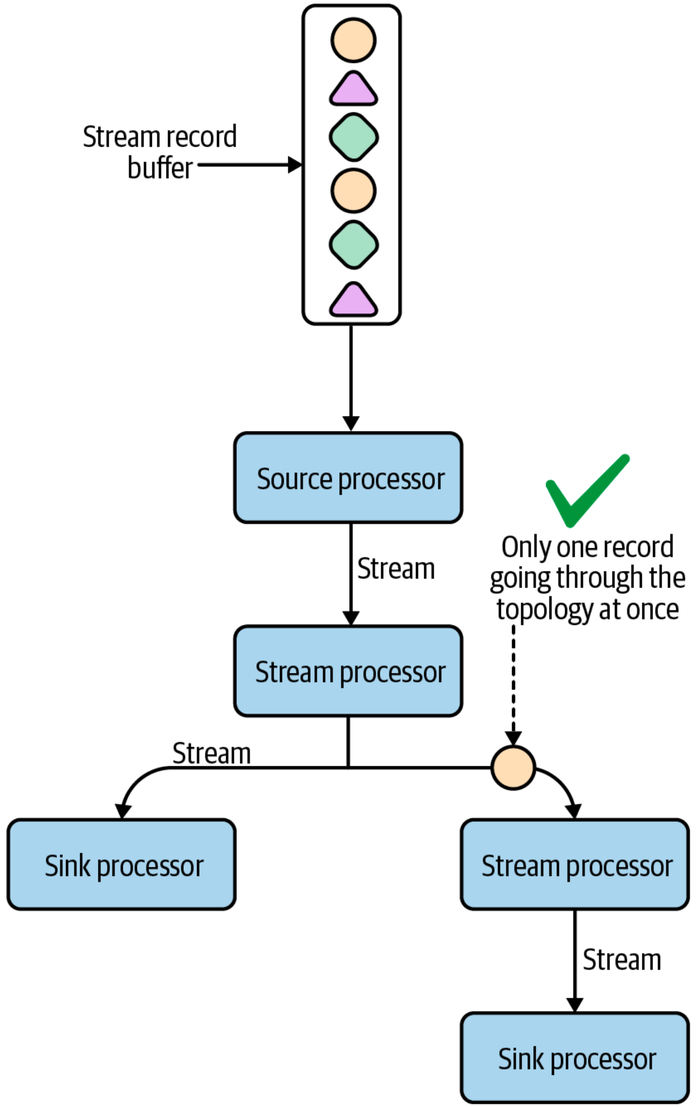
Performs a stateless action on each record, and returns an unchanged stream

```
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);
      }
    });
```

Depth-First Processing



When a new record is received, it is routed through each stream processor in the topology before another record is processed.

This depth-first strategy makes the dataflow much easier to reason about, but also means that slow stream processing operations can block other records from being processed in the same thread.

Benefits of Dataflow Programming

- First, representing the program as a directed graph makes it easy to reason about.
  - Simply find the source and sink processors to determine where data enters and exits your program
- Secondly, standardize the way we frame real-time data processing problems
- Thirdly, Directed graphs are also an intuitive way of visualizing the flow of data for non technical stakeholders.
- Finally, the processor topology, which contains the source, sink, and stream processors, acts as a template that can be instantiated and parallelized very easily across multiple threads and application instances.

Lab : Basic Stateless Stream Transformation – 60 Minutes